ROLF KLEIN

Algorithmische Geometrie

2. AUFLAGE

eXamen.press

Springer

eXamen.press

eXamen.press ist eine Reihe, die Theorie und Praxis aus allen Bereichen der Informatik für die Hochschulausbildung vermittelt.

Rolf Klein

Algorithmische Geometrie

Grundlagen, Methoden, Anwendungen

Mit 213 Abbildungen

Springer

Rolf Klein
Römerstraße 164
53117 Bonn
rolf.klein@uni-bonn.de

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über
http://dnb.ddb.de abrufbar

Ursprünglich erschienen bei Addison-Wesley, 1997

ISBN-10 3-540-20956-5 2. vollst. überarb. Aufl. Springer Berlin Heidelberg New York
ISBN-13 978-3-540-20956-0 2. vollst. überarb. Aufl. Springer Berlin Heidelberg New York

Springer ist ein Unternehmen von Springer Science+Business Media
springer.de

Printed in Germany

Satz: Druckfertige Daten des Autors
Herstellung: LE-TeX Jelonek, Schmidt & Vöckler GbR, Leipzig
Umschlaggestaltung: KünkelLopka GmbH, Heidelberg

Gedruckt auf säurefreiem Papier 33/3142/YL - 5 4 3 2 1

Vorwort

Wie bestimmt man in einer Menge von Punkten am schnellsten zu jedem Punkt seinen nächsten Nachbarn? Wie läßt sich der Durchschnitt von zwei Polygonen effizient berechnen? Wie findet man ein Ziel in unbekannter Umgebung?

Mit diesen und vielen anderen Fragen befaßt sich die Algorithmische Geometrie, ein Teilgebiet der Informatik, dessen Entwicklung vor rund zwanzig Jahren begann und seitdem einen stürmischen Verlauf genommen hat. Aus gutem Grund: Zum einen ist die Beschäftigung mit geometrischen Problemen selbst sehr reizvoll. Oft gilt es, verborgene strukturelle Eigenschaften aufzudecken, bevor ein effizienter Algorithmus entwickelt werden kann. Zum anderen haben die untersuchten Fragen einen direkten Bezug zu realen Problemen in Anwendungsgebieten wie Computergraphik, Computervision, Geographische Informationssysteme oder Robotik.

Dieses Buch gibt eine Einführung in die Algorithmische Geometrie und demonstriert häufig verwendete Techniken an ausgesuchten Beispielen. Es wendet sich an Studierende, die über elementare algorithmische Grundkenntnisse verfügen, und an alle, die beruflich mit geometrischen Fragen zu tun haben oder sich für dieses Gebiet interessieren. Die Grundlage bildet ein Kurs der FernUniversität Hagen im Umfang einer Hauptstudiumvorlesung von vier Semesterwochenstunden; das Buch ist deshalb für ein Selbststudium konzipiert.

Viele Studentinnen, Mitarbeiter und Kolleginnen haben zur Entstehung dieses Buches beigetragen; ihnen allen gebührt mein Dank. Ganz besonders danke ich Anne Brüggemann-Klein, Christian Icking, Wilfried Lange, Elmar Langetepe und Ines Semrau für zahlreiche fruchtbare Diskussionen und die akribische Durchsicht des Manuskripts, Stefan Wohlfeil für die stilistische Gestaltung, Michael Fischer für die Anfertigung der Abbildungen, ohne die ein Kurs über Geometrie kaum denkbar wäre, Gabriele Goetz und Sigrid Timmerbeil für die Textgestaltung und die sorgfältige Erfassung des Manuskripts und Susanne Spitzer vom Verlag Addison-Wesley für die gute Zusammenarbeit!

Schließlich bitte ich Sie, die Leserinnen und Leser, mir Ihre Wünsche, Anregungen und Verbesserungsvorschläge mitzuteilen.

Hagen, im Oktober 1996 Rolf Klein

Vorwort zur zweiten Auflage

Seit dem Erscheinen der ersten Auflage dieses Buches über Algorithmische Geometrie im Verlag Addison Wesley Longman habe ich von vielen Leserinnen und Lesern, und seit meinem Wechsel an die Universität Bonn auch von vielen Hörerinnen und Hörern meiner Vorlesungen, Hinweise auf Fehler im Text und Vorschläge für mögliche Verbesserungen bekommen; ihnen allen gilt mein Dank. In der nun vorliegenden zweiten Auflage, die im Springer-Verlag erscheint, wurden alle bekanntgewordenen Fehler korrigiert, zahlreiche Abschnitte überarbeitet und dabei mehrere Beweise vereinfacht. Insgesamt wurde der Text an die rasch fortschreitende Entwicklung des Gebietes angepaßt, ohne seinen Charakter zu verändern: Nach wie vor ist das Buch zum Selbststudium geeignet. Dazu mögen auch die interaktiven Java-Applets beitragen, die auf unserem Server zur Verfügung stehen und es ermöglichen, mit komplizierten geometrischen Strukturen und Algorithmen selbst zu experimentieren.

Allen Studierenden und Mitarbeitern, die zu diesem Buch beigetragen haben, gebührt mein Dank. Ganz besonders danke ich Herrn Dipl.-Inform. Thomas Kamphans, der die zweite Auflage gesetzt und kritisch gelesen hat, sowie Herrn Dr. Frank Schmidt und Herrn Dr. Hermann Engesser vom Springer-Verlag für die gute Zusammenarbeit!

Sicher ist auch die zweite Auflage noch in mancherlei Hinsicht verbesserungsfähig. Deshalb bitte ich Sie, die Leserinnen und Leser, mir Ihre Wünsche, Anregungen und Vorschläge mitzuteilen.

Bonn, im Dezember 2004

Rolf Klein
`http://web.informatik.uni-bonn.de/I/staff/klein.html`

Inhalt

1

Grundlagen

1.1 Einführung

Bereits im Altertum haben sich Wissenschaftler wie Pythagoras und Euklid mit geometrischen Problemen beschäftigt. Ihr Interesse galt der Entdeckung geometrischer Sachverhalte und deren Beweis. Sie operierten ausschließlich mit geometrischen Figuren (Punkten, Geraden, Kreisen etc.). Erst die Einführung von Koordinaten durch Descartes machte es möglich, geometrische Objekte durch Zahlen zu beschreiben.

Heute gibt es in der Geometrie verschiedene Richtungen, deren unterschiedliche Ziele man vielleicht an folgendem Beispiel verdeutlichen kann. Denken wir uns eine Fläche im Raum, etwa das Paraboloid, das durch Rotation einer Parabel um seine Symmetrieachse entsteht. In der *Differentialgeometrie* werden mit analytischen Methoden Eigenschaften wie die Krümmung der Fläche an einem Punkt definiert und untersucht.

Die *Algebraische Geometrie* faßt das Paraboloid als Nullstellenmenge des Polynoms $p(X, Y, Z) = X^2 + Y^2 - Z$ auf; hier würde man zum Beispiel den Durchschnitt mit einer anderen algebraischen Menge, etwa dem senkrechten Zylinder $(X - x_0^2) + (Y - y_0^2) - r^2$, betrachten und sich fragen, durch welche Gleichungen der Durchschnitt beschrieben wird.[1]

Aufgaben der Algorithmischen Geometrie

Die *Algorithmische Geometrie*, Thema dieses Buches, verfolgt andere Ziele. Ihre Aufgaben bestehen in

- der Entwicklung von *effizienten* und *praktikablen* Algorithmen zur Lösung geometrischer Probleme und in
- der Bestimmung der *algorithmischen Komplexität* geometrischer Probleme.

Anwendungsbereiche

Die untersuchten Probleme haben meistens sehr reale Anwendungshintergründe. Bei der *Bahnplanung für Roboter* geht es darum, eine Bewegung von einer Anfangskonfiguration in eine Endkonfiguration zu planen, die Kollisionen mit der Umgebung vermeidet und außerdem möglichst effizient ist. Wer je eine Leiter durch verwinkelte Korridore getragen hat, kann sich ein Bild von der Schwierigkeit dieser Aufgabe machen. Sie wächst noch, wenn der Roboter seine Umgebung noch gar nicht kennt, sondern sie während der Ausführung erkunden muß.

Beim *computer aided geometric design* (CAGD) kommt es unter anderem darauf an, Durchschnitt und Vereinigung geometrischer Körper schnell zu berechnen. Oder es sollen interpolierende Flächen durch vorgegebene Stützpunkte konstruiert werden.

Bei der Arbeit mit *geographischen Daten*, die in der Regel in Datenbanken gespeichert sind, müssen immer wieder Anfragen beantwortet werden, die sich auf Kombinationen von geometrischen Eigenschaften und Standardmerkmalen beziehen. So möchte man zum Beispiel wissen, welche Städte im Ruhrgebiet vom nächstgelegenen Erholungsgebiet mehr als k Kilometer weit entfernt sind.

Neben diesen großen Anwendungsbereichen gibt es zahlreiche Einzelanwendungen von Methoden der Algorithmischen Geometrie, zum Beispiel beim Fräsen metallener Werkstücke. Ein großer amerikanischer Jeanshersteller verwendet geometrische Verfahren, um die zuzuschneidenden Einzelteile so auf der Stoffbahn zu plazieren, daß möglichst wenig Abfall entsteht.

Komplizierte Anwendungsprobleme wie diese können wir hier nicht behandeln. Oft tritt aber der geometrische Kern eines Problems viel deutlicher zutage, wenn von äußeren Bedingungen abstrahiert wird.

Nehmen wir zum Beispiel an, daß für jedes Haus in einer dünn besiedelten Gegend das nächstgelegene Nachbarhaus ermittelt werden soll. Da die Abstände zwischen den Häusern relativ groß sind, können wir sie uns einfach als Punkte in der Ebene vorstellen. Damit hat unsere Aufgabe folgende Form: Gegeben sind n Punkte in der Ebene. Zu jedem Punkt soll sein nächster Nachbar bestimmt werden. Auf Englisch heißt dieses Problem *all nearest neighbors*. Abbildung 1.1 zeigt ein Beispiel mit 11 Punkten. Die Pfeile weisen jeweils zum nächsten Nachbarn.

all nearest neighbors

Eine mögliche Lösung liegt auf der Hand: Man könnte nacheinander jeden der n Punkte hernehmen und die Abstände zu allen übrigen $n-1$ Punkten bestimmen; ein Punkt mit dem kleinsten

[1] Aus der elementaren Mathematikausbildung sind solche Fragestellungen wohlvertraut: Die Analysis betrachtet Tangenten an Kurven und die Lineare Algebra untersucht Durchschnitte von linearen Teilräumen.

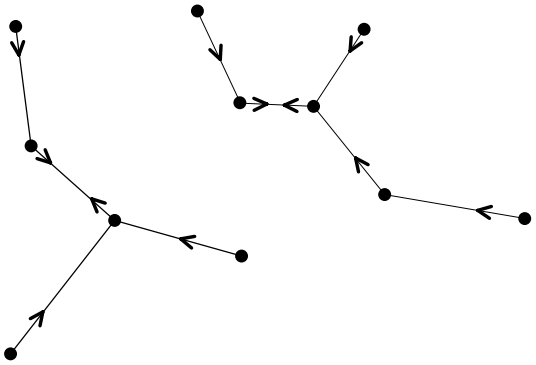


Abb. 1.1 Elf Punkte und ihre nächsten Nachbarn.

Abstand ist dann gesuchter nächster Nachbar des gerade betrachteten Punkts. Bei diesem Vorgehen werden mindestens $\frac{1}{2}n(n-1)$ Berechnungen durchgeführt, weil jedes Punktepaar einmal betrachtet wird. Für $n = 10^5$ wäre ein PC hiermit mehrere Tage beschäftigt!

Für die Algorithmische Geometrie stellt sich hier die Frage, ob das Problem der nächsten Nachbarn sich nicht auch effizienter lösen läßt. Unter *Effizienz* verstehen wir den sparsamen Umgang mit den Ressourcen Rechenzeit und Speicherplatz (in unserem Beispiel liegt das Problem zunächst nur in der Rechenzeit). Im Unterschied zum Vorgehen rein theoretischer Wissenschaften können wir uns aber nicht damit begnügen, die bloße *Existenz* eines effizienten Lösungsverfahrens nachzuweisen; wir wollen vielmehr unsere Algorithmen konkret angeben und im Prinzip auch implementieren können, denn nur so können wir zur Lösung realer Probleme beitragen.

Effizienz

Praktikabilität

Beim Problem *all nearest neighbors* läßt sich der Zeitaufwand verringern, indem man eine strukturelle Eigenschaft des Problems ausnutzt, die das naive Verfahren ganz übersieht: Der nächste Nachbar eines Punktes muß in dessen näherer Umgebung zu finden sein; es sollte also nicht notwendig sein, die Entfernungen zu *allen* anderen Punkten – auch den weit entfernten – zu prüfen. Wir werden sehen, wie man diesen Umstand ausnutzen und mit größenordnungsmäßig $n \log n$ vielen Rechenschritten auskommen kann. Damit sinkt die Rechenzeit für 10^5 Punkte auf dem PC auf wenige Minuten!

Ausnutzen struktureller Eigenschaften

Natürlich wird man fragen, ob sich die Effizienz der Lösung noch weiter verbessern läßt, etwa zu einer Anzahl von Rechenoperationen, die linear in n ist. Wir werden später sehen, daß das nicht möglich ist: Kein Verfahren zur Lösung des Problems der

nächsten Nachbarn kann mit weniger als größenordnungsmäßig $n \log n$ vielen Schritten auskommen.

untere Schranke

Durch beides zusammen – die Angabe einer *unteren Schranke* und die Konstruktion eines Algorithmus, der diese Schranke nicht

algorithmische Komplexität

überschreitet, ist die *algorithmische Komplexität* eines Problems festgelegt. Auch hierhin besteht eine Aufgabe der Algorithmischen Geometrie, die aber mitunter recht schwierig ist. Für zahlreiche wichtige Probleme ist die genaue Komplexität noch nicht bekannt.

Das Gebiet der Algorithmischen Geometrie ist noch recht jung. Seine Entwicklung begann mit der Arbeit von Shamos und Hoey [135] im Jahre 1975 und verlief seitdem recht stürmisch; in der elektronischen Bibliographie *geombib* sind heute mehr als 13000 einschlägige Arbeiten verzeichnet. Inzwischen haben sich

Paradigmen

einige algorithmische Techniken, sogenannte *Paradigmen*, als besonders wichtig herausgestellt, weil man mit ihrer Hilfe nicht nur ein einzelnes Problem, sondern eine Vielzahl von Problemen mit ähnlicher Grundstruktur lösen kann. Solchen allgemeinen Techniken wollen wir besondere Aufmerksamkeit schenken.

Auf der anderen Seite gibt es einzelne Probleme, die sich mit ganz verschiedenen Techniken optimal lösen lassen. Auch hiermit werden wir uns näher beschäftigen.

Voraussetzungen

Ein paar Worte zum Inhalt dieses Buches. Vorausgesetzt werden elementare Kenntnisse im Bereich *Algorithmen und Datenstrukturen*, (z. B. über Heaps, AVL-Bäume und Sortierverfahren mit Laufzeit $O(n \log n)$). Diese Themen werden in allen einschlägigen Textbüchern behandelt. Stellvertretend für viele seien die Bücher von Cormen et al. [36], Güting [69], Mehlhorn [102, 103, 101], Ottmann und Widmayer [116] und Nievergelt und Hinrichs [110] genannt, die auch Abschnitte über geometrische Algorithmen enthalten.

Außerdem werden an einigen Stellen elementare Tatsachen aus Analysis und Linearer Algebra benutzt, die sich in der Literatur leicht finden lassen. Wichtiger als umfangreiche Mathematikkenntnisse ist aber eine gewisse handwerkliche Geschicklichkeit im Umgang mit den gebräuchlichen Methoden, insbesondere beim *Beweisen* von Sachverhalten.

Was an spezielleren Begriffen und Tatsachen in diesem Buch

Gliederung

benutzt wird, ist in Abschnitt 1.2 zusammengestellt; man braucht diesen Abschnitt nicht unbedingt als ersten zu lesen, sondern kann später bei Bedarf dort nachschlagen.

sweep

In Kapitel 2 geht es um das Sweep-Verfahren, eines der wichtigsten Paradigmen in der Algorithmischen Geometrie. Wir werden verschiedene ein- und zweidimensionale Probleme diskutieren, die sich damit sehr elegant lösen lassen. Die Frage nach der Anwen-

dung von *sweep* auf höherdimensionale Probleme führt uns in Kapitel 3 zur Betrachtung höherdimensionaler Datenstrukturen, die neben Anfragen auch das Einfügen und Entfernen von Punkten effizient unterstützen. Hierfür werden allgemeine Dynamisierungstechniken vorgestellt.

Datenstrukturen

Dynamisierung

Kapitel 4 betrachtet einige klassische geometrische Probleme in der Ebene, die mit der Bildung von Durchschnitten oder mit Sichtbarkeit zu tun haben. Berechnet werden die *konvexe Hülle* von Punkten, der *Schnitt von Halbebenen*, das *Sichtbarkeitspolygon* und der *Kern* eines Polygons.

Durchschnitte

Sichtbarkeit

In Kapitel 5 geht es um Distanzprobleme in der Ebene und ihre Lösung mit Hilfe von *Voronoi-Diagramm* und *Delaunay-Triangulation*. Diese – zueinander dualen – Strukturen sind nicht nur innerhalb der Informatik, sondern auch in zahlreichen anderen Wissenschaften von Bedeutung. Ihrer effizienten Berechnung mit verschiedenen algorithmischen Techniken ist Kapitel 6 gewidmet.

Distanzprobleme

Zum Schluß betrachten wir in Kapitel 7 Aufgaben der Bahnplanung für autonome Roboter in unbekannter Umgebung. Am Anfang steht das alte Problem, aus einem unbekannten Labyrinth zu entkommen; am Ende geben wir eine Strategie an, mit deren Hilfe man in einem unbekannten Polygon den Kern finden kann.

Bahnplanung

In jedem Kapitel findet man eine Reihe von *Übungsaufgaben* und, jeweils am Kapitelende, ihre Lösungen. Sie dienen zur Selbstkontrolle beim Lesen, zur Einübung des Stoffs und zu einem geringen Teil auch zur Ergänzung. Ich möchte alle Leser nachdrücklich ermutigen, sich mit den Aufgaben zu beschäftigen. Sie sind mit einem Schwierigkeitsindex von 1 bis 5 versehen. Eine Aufgabe vom Grad 1 sollte (spätestens) nach der Lektüre des Kapitels sofort lösbar sein, beim Grad 5 kann es schon etwas länger dauern, bis sich die richtige Idee einstellt. Die Übungsaufgaben sind mit nebenstehendem Zeichen gekennzeichnet, das auch den Schwierigkeitsindex anzeigt.

Übungsaufgaben

Manche Aufgaben sind von dem in der Praxis besonders häufig vorkommenden Typ „wahr oder falsch?“ Auch wenn man eine solche Aufgabe nicht formal korrekt lösen kann, sollte man wenigstens versuchen, sich anhand von Beispielen oder durch Experimente mit dem Computer eine Meinung zu bilden, bevor man die Lösung nachliest.

Ein einführendes Buch wie dieses kann keinen vollständigen Überblick über die Entwicklung des Faches oder den augenblicklichen Stand der Forschung bieten. Die einzelnen Kapitel enthalten deshalb Hinweise auf weiterführende Bücher zu den jeweils behandelten Themen. Hier folgen ein paar allgemeine Referenzen.

Die Algorithmische Geometrie entstand mit der Arbeit von

Literaturangaben

Shamos und Hoey [135]. Sie bildete eine Grundlage für das Buch von Preparata und Shamos [120], ein Klassiker, der aber an manchen Stellen nicht mehr auf dem neuesten Stand ist. Modern und sehr gut lesbar sind die Lehrbücher von de Berg et al. [38] und von Boissonnat und Yvinec [19]. Ein weiteres modernes Lehrbuch mit vielen Programmierbeispielen stammt von O'Rourke [114]; etwas elementarer geht Laszlo [89] vor. Auch das Buch von Aumann und Spitzmüller [8] behandelt im ersten Teil Probleme der Algorithmischen Geometrie und wendet sich danach eher differentialgeometrischen Fragestellungen zu.

Ein anderer Klassiker ist Edelsbrunners Buch [51]. Es beschäftigt sich besonders mit den kombinatorischen Aspekten der Algorithmischen Geometrie, also mit der Struktur und Komplexität geometrischer Objekte. Diese Fragen stehen auch in den Büchern von Matoušek [100] und Pach und Agarwal [118] im Mittelpunkt. Daneben gibt es mehrere Werke, die sich mit speziellen Themen der Algorithmischen Geometrie befassen. So geht es zum Beispiel im Buch von Mulmuley [107] hauptsächlich um randomisierte geometrische Algorithmen, während Chazelle [26] und Matoušek [99] Eigenschaften der Verteilung geometrischer Objekte untersuchen. Einen enzyklopädischen Überblick über die Algorithmische Geometrie vermitteln die beiden Handbücher, die von Sack und Urrutia [123] sowie von Goodman und O'Rourke [63] herausgegeben wurden. Auch das Kapitel von Yao in [141] behandelt verschiedene Bereiche der Geometrie.

Wer sich für Flächen und ihr Geschlecht oder für andere Themen der klassischen, geometrisch orientierten *Topologie* interessiert, sei auf Stillwell [137] verwiesen. Wissenswertes zur *Graphentheorie* findet man zum Beispiel in Bollobàs [20]. Schließlich leistet eine Formelsammlung wie der Bronstein [22] hin und wieder gute Dienste.

Die hier und in den folgenden Kapiteln zitierten Arbeiten und Bücher stellen nur einen kleinen Ausschnitt der bis heute erschienenen Literatur zur Algorithmischen Geometrie dar.

Tagungsbände

Neue Ergebnisse werden meistens erst auf Tagungen vorgestellt, bevor sie in Zeitschriften erscheinen. Die Tagungsbände (Proceedings) der entsprechenden Konferenzen bilden also eine sehr aktuelle Informationsquelle. Hierzu zählen folgende jährlich stattfindende Tagungen:

- Annual ACM Symposium on Computational Geometry (SCG)
- European Workshop on Computational Geometry (EuroCG)

(Es erscheint meist kein Tagungsband, aber eine Zusammenstellung von Inhaltsangaben)

- Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)
- IEEE Symposium on Foundations of Computer Science (FOCS)
- Annual ACM Symposium on the Theory of Computing (STOC)
- Annual Symposium on Theoretical Aspects of Computer Science (STACS)
  (Tagungsband erscheint in den *Lecture Notes in Computer Science*, Springer-Verlag)
- International Symposium on Algorithms and Computation (ISAAC)
  (Tagungsband erscheint ebenfalls in den *Lecture Notes in Computer Science*)

Andere wichtige Informationsquellen zur Algorithmischen Geometrie bilden die Literaturdatenbank *geombib* und die Mailinglisten *compgeom*.

Wie man hierauf zugreift und zahlreiche andere Informationen findet man im World Wide Web auf folgender Seite: Info im WWW

`http://www.geometrylab.de/Info/`

Unter

`http://www.geometrylab.de/`

finden sich die Java-Applets zur interaktiven Beschäftigung mit geometrischen Strukturen und Algorithmen, auf die in den folgenden Kapiteln im einzelnen hingewiesen wird.

1.2 Ein paar Grundbegriffe

1.2.1 Topologie

Meistens werden wir Objekte im zwei- oder dreidimensionalen euklidischen Raum betrachten; ein paar topologische Grundbegriffe lassen sich aber ebensogut allgemein definieren.

Ein *metrischer Raum* ist eine Menge M zusammen mit einer Metrik d, die je zwei Elementen p und q aus M eine nicht-negative reelle Zahl $d(p, q)$ als *Abstand* zuordnet. Dabei müssen für alle p, metrischer Raum Metrik

q, r aus M folgende Regeln gelten:

$$\begin{aligned} d(p,q) &= 0 \Longleftrightarrow p = q \\ d(p,q) &= d(q,p) \\ d(p,r) &\leq d(p,q) + d(q,r) \end{aligned}$$

Dreiecksungleichung

Die letzte Bedingung wird *Dreiecksungleichung* genannt.

Wir werden uns meistens in $M = \mathbb{R}^2$ oder $M = \mathbb{R}^3$ bewegen. Für zwei Punkte $p = (p_1, \ldots, p_m)$ und $q = (q_1, \ldots, q_m)$ im $\mathbb{R}^m$ ist die *euklidische Metrik*

$$|pq| = \sqrt{\sum_{i=1}^{m} (p_i - q_i)^2}$$

definiert, die wir durch die Schreibweise $|pq|$ von anderen Metriken unterscheiden wollen.

Ist p ein Element (ein „Punkt“) des metrischen Raums (M, d), und $\varepsilon > 0$ eine reelle Zahl, so heißt die Menge $U_\varepsilon(p) = \{q \in M; d(p,q) < \varepsilon\}$ die *ε-Umgebung* von p. Ein Element a von einer Teilmenge A von M heißt ein *innerer Punkt* von A, wenn es ein $U_\varepsilon(a)$ gibt, das ganz in A enthalten ist. Die Menge der inneren Punkte heißt das *Innere* von A. Eine Teilmenge A von M heißt *offen* (in M), falls jeder ihrer Punkte ein innerer Punkt ist. Man nennt die Familie aller offenen Teilmengen auch die *Topologie* von M. Eine Teilmenge B von M heißt *abgeschlossen*, wenn ihr Komplement $B^c = M \setminus B$ offen ist.

offen

Topologie

abgeschlossen

Ist $a \in A$ kein innerer Punkt von A, so muß jede ε-Umgebung von a sowohl Elemente von A (zumindest a selbst) als auch Elemente von A^c enthalten. Allgemein nennt man Punkte mit dieser Eigenschaft *Randpunkte* von A, auch wenn sie selbst nicht zu A gehören. Der Rand von A wird mit ∂A bezeichnet.

Randpunkt

Betrachten wir als Beispiel das Einheitsquadrat $Q = [-1, 1] \times [-1, 1]$ im $\mathbb{R}^2$; siehe Abbildung 1.2.

Es ist abgeschlossen, sein Rand besteht aus der Menge $\{(x, y) \in \mathbb{R}^2; |x| \leq 1$ und $|y| \leq 1$ und $(|x| = 1$ oder $|y| = 1)\}$, und das Produkt $(-1, 1) \times (-1, 1)$ der Intervalle ohne Endpunkte ist sein Inneres. Wenn wir die XY-Ebene als Teilmenge des $\mathbb{R}^3$ auffassen, so ist Q immer noch abgeschlossen. Als Teilmenge des $\mathbb{R}^3$ hat Q aber keine inneren Punkte, denn ganze ε-Kugelumgebungen haben in Q keinen Platz.

Trotzdem wollen wir auch dann über das Innere des Quadrats Q reden können, wenn es sich im $\mathbb{R}^3$ befindet. Allgemein geht man dazu so vor: Für eine Teilmenge N eines metrischen Raumes M sind die ε-Umgebungen von p in N definiert durch

$$U_\varepsilon^N(p) = \{q \in N; d(p,q) < \varepsilon\} = U_\varepsilon(p) \cap N.$$

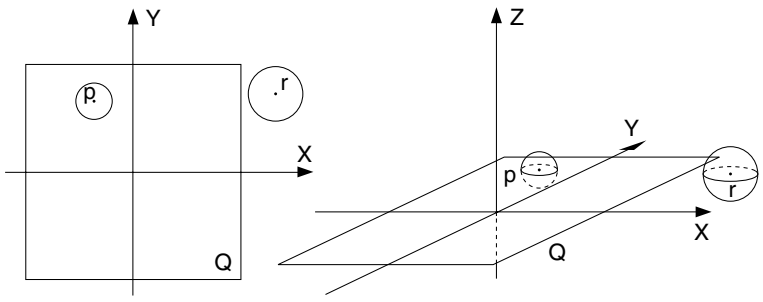


Abb. 1.2 Als Teilmenge des $\mathbb{R}^3$ enthält Q keine inneren Punkte.

Die hierdurch auf N entstehende Topologie heißt die *von M induzierte* oder *Relativtopologie*. In unserem Beispiel erhalten wir als Schnitte der Kugelumgebungen des $\mathbb{R}^3$ mit der XY-Ebene gerade die ε-Kreisumgebungen des $\mathbb{R}^2$ zurück, also die gewöhnliche Topologie der Ebene. Die Punktmenge

$$\{(x, y, z) \in \mathbb{R}^3; |x| < 1 \text{ und } |y| < 1 \text{ und } z = 0\}$$

heißt dann das *relative Innere* von Q. Ebenso sagen wir, daß etwa der Mittelpunkt einer Seite des Quadrats bezüglich der euklidischen Topologie auf $\mathbb{R}$ im relativen Inneren dieses Liniensegments liegt.

beschränkt

Eine Teilmenge A von M heißt *beschränkt*, wenn es eine Zahl D gibt, so daß für alle Punkte a, b in A die Ungleichung $d(a, b) \leq D$ gilt. Das Infimum aller solcher $D \geq 0$ heißt der *Durchmesser* von A. Für nicht beschränkte Mengen ist der Durchmesser unendlich. Eine Teilmenge des $\mathbb{R}^d$ ist *kompakt*, wenn sie beschränkt und abgeschlossen ist.

Durchmesser

kompakt

Mit Hilfe von ε-und δ-Umgebungen läßt sich in gewohnter Weise ausdrücken, wann eine Abbildung von einem metrischen Raum in einen anderen *stetig* ist; die Definition ist aus der Analysis geläufig.

stetig

Manchmal werden wir uns für die *Wege* interessieren, auf denen man von einem Punkt zu einem anderen gelangen kann. Wir verstehen unter einem Weg w von a nach b das Bild einer stetigen Abbildung

Wege

$$f : [0, 1] \to M \text{ mit } f(0) = a, f(1) = b$$

und nennen f eine *Parametrisierung* von w. Zum Beispiel wird durch $f(t) = (\cos 4\pi t, \sin 4\pi t, t)$ im $\mathbb{R}^3$ eine zylindrische Schraubenkurve mit zwei vollen Windungen parametrisiert, die von $a = (1, 0, 0)$ nach $b = (1, 0, 1)$ führt.

Länge eines Weges

Man kann die *Länge eines Weges* durch Approximation von w mit Streckenzügen definieren. Für beliebige Stellen $0 = t_0 < t_1 < \ldots < t_n = 1$ bildet man die Summe über alle Abstände $d(f(t_i), f(t_{i+1}))$, wie in Abbildung 1.3 dargestellt.

Das Supremum dieser Summenwerte über alle möglichen Wahlen von n und von n konsekutiven Stellen im Intervall $[0, 1]$ bezeichnet man als die *Länge* des Weges w. Man nennt w *rektifizierbar*, wenn dieser Wert endlich ist.

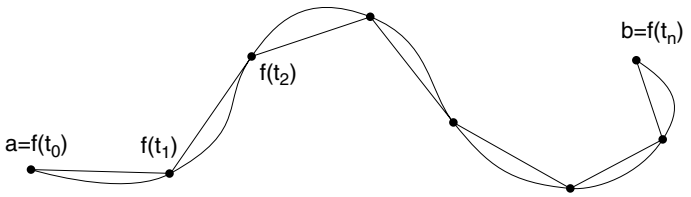


Abb. 1.3 Durch $\sum_{i=0}^{n-1} d(f(t_i), f(t_{i+1}))$ wird die Länge des Weges w von unten approximiert.

Integralformel für die Weglänge

Für eine exakte Berechnung der Länge des Weges eignet sich diese Definition schlecht. Für Wege im $\mathbb{R}^3$, die durch $f(t) = (x(t), y(t), z(t))$ parametrisiert sind, kann man die *Integralformel für die Weglänge*

$$Länge(w) = \int_0^1 \sqrt{x'(t)^2 + y'(t)^2 + z'(t)^2}\, dt$$

verwenden, falls alle drei Koordinatenfunktionen stetig differenzierbar sind.

Übungsaufgabe 1.1 Gegeben sei eine zylindrische Schraubenfeder mit Radius 1 und Höhe 1, wie in Abbildung 1.4 dargestellt. Wenn man die Feder mit einem Gewicht belastet, ohne sie dabei zu überdehnen, nimmt dann ihr Radius ab?

wegzusammenhängend

Gebiet

Eine Teilmenge A von M heißt *wegzusammenhängend*, wenn je zwei Punkte von A durch einen Weg verbunden werden können, der ganz in A verläuft. Mengen im $\mathbb{R}^2$, die offen und wegzusammenhängend sind, werden *Gebiete* genannt. Im folgenden lassen wir das Präfix *weg* weg und reden einfach vom *Zusammenhang*.

Zusammenhangskomponente

Ist A nicht zusammenhängend, können wir für jeden Punkt a von A die *Zusammenhangskomponente*

$$Z(a) = \{b \in A;\ \text{es gibt einen Weg von } a \text{ nach } b \text{ in } A\}$$

bilden. $Z(a)$ ist die größte zusammenhängende Teilmenge von A, die a enthält.

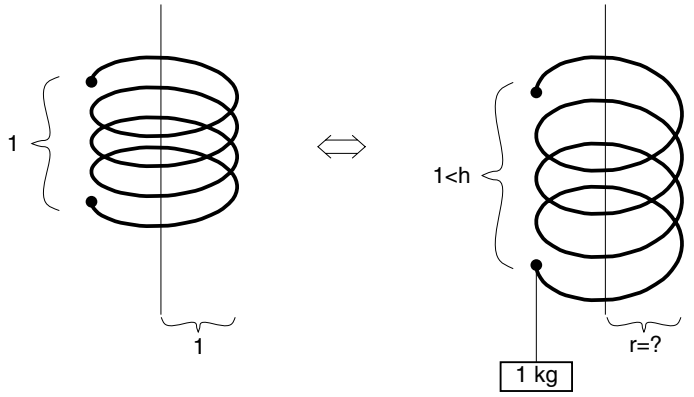


Abb. 1.4 Eine Schraubenfeder in belastetem und unbelastetem Zustand.

Die Zusammenhangskomponenten von zwei Punkten a, b von A sind entweder identisch oder disjunkt, wie die nachfolgende Übungsaufgabe 1.2 zeigt. Daher gibt es eine eindeutige Zerlegung

$$A = \bigcup_{i \in I} Z_i$$

in (möglicherweise unendlich viele) paarweise disjunkte Zusammenhangskomponenten $Z_i = Z(a_i)$.

Übungsaufgabe 1.2 Für zwei Punkte a, b aus einer Teilmenge A eines metrischen Raumes M definieren wir die Relation

$$a \sim b :\Longleftrightarrow \text{ es gibt einen Weg von } a \text{ nach } b, \text{ der ganz in } A \text{ verläuft.}$$

Man gebe einen formalen Beweis für diese Behauptungen:

(i) Die Relation $\sim$ ist eine *Äquivalenzrelation* (d. h. sie ist reflexiv, symmetrisch und transitiv).

(ii) Für zwei Punkte $a, b \in A$ gilt entweder $Z(a) = Z(b)$ oder $Z(a)$ und $Z(b)$ sind disjunkt.

(iii) A besitzt eine eindeutige Zerlegung in Zusammenhangskomponenten.

Ist etwa M der $\mathbb{R}^2$ ohne die X- und Y-Achsen, so besteht M aus genau 4 Zusammenhangskomponenten, nämlich den 4 randlosen Quadranten.

Ein Weg w, der von a wieder zu a führt, heißt *geschlossen*; er wird durch eine Abbildung f mit $f(0) = a = f(1)$ parametrisiert.

einfacher Weg

Man nennt solch ein w einen *einfachen Weg*, wenn er eine Parametrisierung besitzt, die auf dem Intervall $[0, 1)$ injektiv ist, d. h. wenn kein Punkt außer a mehrfach besucht wird.

Jordanscher Kurvensatz

Für einen einfachen geschlossenen Weg w in der Ebene besagt der *Jordansche Kurvensatz*, daß $\mathbb{R}^2 \setminus w$ aus genau zwei Gebieten besteht, einem von w umschlossenen inneren und einem unbeschränkten äußeren Gebiet, dessen Rand ebenfalls gleich w ist. Daß dieser anschaulich „klare“ Sachverhalt recht schwer zu beweisen ist, hat eine Ursache darin, daß die Familie aller Wege auch Mitglieder enthält, die man sich nur schwer anschaulich vorstellen kann (zum Beispiel die sogenannte Hilbertsche Kurve, die ganze Flächenstücke füllt). Wir werden mit solchen Phänomenen in diesem Buch nichts zu tun bekommen.

einfach-zusammenhängend

Wir nennen eine zusammenhängende Teilmenge A des $\mathbb{R}^2$ *einfach-zusammenhängend*, wenn für jeden ganz in A verlaufenden einfachen geschlossenen Weg sein inneres Gebiet ebenfalls in A enthalten ist. In Abbildung 1.5 ist A einfach-zusammenhängend, B aber nicht.

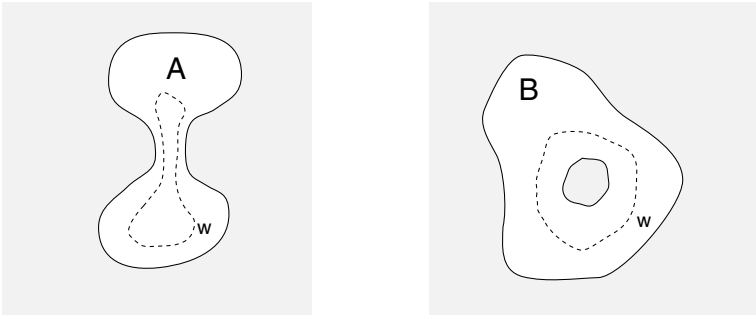


Abb. 1.5 A ist einfach-zusammenhängend, B nicht.

1.2.2 Graphentheorie

Graph

Ein *Graph* besteht aus einer endlichen Knotenmenge V und einem System E von Kanten, das aus Elementen von $V \times V$ besteht. Dabei bedeutet $(p, q) \in E$, daß p und q mit einer Kante verbunden sind. Bei *gerichteten* Graphen ist die Kante $e = (p, q)$ von p nach q orientiert. Fast immer werden unsere Graphen ungerichtet sein. Es ist nicht von vornherein verboten, daß ein Graph Kanten (p, p) von p nach p enthält, sogenannte Schlingen, oder mehrfache Kanten zwischen zwei Knoten. In diesem Fall ist E eine Multimenge,

die mehrfache Elemente $(p,q),(q,p),(p,q),\ldots$ enthalten kann. Ein Graph, bei dem diese Phänomene nicht auftreten, heißt *schlicht*.

schlicht

Man kann einen Graphen $G=(V,E)$ im $\mathbb{R}^2$ oder auf einer anderen Fläche *geometrisch realisieren*, indem man seine Knoten auf paarweise verschiedene Punkte abbildet und seine Kanten auf einfache Wege, die die entsprechenden Punkte verbinden und unterwegs keine solchen Punkte besuchen (wir tun das jedesmal, wenn wir einen Graphen zeichnen; einen Graphen „schön“ zu zeichnen ist übrigens ein wichtiges und schwieriges Problem, zu dem es eine eigene Konferenzreihe und ein Buch von Di Battista et al. [45] gibt; siehe auch z. B. Formann et al. [58], Wagner und Kaufmann [84] und Gutwenger et al. [70]). Das Ergebnis nennt man einen *geometrischen Graphen*.

geometrische Realisierung

geometrischer Graph

In der Wahl der Punkte, auf die die Knoten abgebildet werden, und in der Festlegung der Wege steckt viel Freiheit. Wir nennen zwei Realisierungen eines Graphen auf derselben Fläche *äquivalent*, wenn sie durch stetige Verformung der Kantenwege und Verschiebung der Knoten ineinander überführt werden können, ohne daß jemals ein Kantenweg über einen Knoten „hinweggehoben“ wird. Diese Äquivalenzrelation wird *Homotopie* genannt.

Homotopie

In Abbildung 1.6 sieht man fünf verschiedene Realisierungen desselben Graphen. Nur (i) und (ii) sind in der Ebene äquivalent. Wir können uns aber auch vorstellen, daß dies geometrische Realisierungen auf der Oberfläche einer *Kugel* sind. Dort ist die bei der Realisierung in der Ebene unbeschränkte Fläche beschränkt, wie alle übrigen auch. Auf der Kugel sind die Realisierungen (ii) und (iii) äquivalent: Man kann ja die im Bild linke Kante über die Rückseite der Kugel nach rechts führen. Auch (iii) und (iv) sind auf der Kugel äquivalent, denn man kann r und s über die Kugelrückseite nach oben schieben. Realisierung (v) ist aber zu keiner der anderen äquivalent: Wenn wir den geschlossenen Weg (p,q,s,r,p) mit dieser Orientierung betrachten, liegt t in (i) bis (iv) auf der linken Seite, in (v) aber rechts. Eine Verschiebung von der linken zur rechten Seite ist durch erlaubtes Deformieren nicht zu erreichen.

In diesem Buch interessieren wir uns besonders für solche geometrischen Graphen, bei denen sich keine Kanten kreuzen. Solch ein geometrischer Graph heißt *kreuzungsfrei*. Man nennt einen (abstrakten) Graphen *planar*, wenn er sich im $\mathbb{R}^2$ kreuzungsfrei geometrisch realisieren läßt. Sei nun G ein kreuzungsfreier geometrischer Graph in der Ebene. Wir können G als die Punktmenge ansehen, die aus den Knotenpunkten und den Kantenwegen besteht (im folgenden reden wir wieder von Knoten und Kanten wie bei abstrakten Graphen). Die Zusammenhangskomponenten der Menge $\mathbb{R}^2 \setminus G$ heißen die *Flächen des Graphen*. Sie sind Gebiete

kreuzungsfrei

Flächen des Graphen

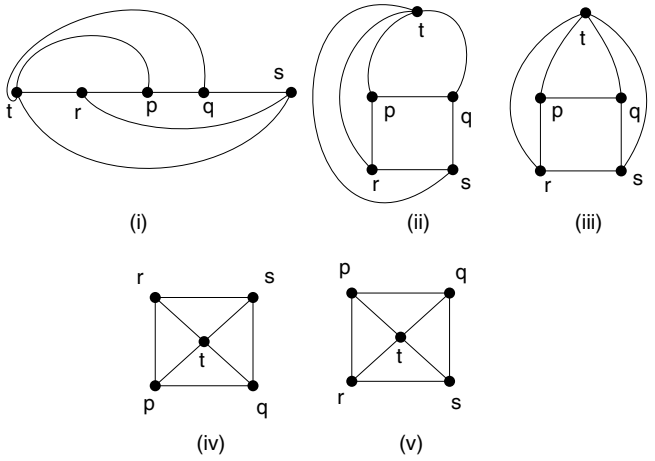


Abb. 1.6 Fünf Realisierungen desselben Graphen.

im $\mathbb{R}^2$. Weil der Graph beschränkt ist, ist genau eine Fläche unbeschränkt; siehe Abbildung 1.7. Ein zusammenhängender Graph heißt ein *Baum*, wenn er keine beschränkten Flächen hat.

Man bezeichnet mit

- v die Anzahl der Knoten (*vertex*)
- e die Anzahl der Kanten (*edge*)
- f die Anzahl der Flächen (*face*)
- c die Anzahl der Zusammenhangskomponenten (*connected component*).

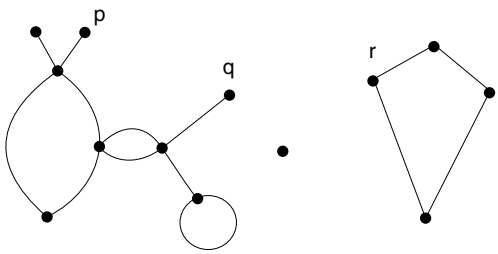


Abb. 1.7 Ein geometrischer Graph mit 13 Knoten, 14 Kanten, 5 Flächen und 3 Zusammenhangskomponenten.

Für den Graphen in Abbildung 1.7 ist $v = 13$, $e = 14$, $f = 5$ und $c = 3$. Allgemein gilt für diese Größen eine wichtige Beziehung, die *Eulersche Formel*.

Theorem 1.1 *Sei G ein kreuzungsfreier geometrischer Graph in der Ebene. Dann gilt*

$$v - e + f = c + 1.$$

Eulersche Formel

Beweis. Wir konstruieren G aus dem leeren Graphen durch sukzessive Hinzunahme von Knoten und Kanten und zeigen durch Induktion, daß die Eulersche Formel stets erfüllt ist. Für den leeren Graphen ist $v = e = c = 0$ und $f = 1$, also stimmt die Behauptung.

Wenn wir einen Knoten hinzunehmen, treten zwei Fälle auf: Liegt der Knoten im relativen Inneren einer schon vorhandenen Kante, erhöhen sich v und e jeweils um 1. Liegt der neue Knoten im Inneren einer Fläche, wachsen v und c um 1. Ansonsten ändert sich nichts.

Bei Hinzunahme einer Kante (mit schon vorhandenen Endknoten) gibt es auch zwei Fälle:

Wenn die Kante zwei bisher disjunkte Zusammenhangskomponenten miteinander verbindet (z. B. die Kante (q, r) in Abbildung 1.7), wächst e um 1 und c fällt um 1; sonst bleibt alles unverändert. Wenn die neue Kante dagegen Knoten *derselben* Zusammenhangskomponente verbindet, etwa p und q, entsteht eine neue Fläche. Folglich wachsen e und f beide um 1, und die Formel gilt weiterhin. Weil sich jeder Graph aus einem kleineren Graphen durch Hinzunahme eines Knotens oder einer Kante gewinnen läßt, ergibt sich die Behauptung. □

Die Eulersche Formel hat viele Konsequenzen. Sie impliziert zum Beispiel, daß die Anzahl der Flächen nicht davon abhängt, *wie* man einen planaren Graphen kreuzungsfrei zeichnet; vgl. Abbildung 1.6. Wichtiger noch ist folgende Aussage, die wir später wiederholt anwenden werden: Bei „anständigen“ kreuzungsfreien geometrischen Graphen sind die Anzahlen von Knoten, Kanten und Flächen einander in etwa gleich. Diese Behauptung soll im folgenden präzisiert werden.

Unter dem *Grad eines Knotens* versteht man die Anzahl der Kanten, die diesen Knoten zum Endpunkt haben (bei gerichteten Graphen kann man noch zwischen den ein- und den ausgehenden Kanten unterscheiden).

Korollar 1.2 *Sei G ein kreuzungsfreier geometrischer Graph, dessen Knoten alle mindestens den Grad 3 haben. Dann gilt*

$$\begin{aligned} v &\leq \frac{2}{3}e \\ v &\leq 2(f - c - 1) < 2f \\ e &\leq 3(f - c - 1) < 3f. \end{aligned}$$

Ferner besteht der Rand einer Fläche im Mittel aus höchstens 6 Kanten.

Beweis. Von jedem Knoten gehen mindestens drei Kanten aus. Wenn wir über alle Knoten aufsummieren, wird dabei jede Kante zweimal gezählt. Also ist $2e \geq 3v$. Damit ist die erste Behauptung bewiesen. Löst man diese Abschätzung nach e und nach v auf und setzt sie in die Eulersche Formel ein, ergeben sich sofort die nächsten beiden Aussagen. Sei nun m_i die Anzahl der Kanten auf dem Rand der i-ten Fläche, für $1 \leq i \leq f$. Wir erhalten dann für die mittlere Kantenzahl

$v - e + f = c + 1$

$$m = \frac{1}{f} \sum_{i=1}^{f} m_i$$

die Abschätzung

$$mf = \sum_{i=1}^{f} m_i \leq 2e < 6f,$$

da jede Kante auf dem Rand von höchstens zwei Flächen liegen kann. □

Die Anzahl der Knoten und Kanten kann also die Anzahl der Flächen nicht wesentlich übersteigen. Für *schlichte* Graphen gilt auch eine Art Umkehrung.

Korollar 1.3 *G sei kreuzungsfrei, nichtleer und schlicht, und alle Knoten haben mindestens den Grad 3. Dann ist $f \leq \frac{2}{3}e$ und $f < 2v$.*

Der Beweis ist Gegenstand von Übungsaufgabe 1.6.

Übungsaufgabe 1.3 Unter K_5 versteht man den Graphen mit fünf Knoten, bei dem jeder Knoten mit jedem anderen durch genau eine ungerichtete Kante verbunden ist. Kann man K_5 kreuzungsfrei

(i) im $\mathbb{R}^2$

(ii) auf der Oberfläche des Torus (Fahrradschlauch) geometrisch realisieren?

Übungsaufgabe 1.4 Unter $K_{3,3}$ versteht man den Graphen mit 6 Knoten, die in zwei Teilmengen A, B zu je 3 Knoten aufgeteilt sind. Jeder Knoten aus A ist mit jedem Knoten aus B durch eine Kante verbunden; andere Kanten gibt es nicht. Kann man $K_{3,3}$ kreuzungsfrei im $\mathbb{R}^2$ realisieren?

Wie die vorangegangenen Übungsaufgaben 1.3 und 1.4 zeigen, sind die beiden Graphen K_5 und $K_{3,3}$ nicht planar. Ein berühmtes Resultat von Kuratowski besagt, daß ein beliebiger Graph genau dann planar ist, wenn er keinen Teilgraphen enthält, der nach eventuellem Entfernen von Knoten vom Grad 2 mit K_5 oder $K_{3,3}$ übereinstimmt. Hieraus lassen sich Algorithmen zum Planaritätstest gewinnen; siehe Diestel [47].

Planaritätstest

Der in Übungsaufgabe 1.3 (ii) angesprochene Torus ist topologisch nichts anderes als eine Kugel mit einem aufgesetztem Henkel. Offenbar kann man jeden Graphen kreuzungsfrei auf einer solchen Fläche realisieren, wenn man genügend viele Henkel verwendet. Die minimale Henkelzahl nennt man das *Geschlecht* des Graphen. Seine Bestimmung ist ein NP-vollständiges Problem; siehe Thomassen [138].

Geschlecht

Die Aussagen von Theorem 1.1, Korollar 1.2 und Korollar 1.3 stimmen übrigens auch für kreuzungsfreie Graphen auf der Kugeloberfläche. Die Beweise sind identisch.

Übungsaufgabe 1.5 Sei S eine Menge von Punkten in der Ebene. Zwei Punkte aus S werden mit einem Liniensegment verbunden, wenn ein Punkt nächster Nachbar des anderen ist.

(i) Man zeige, daß der resultierende Graph G kreuzungsfrei ist.

(ii) Angenommen, jeder Punkt in S hat genau einen nächsten Nachbarn in S. Man beweise, daß es in G dann keinen geschlossenen Weg gibt und deshalb G aus endlich vielen Bäumen besteht.

(iii) Der in Abbildung 1.1 abgebildete Graph erfüllt die Voraussetzung von (ii). Bei ihm wurden die Kanten so orientiert, daß sie zum nächsten Nachbarn zeigen. Der Graph besteht aus zwei Bäumen und enthält zwei Kanten, die in beide Richtungen orientiert sind. Beruht diese Übereinstimmung auf Zufall?

Ein wichtiges Prinzip ist das der *Dualität*. Sei G ein kreuzungsfreier, nichtleerer, zusammenhängender Graph auf der Kugeloberfläche. Wir konstruieren einen Graphen G^* folgendermaßen:

Dualität

- Wähle einen Punkt p_F^* im Inneren jeder Fläche F von G. Diese Punkte sind die Knoten von G^*.
- Für jede Kante e von G mit angrenzenden Flächen F und F' verbinde p_F^* mit $p_{F'}^*$ mit einer Kante e^*, die nur e und sonst keine andere Kante kreuzt.

Abbildung 1.8 zeigt ein Beispiel. Der Graph G^* ist gestrichelt gezeichnet. Er ist nach Definition kreuzungsfrei. Die duale Kante e^* von e ist eine Schleife am Knoten q^*, weil an e auf beiden Seiten dieselbe Fläche angrenzt.

In Abbildung 1.8 hätten wir z. B. die unterste Kante (q^*, p^*) ebensogut „obenherum“ führen können; insofern ist diese ebene Darstellung von G^* nicht eindeutig. Als geometrischer Graph auf der Kugeloberfläche ist G^* durch unsere Konstruktionsvorschriften aber eindeutig bestimmt (natürlich nur bis auf Verformungsäquivalenz). Man nennt G^* den *dualen Graphen* von G. Diese Bezeichnung ist berechtigt, denn $(G^*)^*$ ist wieder zu G äquivalent!

dualer Graph

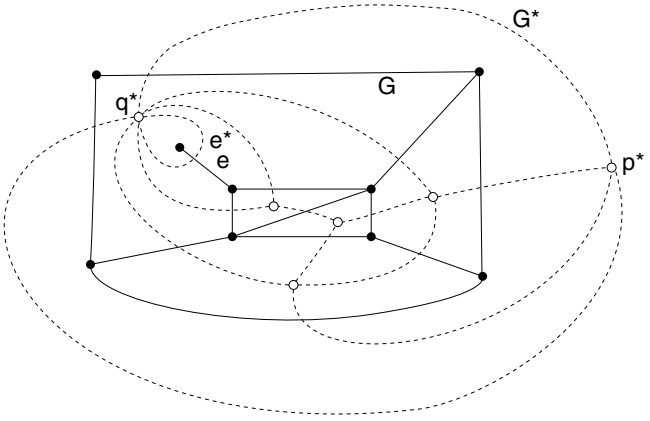


Abb. 1.8 Ein Graph und sein dualer Graph.

Warum interessieren wir uns für duale Graphen? Zum einen, weil wir später auf zwei geometrische Graphen in der Ebene stoßen werden (*Voronoi-Diagramm* und *Delaunay-Triangulation*), die auf den ersten Blick ganz unterschiedlich definiert sind, in Wahrheit aber zueinander dual sind.

Und zum anderen, weil *Dualisierung* prinzipiell eine nützliche Technik darstellt, um Unbekanntes auf Bekanntes zurückzuführen. Nehmen wir zum Beispiel an, daß jede Fläche von G mindestens 3 Kanten in ihrem Rand enthält. Für G^* bedeutet das, daß jeder Knoten mindestens den Grad 3 hat. Also folgt aus Korollar 1.2, daß für die Kanten und Flächen von G^* die Abschätzung $e^* < 3f^*$ gilt. Wegen $e^* = e$ und $f^* = v$ haben wir also ohne Anstrengung bewiesen, daß für G die Abschätzung $e < 3v$ gilt (Tip für die Lösung von Übungsaufgabe 1.6).

Dualisierung als Beweistechnik

Übungsaufgabe 1.6 Sei G ein nichtleerer, kreuzungsfreier, schlichter geometrischer Graph in der Ebene, dessen Knoten alle einen Grad ≥ 3 haben.

(i) Man zeige, daß $3f \leq 2e$ gilt.

(ii) Man beweise, daß $e < 3v$ und $f < 2v$ gelten.

(iii) Gilt eine dieser Behauptungen auch ohne die Voraussetzung der Schlichtheit?

Wir werden meistens mit solchen kreuzungsfreien geometrischen Graphen zu tun haben, deren Kanten geradlinig verlaufen. Solche Graphen sind immer schlicht. Zu ihrer Speicherung kann man außer der bekannten Adjazenzliste oder der Adjazenzmatrix eine Datenstruktur namens *doubly connected edge list* (*DCEL*) verwenden, die der geometrischen Struktur besser gerecht wird.

doubly connected edge list (*DCEL*)

Zu jeder Kante $e = (p, q)$ mit den angrenzenden Flächen F und F' speichert man außer den Namen von p, q, F, F' auch Verweise auf die erste Kante e' mit Endpunkt p, die im Uhrzeigersinn auf e folgt, und auf die erste Kante e'' mit Endpunkt q im Uhrzeigersinn hinter e; siehe Abbildung 1.9.

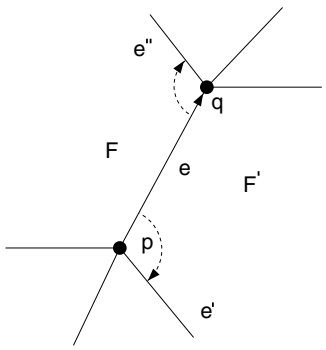


Abb. 1.9 Die Kante e' ist die erste im Uhrzeigersinn hinter e am Punkt p, entsprechend e'' am Punkt q.

Unter Benutzung der Verweise auf die Nachfolgerkanten kann man effizient

- im Uhrzeigersinn alle Kanten mit vorgegebenem Endpunkt und
- gegen den Uhrzeigersinn alle Kanten auf dem Rand einer vorgegebenen Fläche

ausgeben, wenn eine Startkante bekannt ist.

Wenn man zusätzlich für jede Kante Verweise auf ihre Nachfolgerkanten *entgegen* dem Uhrzeigersinn speichert, erhält man die *quad edge data structure* (*QEDS*); siehe Guibas und Stolfi [68]. Mit ihrer Hilfe lassen sich die Ränder von Flächen und die von einem Knoten ausgehenden Kanten bequem in beiden Richtungen durchlaufen. Ein großer Vorteil dieser Datenstruktur: Hat man die *QEDS* eines Graphen G, kennt man auch die *QEDS* seines dualen

quad edge data structure (*QEDS*)

Graphen G^*: Man braucht nur jede Kante durch ihre duale Kante zu ersetzen. Alle Verweise bleiben erhalten.

1.2.3 Geometrie

geometrische Objekte

Die einfachsten *geometrischen Objekte* sind *Punkte* $p = (p_1, \ldots, p_d)$ im $\mathbb{R}^d$ und *Liniensegmente* $pq = \{p + a(q - p);\ 0 \leq a \leq 1\}$.[2] Ein Weg w, der aus einer endlichen Folge von Liniensegmenten $pq, qr, \ldots, tu, uv$ besteht, heißt eine *polygonale Kette*. Ist w eine einfache, geschlossene polygonale Kette in der Ebene, so heißt das von w umschlossene innere Gebiet zusammen mit w ein

Polygon

Polygon, P. Sein Rand, ∂P, ist gerade w.[3] Die Liniensegmente in w heißen die *Kanten* von P, ihre Endpunkte heißen *Ecken* von P (wir können P auch als kreuzungsfreien geometrischen Graphen auffassen und müßten die Ecken dann „Knoten“ nennen; der Begriff „Ecke“ ist aber gebräuchlicher).

Polygone treten unter anderem als Grundrisse von Räumen auf, in denen sich Roboter bewegen. Man interessiert sich zum Beispiel dafür, welcher Teil des Polygons P von einem Punkt $p \in P$ aus sichtbar ist. Hierbei heißt ein Punkt $q \in P$ von p aus *sichtbar*, wenn das Liniensegment pq ganz in P enthalten ist. Die Menge

Sichtbarkeitspolygon

$vis(p)$ aller von p aus sichtbaren Punkte heißt das *Sichtbarkeitspolygon* von p in P. Es besteht aus (Teilen von) Kanten von P und aus künstlichen Kanten, die durch ins Innere von P hereinragende spitze Ecken (d. h. solche mit Innenwinkel $> \pi$) entstehen, welche die Sicht einschränken; siehe Abbildung 1.10.

Polyeder

Das Gegenstück zum einfachen Polygon ist das einfache *Polyeder*. Sein Rand – die *Oberfläche* – besteht aus Polygonen. Jede Kante gehört dabei zu genau zwei Polygonen. Diese Oberfläche zerlegt den Raum in zwei Gebiete, ein beschränktes inneres und ein unbeschränktes äußeres Gebiet. So wie ein Polygon bis auf Deformation einem Kreis gleicht, läßt sich ein Polyeder zu einer Kugel „ausbeulen“. Abbildung 1.11 (ii) zeigt ein Polyeder, das sich als Vereinigung eines Quaders mit einem Tetraeder ergibt (sowie dessen konvexe Hülle, siehe weiter unten).

konvex

Eine Teilmenge K vom $\mathbb{R}^d$ heißt *konvex*, wenn sie zu je zwei Punkten p, q auch das Liniensegment pq enthält. Der $\mathbb{R}^d$ ist wie auch alle Quader und Kugeln konvex.

Eine Teilmenge K der Ebene ist genau dann konvex, wenn durch jeden Punkt p auf dem Rand von K eine Gerade führt, so

[2] Hierbei fassen wir Punkte als Ortsvektoren auf; die Multiplikation mit a erfolgt koordinatenweise.

[3] Manche Autoren sind beim Begriff *Polygon* liberaler und erlauben, daß der Rand von P Selbstschnitte aufweist oder nicht zusammenhängend ist, das Innere von P also Löcher hat. Um solche Fälle auszuschließen, nennt man P oft ein *einfaches Polygon*.

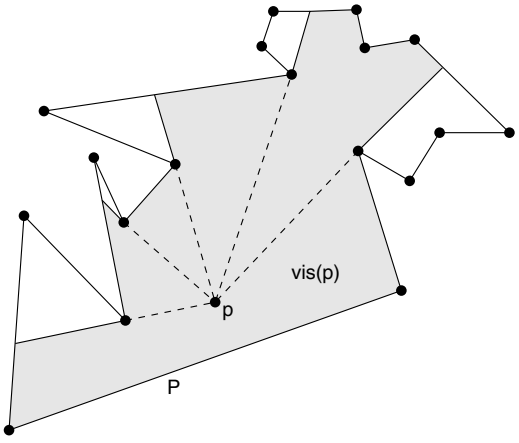


Abb. 1.10 Ein Polygon P und das Sichtbarkeitspolygon eines seiner Punkte p.

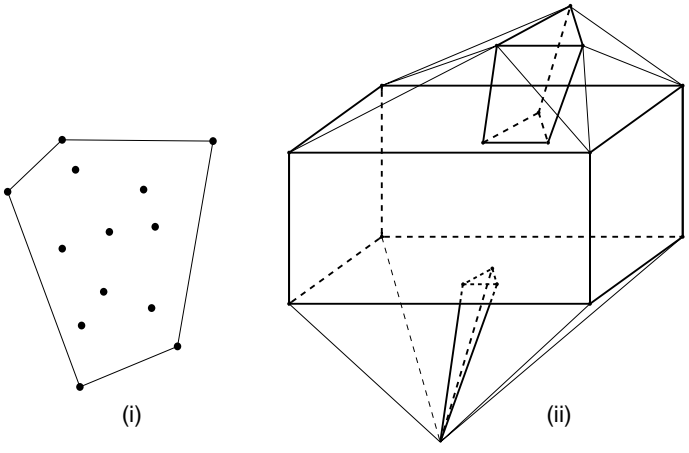


Abb. 1.11 Die konvexe Hülle von (i) einer Punktmenge in der Ebene und (ii) der Vereinigung eines Quaders mit einem Tetraeder.

daß K disjunkt ist von einer der beiden offenen Halbebenen, in die diese Gerade den $\mathbb{R}^2$ teilt (K liegt dann ganz in der Vereinigung der anderen offenen Halbebene mit der Geraden). Eine solche Gerade heißt eine *Stützgerade* von K im Punkt p. Beispiele finden sich in Abbildung 1.12.

Stützgerade

Analog sind konvexe Mengen K im $\mathbb{R}^3$ dadurch charakterisiert, daß sich in jedem Punkt auf ihrem Rand eine *Stützebene* anlegen läßt, die den $\mathbb{R}^3$ so in zwei offene Halbräume teilt, daß einer der beiden mit K einen leeren Durchschnitt hat.

Stützebene

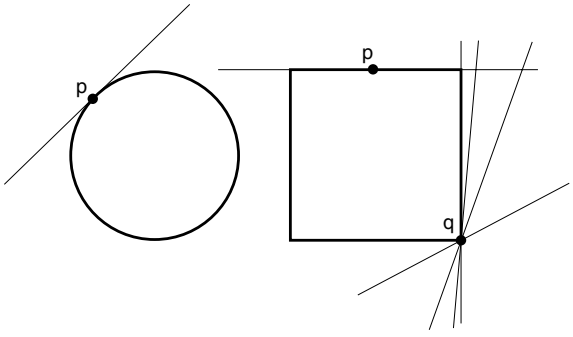


Abb. 1.12 Stützgeraden von Kreis und Rechteck.

Stützhyperebene

In Analogie zum zwei- und dreidimensionalen Raum kann man eine konvexe Menge K im $\mathbb{R}^d$ dadurch charakterisieren, daß es zu jedem ihrer Randpunkte eine *Stützhyperebene* gibt, die den $\mathbb{R}^d$ so in zwei offene Halbräume teilt, daß einer von ihnen von K disjunkt ist.

Eine weitere nützliche Eigenschaft: Sind K_1 und K_2 zwei *disjunkte* konvexe Mengen, so existiert eine Hyperebene h mit Halbräumen H_1 und H_2, so daß $K_1 \subseteq h \cup H_1$ und $K_2 \subseteq h \cup H_2$ gelten. Die Hyperebene h *trennt* die beiden konvexen Mengen.

Übungsaufgabe 1.7 Kann man zu zwei disjunkten konvexen Mengen in der Ebene stets eine Gerade finden, die die beiden Mengen trennt und mindestens eine von ihnen nicht berührt?

Übungsaufgabe 1.8 Sei P ein Polygon. Man zeige:

(i) P ist genau dann konvex, wenn von jedem Punkt $p \in P$ ganz P sichtbar ist.

(ii) Besteht P aus n Kanten, so ist für jeden Punkt $p \in P$ auch $vis(p)$ durch höchstens n Kanten berandet.

Übungsaufgabe 1.9 Seien P und Q zwei einander schneidende konvexe Polygone mit m bzw. n Kanten. Wie viele Kanten kann das Polygon $P \cup Q$ höchstens besitzen?

Für eine beliebige Teilmenge A des $\mathbb{R}^d$ ist

$$ch(A) = \bigcap_{\substack{K \supseteq A \\ K \text{ konvex}}} K$$

konvexe Hülle

die kleinste konvexe Menge, die A enthält. Sie wird die *konvexe Hülle* von A genannt. In Abbildung 1.11 sind zwei Beispiele konvexer Hüllen abgebildet. Zur praktischen Berechnung von $ch(A)$

taugt unsere Definition natürlich nicht. Wir werden aber später effiziente Algorithmen zur Konstruktion von konvexen Hüllen endlicher Punktmengen kennenlernen.

Die Bedeutung der konvexen Hülle $ch(A)$ liegt unter anderem darin, daß sie die Originalmenge A umfaßt und sich als konvexe Menge selbst schön von ihrer Umwelt „trennen“ läßt.

Wir können konvexe Hüllen auch dazu verwenden, in jeder Dimension d die einfachste echt d-dimensionale Menge zu definieren.

Für zwei Punkte p, q im $\mathbb{R}^d$, die nicht identisch sind, ist die konvexe Hülle

$$ch\{p,q\} = pq = \{ap + bq\,;\, 0 \le a, b, \text{ und } a + b = 1\}$$

das Liniensegment von p nach q. Wenn wir die Einschränkung $0 \le a, b$ an die Parameter a und b fortlassen, erhalten wir die *Gerade* durch p und q, den affinen Raum[4] kleinster Dimension, der beide Punkte enthält.

Für drei Punkte p, q, r, die nicht auf einer gemeinsamen Geraden liegen, ist die konvexe Hülle

$$\begin{aligned} ch\{p,q,r\} &= tria(p,q,r) \\ &= \{ap + bq + cr\,;\, 0 \le a, b, c, \text{ und } a + b + c = 1\} \end{aligned}$$

das *Dreieck* mit den Ecken p, q und r. Durch $\{ap + bq + cr\,;\, a, b, c \in \mathbb{R}\}$ wird die *Ebene* durch p, q, r beschrieben. Analog ist für vier Punkte, die nicht auf einer gemeinsamen Ebene liegen,

$$ch\{p,q,r,s\} = \{ap+bq+cr+es\,;\, 0 \le a, b, c, e \text{ und } a+b+c+e = 1\}$$

das *Tetraeder* mit den Ecken p, q, r, s. Es liegt in dem eindeutig bestimmten affinen Teilraum des $\mathbb{R}^d$, der die vier Punkte enthält.

Diese Konstruktion läßt sich in höhere Dimensionen fortsetzen. Die konvexe Hülle von $n + 1$ Punkten im $\mathbb{R}^d$, $n \le d$, die nicht in einem $(n-1)$-dimensionalen Teilraum liegen, nennt man das *Simplex* der Dimension n. Simplex

Hin und wieder werden elementare Sätze aus der *Trigonometrie* benötigt. Wir werden im wesentlichen mit den folgenden Tatsachen auskommen.

Nach dem *Satz des Pythagoras* gilt im rechtwinkligen Dreieck die Formel $a^2 + b^2 = c^2$ für die Längen der Kanten a, b, die den rechten Winkel einschließen, und die Länge c der gegenüberliegenden Kante. Dieser Satz ist zur Aussage $\sin^2\beta + \cos^2\beta = 1$ äquivalent; siehe Abbildung 1.13 (i). Satz des Pythagoras

Eine Verallgemeinerung des Satzes von Pythagoras auf nichtrechtwinklige Dreiecke stellt der *Kosinussatz* dar, der in Abbil- Kosinussatz

[4] Ein *affiner Teilraum* des $\mathbb{R}^d$ ist eine Menge $p + V$, wobei V ein Untervektorraum des $\mathbb{R}^d$ ist und p ein d-dimensionaler Translationsvektor. Die Dimension des affinen Teilraums entspricht der von V.

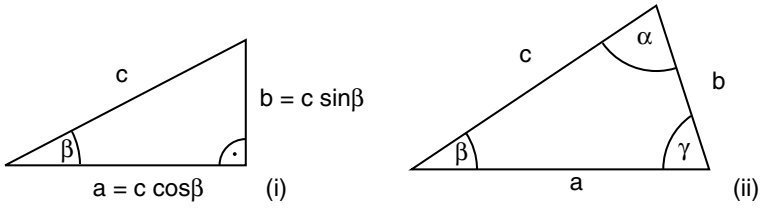


Abb. 1.13 In (i) gilt $a^2 + b^2 = c^2(\cos^2\beta + \sin^2\beta) = c^2$, der Satz des Pythagoras. In (ii) besagt der Kosinussatz $a^2 + b^2 - 2ab\cos\gamma = c^2$. Der Sinussatz lautet $a/\sin\alpha = b/\sin\beta = c/\sin\gamma$.

Sinussatz

dung 1.13 (ii) illustriert ist. Daneben gilt der *Sinussatz*.

Skalarprodukt

Sind p und q zwei Punkte im $\mathbb{R}^d$, so gilt für ihr *Skalarprodukt* $p \cdot q$

$$\sum_{i=1}^{d} p_i q_i = p \cdot q = |p|\,|q|\cos\alpha,$$

wobei α den Winkel zwischen ihren Ortsvektoren am Nullpunkt bezeichnet. Allgemein ist für die Winkelmessung (und für das

positive Richtung

Durchlaufen geschlossener Kurven in der Ebene) die positive Richtung *gegen* den Uhrzeigersinn; beim Skalarprodukt kommt es aber auf die Reihenfolge von p und q nicht an.

orthogonal

Zwei Vektoren stehen genau dann *aufeinander senkrecht*, wenn ihr Skalarprodukt den Wert Null hat.

Satz des Thales

Ist pq eine Sehne im Kreis K, so können wir an jedem Punkt r auf dem Kreisbogen von q nach p den Winkel zwischen rp und rq betrachten. Der *Satz des Thales* besagt, daß dieser Winkel für alle Punkte r aus demselben Kreisbogen gleich groß ist. Insbesondere: α ist $< \pi/2$ für Punkte r auf dem längeren Kreisbogenstück und $\beta > \pi/2$ für Punkte aus dem kürzeren Bogen; siehe Abbildung 1.14. Es gilt $\alpha + \beta = \pi$. Wenn wir im Beispiel von Abbildung 1.14 den Sehnenendpunkt q festhalten und mit p auf dem Kreisrand nach rechts wandern, wird α größer, und β schrumpft um denselben Betrag. Wenn dann die Sehne pq durch den Mittelpunkt des Kreises geht, so ist $\alpha = \beta = \pi/2$.

Elementar-operationen

Zum Schluß dieses Abschnitts über geometrische Grundlagen wollen wir noch ein paar praktische Hinweise und Beispiele geben. Bei der algorithmischen Lösung geometrischer Probleme treten gewisse *Elementaroperationen* auf, die zum Teil viele Male auszuführen sind. Zum Beispiel soll bestimmt werden, auf welcher Seite einer Geraden im $\mathbb{R}^2$ ein gegebener Punkt liegt (siehe unten bei Halbebenentest). Obwohl man solche elementaren Aufgaben bei der Diskussion der Komplexität von Problemen gern

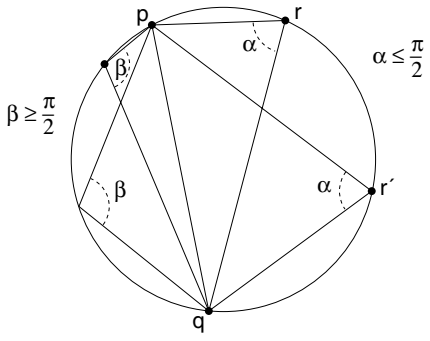


Abb. 1.14 An allen Punkten r aus demselben Kreisbogen ist der Winkel über der Sehne pq derselbe.

als trivial ansieht, müssen sie in der Praxis doch implementiert werden. Hierbei kann man mehr oder weniger geschickt vorgehen. Eine allzu sorglose Implementierung kann sich nachteilig auf die Übersichtlichkeit, die Robustheit und die praktische Effizienz des Algorithmus auswirken.

Vermeidung unnötiger Berechnungen

Eine ebenso simple wie wirkungsvolle Maßnahme besteht in der *Vermeidung unnötiger Berechnungen*. Bei der Bestimmung der nächsten Nachbarn für eine vorgegebene Punktmenge sind immer wieder Tests des Typs „Liegt p näher an q als an r?“ auszuführen. Hierbei ist es nicht nötig, jeweils den echten euklidischen Abstand $|pq| = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2}$ auszurechnen. Man kann ebensogut die Quadrate $|pq|^2$ miteinander vergleichen und die Berechnung der Wurzeln einsparen.

Bisektor

Angenommen, es soll zu zwei verschiedenen Punkten p, q in der Ebene die Mittelsenkrechte $B(p, q)$ auf dem Liniensegment pq berechnet werden. Diese Gerade wird für uns später von Interesse sein, weil sie aus genau den Punkten besteht, die zu p und q denselben Abstand haben. Man nennt $B(p, q)$ den *Bisektor* von p und q; siehe Abbildung 1.15 (i).

Die Gerade $B(p, q)$ muß durch den Mittelpunkt $m = \frac{1}{2}(p + q)$ von pq laufen; wir machen daher den Ansatz

$$B(p, q) = \{m + al; a \in \mathbb{R}\}$$

für die parametrisierte Darstellung, wobei der Vektor $l = (l_1, l_2)$ noch zu bestimmen ist. Weil l auf pq senkrecht steht, muß für das Skalarprodukt gelten

$$l_1(q_1 - p_1) + l_2(q_2 - p_2) = 0.$$

Es mag naheliegend erscheinen, nun $l_1 = 1$ und

$$l_2 = \frac{p_1 - q_1}{q_2 - p_2}$$

in die Darstellung von $B(p, q)$ einzusetzen. Für den Fall, daß pq waagerecht ist, würde diese Formel aber zu einer Division durch 0 führen! Besser setzen wir $l_1 = (q_2 - p_2)$, $l_2 = (p_1 - q_1)$ und erhalten

$$B(p, q) = \{\frac{1}{2}(p + q) + a(q_2 - p_2, p_1 - q_1); a \in \mathbb{R}\}$$

als Parameterdarstellung des Bisektors; diese Formel gilt in *jedem* möglichen Fall. Sie läßt sich leicht zur Geradengleichung umformen, indem man die Parameterdarstellungen der X- und Y-Koordinate nach a auflöst und gleichsetzt. Man erhält

$$X(p_1 - q_1) + Y(p_2 - q_2) + \frac{q_1^2 + q_2^2 - p_1^2 - p_2^2}{2} = 0.$$

In dem (durch unsere Annahme ausgeschlossenen) Fall $p = q$ ergibt die Parameterdarstellung den Punkt p, die Geradengleichung dagegen die ganze Ebene. Das zweite Ergebnis kann sinnvoller sein, wenn man wirklich an der Menge aller Punkte mit gleichem Abstand zu p und q interessiert ist.

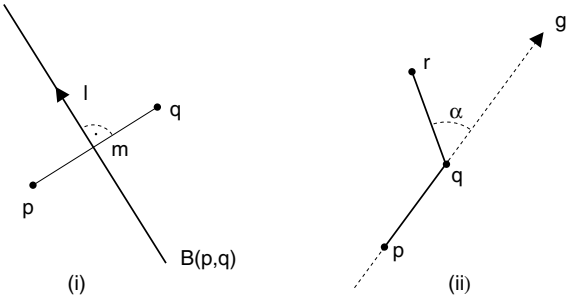


Abb. 1.15 (i) Der Bisektor von zwei Punkten. (ii) Ist der Winkel α positiv oder negativ?

Halbebenentest

Eine andere häufig auftretende Elementaroperation ist der *Halbebenentest*; siehe Abbildung 1.15 (ii). Gegeben sind drei Punkte p, q, r in der Ebene. Man will wissen, ob r links oder rechts von der Geraden g durch p, q liegt, die in Richtung von p nach q orientiert ist. Ebensogut kann man fragen, ob das Liniensegment qr im Punkt q von pq nach links oder nach rechts abbiegt.

Ein möglicher Ansatz besteht darin, zunächst die Geradengleichung

$$\begin{aligned} G(X,Y) &= (q_2-p_2)X + (p_1-q_1)Y - p_1q_2 + p_2q_1 \\ &= (q_2-p_2)(X-p_1) + (p_1-q_1)(Y-p_2) \end{aligned}$$

für die Gerade g aufzustellen. Der Punkt $r = (r_1, r_2)$ liegt genau dann links von g, wenn $G(r_1, r_2) < 0$ ist.

Eine andere Möglichkeit ist die Verwendung der Formel für die *Dreiecksfläche mit Vorzeichen*. Danach ergibt

Dreiecksfläche mit Vorzeichen

$$\frac{1}{2}\begin{vmatrix} p_1 & p_2 & 1 \\ q_1 & q_2 & 1 \\ r_1 & r_2 & 1 \end{vmatrix} = (r_1-p_1)\frac{r_2+p_2}{2} + (q_1-r_1)\frac{q_2+r_2}{2} + (p_1-q_1)\frac{p_2+q_2}{2}$$

den *positiven* Flächeninhalt des Dreiecks $tria(p, q, r)$, falls (p, q, r) die *positive* Folge der Dreiecksecken ist (d. h. die entgegen dem Uhrzeigersinn), sonst ergibt sich der *negative* Flächeninhalt. Hat daher die Determinante einen positiven Wert, so liegt r links von der orientierten Geraden g. Der Wert ist genau dann gleich 0, wenn r auf g liegt.

Übungsaufgabe 1.10 Man bestimme den Winkel α beim Halbebenentest, wie er in Abbildung 1.15 dargestellt ist.

Bei diesen Überlegungen sind wir stillschweigend davon ausgegangen, daß wir mit reellen Zahlen *exakt* rechnen können, was in der Praxis schon an der Endlichkeit der Zahldarstellung scheitern muß.

numerische Probleme

Mit *rationalen Zahlen* kann man im Prinzip beliebig genau rechnen. Auch für den Umgang mit *algebraischen Zahlen* gibt es exakte formale Methoden; man verwendet die jeweiligen Polynomgleichungen und berechnet Intervalle, die die verschiedenen reellen Nullstellen voneinander isolieren. Die Werte analytischer Funktionen, wie *sin*, *exp*, *log*, lassen sich durch Potenzreihen beliebig genau approximieren. Alle diese Ansätze sind aber überaus rechenzeitintensiv und benötigen zum Teil auch enormen Speicherplatz, etwa für die Verwaltung langer Zähler und Nenner rationaler Zahlen.

Ein Ausweg liegt in einer geschickten Verbindung von unpräziser und präziser Arithmetik. Solange es geht, arbeitet man mit einer Zahldarstellung fester Länge und führt numerische *Fehlerintervalle* mit. Wenn irgendwann zum Beispiel das Vorzeichen einer Determinante bestimmt werden soll und die Genauigkeit dazu nicht mehr ausreicht, weil der berechnete Wert zu nahe bei null

liegt, besinnt man sich auf die formale Definition der beteiligten Zahlen und rechnet präzise.

Diese Beispiele zeigen, daß auch die Implementierung geometrischer Algorithmen ein interessantes Gebiet darstellt, das für die Praxis natürlich besonders wichtig ist. Inzwischen gibt es große Algorithmensysteme wie LEDA und die in einem europäisch-israelischen Verbundprojekt entstandene Bibliothek CGAL. Beide enthalten eine Vielzahl kombinatorischer und geometrischer Algorithmen und sind sehr gut dokumentiert. Das System LEDA wird im Buch von Mehlhorn und Näher [104] beschrieben, Einzelheiten zu CGAL findet man unter der Adresse `http://www.cgal.org` und in Fabri et al. [53].

LEDA

CGAL

1.2.4 Komplexität von Algorithmen

Unterschiedliche Algorithmen zur Lösung desselben Problems lassen sich miteinander vergleichen, indem man feststellt, wie effizient sie von den Ressourcen *Rechenzeit* und *Speicherplatz* Gebrauch machen. Der Bedarf hängt meist von der Größe des Problems ab, das es zu lösen gilt, zum Beispiel von der Anzahl n der Punkte, für die ein nächster Nachbar bestimmt werden soll.

Rechenzeit

Speicherplatz

Im Prinzip gibt es zwei verschiedene Möglichkeiten, um konkurrierende Algorithmen A und B miteinander zu vergleichen. Für den *Anwender* kann es durchaus attraktiv sein, beide Verfahren in seiner Zielsprache auf seinem Rechner zu implementieren und dann für solche Problemgrößen laufen zu lassen, die für ihn von Interesse sind. Rechenzeit und Speicherplatz können dann direkt nachgemessen werden.

experimentelles Vorgehen

Der *Entwerfer* von Algorithmen, und wer immer sich für die inhärente Komplexität eines Problems interessiert, wird von konkreten Programmiersprachen und Rechnern abstrahieren und analytisch vorgehen. Hierzu wird ein abstraktes Modell gebildet, in dem sich über die Komplexität von Algorithmen und Problemen reden läßt.

analytisches Vorgehen

Dieses Modell (RAM, O, Ω, Θ) wird im folgenden vorgestellt. Im Anschluß daran kommen wir auf die beiden Ansätze für die Beurteilung von Algorithmen – den experimentellen und den analytischen – zurück.

Beim analytischen Ansatz mißt man nicht die Ausführungszeit auf einem konkreten Prozessor, sondern betrachtet die Anzahl der Schritte, die eine *idealisierte Maschine* bei Ausführung eines Algorithmus zur Lösung eines Problems der Größe n machen würde. Dieses Maschinenmodell sollte möglichst einfach sein, um die Analyse zu erleichtern.

idealisierte Maschine

In der Algorithmischen Geometrie wird als Modell oft die REAL RAM verwendet, eine *random access machine*, die mit reellen Zahlen rechnen kann. Sie verfügt über abzählbar unendlich viele Speicherzellen, die mit den natürlichen Zahlen adressiert werden. Jede Speicherzelle kann eine beliebige reelle oder ganze Zahl enthalten. Daneben stehen ein Akkumulator und endlich viele Hilfsregister zur Verfügung.

REAL RAM

Im Akkumulator können die Grundrechenarten $(+, -, \cdot)$ ausgeführt werden, zusätzlich die Division reeller Zahlen und die auch in vielen Programmiersprachen bekannten Operationen *div* und *mod* für ganze Zahlen. Außerdem kann getestet werden, ob der Inhalt des Akkumulators größer als Null ist. In Abhängigkeit vom Ergebnis kann das Programm verzweigen. Die Speicherzellen können direkt adressiert werden („Lade den Inhalt von Zelle i in den Akkumulator“) oder indirekt („Lade den Inhalt derjenigen Zelle in den Akkumulator, deren Adresse in Zelle j steht“). Im letzten Fall muß der Inhalt von Zelle j eine natürliche Zahl sein.

Manchmal erlaubt man der REAL RAM zusätzliche Rechenoperationen, wie etwa $\sqrt{}$, *sin*, *cos* oder die Funktion *trunc*. Das sollte man dann explizit erwähnen, weil es einen Einfluß auf die prinzipielle Leistungsfähigkeit des Modells haben kann.

Die Ausführung eines jeden Befehls (Rechenoperation, Speicherzugriff oder Test mit Programmsprung) zählt als ein *Elementarschritt* der RAM.

Elementarschritt

Dieses Modell ist einem realen Rechner ähnlich, soweit es die prinzipielle Funktionsweise des Prozessors und die Struktur des Hauptspeichers betrifft. Ein wesentlicher Unterschied besteht aber darin, daß in einer Zelle eines realen Hauptspeichers nur ein Wort fester Länge Platz hat und nicht eine beliebig große Zahl. Infolgedessen kann man „in einem Schritt“ nur solche Zahlen verarbeiten, deren Darstellung die Wortlänge nicht überschreitet. Anders gesagt: Das Modell der RAM ist realistisch, solange man mit beschränkten Zahlen rechnet.[5]

wie real ist die RAM?

Sei nun Π ein Problem (z. B. *all nearest neighbors*), und sei $P \in \Pi$ ein Beispiel des Problems der Größe $|P| = n$ (also z. B. eine konkrete Menge von n Punkten in der Ebene). Ist dann A ein Algorithmus zur Lösung des Problems Π, in RAM-Anweisungen formuliert, so bezeichnet $T_A(P)$ die Anzahl der Schritte, die die RAM ausführt, um die Lösung für das Beispiel P zu berechnen.

[5] Vom theoretischen Standpunkt aus kann man die RAM als Registermaschine mit abzählbar unendlich vielen Registern oder als Turingmaschine mit Halbband ansehen, dessen Felder jeweils einen Buchstaben aus einem unendlichen Alphabet enthalten können. Ein echter Leistungszuwachs im Sinne einer Erweiterung der Klasse der berechenbaren Funktionen ergibt sich hierdurch aber nicht.

Mit

$$T_A(n) = \max_{P\in\Pi,\,|P|=n} T_A(P)$$

worst case bezeichnen wir die Laufzeit von A im *worst case*. Entsprechend ist der Speicherverbrauch $S_A(n)$ die maximale Anzahl belegter Speicherzellen bei der Bearbeitung von Beispielen der Größe n.

Tatsächlich interessieren wir uns nicht so sehr für die genauen Werte von $T_A(n)$ und $S_A(n)$, sondern dafür, wie schnell diese Größen für $n \to \infty$ wachsen. Zur Präzisierung dient die *O*-Notation *O-Notation*. Es bezeichnen f, g Funktionen von den natürlichen Zahlen in die nicht-negativen reellen Zahlen. Man definiert:

$$\begin{aligned} O(f) \quad = \quad & \{\, g;\ \text{es gibt } n_0 \geq 0 \text{ und } C > 0, \\ & \qquad \text{so daß } g(n) \leq Cf(n) \text{ für alle } n \geq n_0 \text{ gilt} \,\}, \end{aligned}$$

und für ein $g \in O(f)$ sagt man: g „ist (in) groß O von f“. Inhaltlich bedeutet das, daß fast überall die Funktion g durch f nach oben beschränkt ist – bis auf einen konstanten Faktor.

Man erlaubt endlich viele Ausnahmestellen $n < n_0$, damit man obere Schranke auch solche Funktionen f als *obere Schranken* verwenden kann, die für kleine Werte von n den Wert Null annehmen. Die Aussagen

$$3n + 5 \in O(n),\ \log_2(n+1) \in O(\lfloor \log_{256} n \rfloor)$$

wären sonst nicht korrekt. Hierbei bedeutet $\log_b(n)$ den *Logarithmus zur Basis b*, und $\lfloor x \rfloor$ gibt die *größte ganze Zahl* $\leq x$ an; siehe Übungsaufgabe 1.11 (ii) und (iii).

Zum Beispiel liefert für die Funktion

$$g(n) = 13n^3 - 19n^2 + 68n + 1 - \sin n$$

bereits die grobe Abschätzung

$$\begin{aligned} g(n) \quad \leq \quad & 13n^3 + 68n + 2 \\ \leq \quad & (13 + 68 + 2)n^3 = 83n^3 \text{ für alle } n \geq 1 \end{aligned}$$

Konstante C die Aussage $g \in O(n^3)$. Interessiert man sich auch für die *Konstante* C in O, d. h. für den Faktor C, der oben in der Definition von $O(f)$ vorkommt und für den Beweis der Aussage $g \in O(f)$ benötigt wird, so wird man etwas feiner abschätzen und

$$g(n) \leq 13n^3 \text{ für alle } n \geq 4$$

erhalten. Die Aussage $g \in O(n^4)$ ist erst recht richtig, aber schwächer. Dagegen liegt g nicht in $O(n^2)$, denn wegen

$$\begin{aligned} g(n) \quad \geq \quad & 13n^3 - 19n^2 \\ = \quad & 12n^3 + (n - 19)n^2 \\ \geq \quad & 12n^3 \text{ für alle } n \geq 19 \end{aligned}$$

ist g von unten fast überall durch n^3 beschränkt, bis auf einen konstanten Faktor. Die Aussage $g(n) \leq Cn^2$ ist dann falsch für alle $n > \max(\frac{C}{12}, 18)$!

untere Schranke

Zur Bezeichnung von *unteren Schranken* dient die *Omega-Notation*. Man definiert

Ω-Notation

$$\begin{aligned} g \in \Omega(f): \quad &\Longleftrightarrow \quad f \in O(g) \\ &\Longleftrightarrow \quad \text{es gibt } n_0 \geq 0 \text{ und } c > 0, \\ &\qquad\quad \text{so daß } g(n) \geq cf(n) \text{ für alle } n \geq n_0 \text{ gilt.} \end{aligned}$$

In unserem Beispiel ist $g \in \Omega(n^3)$, erst recht also $g \in \Omega(n^2)$. Schließlich verwendet man die *Theta-Notation*

Θ-Notation

$$g \in \Theta(f): \quad \Longleftrightarrow \quad g \in O(f) \text{ und } g \in \Omega(f),$$

um auszudrücken, daß g und f etwa gleich groß sind. Diese Beziehung ist offenbar eine Äquivalenzrelation. Für die Funktion g von oben gilt $g \in \Theta(n^3)$. Wir sagen, g *wächst wie* n^3 oder g *hat die Größenordnung* n^3.

Übungsaufgabe 1.11 Wir betrachten Funktionen von den natürlichen Zahlen in die nicht-negativen reellen Zahlen.

(i) Welche Funktionen liegen in $\Theta(1)$?

(ii) Wann gilt $f(n) \in O(\lfloor f(n) \rfloor)$?

(iii) Seien a, b, c, reell mit $1 < b$, $1 \leq a, c$. Man zeige:

$$\log_b(an + c) \in \Theta(\log_2 n).$$

(iv) Seien $f_1, f_2, \ldots$ Funktionen in $O(g)$. Gilt dann

$$h(n) = \sum_{i=1}^{n} f_i(n) \in O(n \cdot g(n))?$$

Sei nun Π wieder ein Problem. Wenn ein Algorithmus A zu seiner Lösung existiert, für den $T_A(n) \in O(f)$ gilt, so sagen wir, Π *hat die Zeitkomplexität* $O(f)$. Können wir beweisen, daß für jeden Algorithmus A zur Lösung von Π die Aussage $T_A(n) \in \Omega(f)$ gelten muß, so sagen wir, Π *hat die Zeitkomplexität* $\Omega(f)$. Gilt beides, so können wir sagen, Π *hat die Zeitkomplexität* $\Theta(f)$. In diesem Fall haben wir durch die Bestimmung der genauen Zeitkomplexität von Π eine unserer Grundaufgaben gelöst. Entsprechendes gilt für die Speicherplatzkomplexität eines Problems.

Komplexität eines Problems

Alle oben eingeführten Begriffe lassen sich direkt auf Situationen übertragen, bei denen die Größe des Problems durch mehr als

mehrere Größenparameter

einen Parameter gemessen wird (zum Beispiel wird bei Graphenalgorithmen häufig zwischen der Knotenzahl v und der Kantenzahl e unterschieden).

Müssen wir nun unsere Algorithmen in RAM-Befehlen formulieren? Zum Glück nicht! Durch die Beschränkung auf die Betrachtung der *Größenordnung* von $T_A(n)$ und $S_A(n)$ gewinnen wir Unabhängigkeit von den Details der Implementierung. So ist es zum Beispiel unerheblich, wie viele RAM-Befehle wir bei der Implementierung einer **for**-Schleife für das Erhöhen der Zählervariablen und den Vergleich mit der oberen Schranke exakt benötigen; die genaue Anzahl wirkt sich nur auf den jeweiligen konstanten Faktor aus.

Unabhängigkeit vom Detail

Was ist nun „besser“: ein experimenteller Vergleich von zwei konkurrierenden Algorithmen A und B oder ein Vergleich der analytisch bestimmten Größenordnungen von T_A und T_B?

Experiment vs. Analyse

Ein Test in der endgültigen Zielumgebung ist sehr aussagekräftig, vorausgesetzt

- es werden vom Typ her solche Beispiele P des Problems Π als Eingabe verwendet, wie sie auch in der realen Anwendung auftreten, und
- es wird mit realistischen Problemgrößen gearbeitet.

Wird einer dieser Gesichtspunkte vernachlässigt, kann es später zu Überraschungen kommen. Wer etwa Sortieren durch Einfügen und Quicksort experimentell miteinander vergleicht und dabei nur Eingabefolgen verwendet, die schon teilweise vorsortiert sind, wird feststellen, daß Sortieren durch Einfügen besser abschneidet. Wenn in der Anwendung später beliebige Eingabefolgen zu sortieren sind, wäre Quicksort die bessere Wahl.

Oder angenommen, die beiden Algorithmen haben im *worst case* das Laufzeitverhalten

$$\begin{aligned} T_A(n) &= 64n \log_2 n \\ T_B(n) &= 2n^2, \end{aligned}$$

und die Anwendung ist so beschaffen, daß der schlimmste Fall bei beiden Algorithmen häufig eintritt. Werden beim Test nur Beispiele der Größe $n < 256$ verwendet, so schneidet Algorithmus B besser ab als A.

Mit steigender Problemgröße fällt B aber immer weiter zurück: Für $n = 2^{10}$ ist A dreimal so schnell, für $n = 2^{15}$ fast siebzigmal so schnell wie B; siehe Abbildung 1.16.

Die Erfahrung zeigt, daß die Größe der zu behandelnden Problembeispiele mit der Zeit immer weiter zunimmt. Andererseits

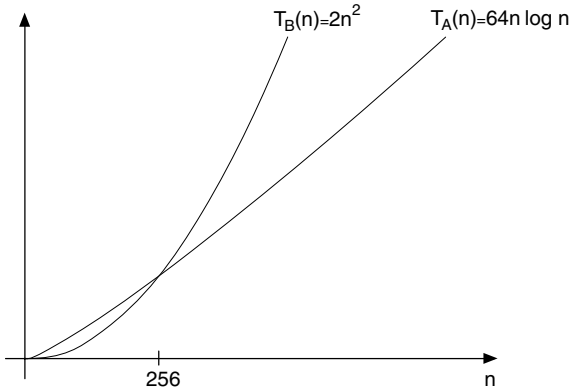


Abb. 1.16 Ab $n = 256$ ist Algorithmus A im *worst case* schneller als B.

steigt auch die Leistungsfähigkeit der Hardware. Die Frage nach dem „besseren“ Algorithmus stellt sich dann aufs neue.

Prognosen

Hier ist das analytische Vorgehen wertvoll, weil es uns verläßliche Prognosen ermöglicht. Zum Beispiel läßt sich bei Verwendung des „quadratischen“ Algorithmus B durch Verdopplung der Rechnerleistung nur ein Zuwachs der in derselben Zeit behandelbaren Problemgröße um den Faktor $\sqrt{2} \approx 1{,}414$ erkaufen. Verwendet man statt dessen den Algorithmus A, so hat dieser Faktor für $n = 2^{15}$ den Wert 1,885, für $n = 2^{18}$ ist er schon größer als 1,9.

Prognosen wie diese lassen sich übrigens auch ohne Kenntnis der Konstanten C machen, die sich in der O-Notation verbirgt; zur Abschätzung der absoluten Rechenzeit wird aber der Wert von C (oder zumindest eine gute Näherung) benötigt.

optimal vs. praktikabel

Die Frage nach der Komplexität eines Problems führt zuweilen zur Entdeckung eines Algorithmus, dessen Laufzeitverhalten zwar größenordnungsmäßig optimal ist, der aber trotzdem nicht praktikabel ist. Der Grund kann in der Schwierigkeit seiner Implementierung liegen, oder darin, daß die fragliche Konstante C so groß ist, daß der Algorithmus seine Vorzüge bei gängigen Problemgrößen nicht ausspielen kann.

In der Algorithmischen Geometrie beruht die Entdeckung eines solchen optimalen Lösungsverfahrens oft auf einer neu gewonnenen Einsicht in die Struktur des Problems. Ist diese Einheit erst vorhanden und richtig verarbeitet, so kann man im nächsten Schritt oft auch einen Algorithmus entwickeln, der praktikabel *und* optimal ist, oder zumindest nicht sehr weit vom Optimum entfernt. Vor diesem Hintergrund wird klar, daß die Ermittlung

praktische Konsequenzen

der genauen Komplexität eines Problems nicht Theorie um ihrer selbst willen ist, sondern zu konkreten Verbesserungen in der Praxis führen kann.

Bei der Definition von $T_A(n)$ und $S_A(n)$ hatten wir den *worst case* vor Augen, der sich für die schlimmstmöglichen Problembeispiele der Größe n ergeben kann. In manchen Anwendungen treten solche schlimmen Fälle erfahrungsgemäß nur selten auf. Hat man eine Vorstellung davon, mit welcher Wahrscheinlichkeit jedes Beispiel P des Problems Π als mögliche Eingabe vorkommt, so kann

mittlere Kosten

man statt $T_A(n)$ die *zu erwartende Laufzeit* betrachten, die sich bei Mittelung über alle Beispiele ergibt.

Bei geometrischen Problemen ist die Schätzung solcher Wahrscheinlichkeiten oft schwierig (wie sollten wir zum Beispiel ein „normales“ einfaches Polygon mit n Ecken von einem „weniger normalen“ unterscheiden)? Wir werden deshalb in diesem Buch keine Annahmen über die Wahrscheinlichkeit unserer Problembeispiele machen und nach wie vor mit $T_A(n)$ das Maximum aller $T_A(P)$ mit $|P| = n$ bezeichnen.

Wir können aber unserem Algorithmus gestatten, bei der Bearbeitung eines Beispiels P *Zufallsentscheidungen* zu treffen. Auch

randomisierter Algorithmus

ein derartiger *randomisierter Algorithmus* berechnet für jedes P die korrekte Lösung. Er kann aber zwischen mehreren alternativen Vorgehensweisen wählen. Je nach der Beschaffenheit von P kann die eine oder die andere schneller zum Ziel führen. Wir interessieren uns für den *Mittelwert*.

Genauer: Wir erlauben dem Algorithmus A, bei der Bearbeitung eines Problembeispiels P der Größe n insgesamt $r(n)$ mal eine Münze zu werfen (und in Abhängigkeit vom Ergebnis intern zu verzweigen). Diese Münzwürfe liefern zusammen einen von $2^{r(n)}$ möglichen Zufallsvektoren der Länge $r(n)$. Für jeden solchen Zufallsvektor z ergibt sich eine Laufzeit $T_A(P, z)$. Wir definieren dann

$$T_A(P) = \frac{1}{2^{r(n)}} \sum_z T_A(P, z)$$

als die *zu erwartende* Schrittzahl von A bei Bearbeitung von P. An den übrigen Definitionen ändert sich nichts. Entsprechend wird der Speicherplatzbedarf definiert.

deterministischer Algorithmus

Wenn wir hervorheben wollen, daß ein Algorithmus nicht mit Randomisierung arbeitet, nennen wir ihn *deterministisch*. Randomisierte Algorithmen sind oft viel einfacher als ihre deterministischen Konkurrenten. Sie werden uns an mehreren Stellen begegnen.

In diesem Buch werden ausschließlich sequentielle Algorithmen betrachtet. Wer sich für parallele Verfahren in der Algorith-

mischen Geometrie interessiert, sei z. B. auf Dehne und Sack [40] verwiesen.

1.2.5 Untere Schranken

Ein Problem hat die Zeitkomplexität $\Omega(f)$, wenn es für jeden Algorithmus eine Konstante $c > 0$ gibt, so daß sich für jedes hinreichend große n Beispiele der Größe n finden lassen, für deren Lösung der Algorithmus mindestens $cf(n)$ viele Schritte benötigt. Die Funktion f heißt dann eine *untere Schranke* für das Problem.

triviale untere Schranken

Manchmal lassen sich *triviale untere Schranken* f sehr leicht angeben. Um zum Beispiel in einer Menge von n Punkten in der Ebene einen am weitesten links gelegenen zu bestimmen, muß man zumindest jeden Punkt einmal inspizieren; also ist $f(n) = n$ eine untere Schranke für das Problem. Ebenso benötigt ein Verfahren zur Bestimmung der k Schnittpunkte einer Menge von n Geraden mindestens k Schritte, um die Schnittpunkte auszugeben. Hier ist also $f(n, k) = k$ eine untere Schranke.

Meist ist es aber schwierig, *dichte* untere Schranken f zu finden, d. h. solche, für die $\Theta(f)$ die genaue Komplexität des Problems ist. Ein prominentes Beispiel ist *Sortieren durch Vergleiche*.

Sortieren durch Vergleiche

Zu sortieren ist eine Folge von n paarweise verschiedenen Objekten $(q_1, \ldots, q_n)$ aus einer Menge Q, auf der eine vollständige Ordnung $<$ definiert ist. Wir stellen uns vor, daß die Objekte q_i in einem Array gespeichert sind, auf das wir keinen direkten Zugriff haben. Unsere Eingabe besteht aus der Objektanzahl n und einer Funktion K. Ein Aufruf $K(i, j)$ vergleicht die Objekte an den Positionen i und j im Array und liefert folgendes Ergebnis:

$$K(i,j) = \begin{cases} 1, & \text{falls } q_i < q_j \\ 0, & \text{falls } q_i > q_j. \end{cases}$$

Die Sortieraufgabe besteht in der Berechnung derjenigen Permutation $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$, für die gilt:

$$q_{\pi(1)} < q_{\pi(2)} < \ldots < q_{\pi(n)}.$$

Theorem 1.4 *Sortieren durch Vergleiche hat die Komplexität* $\Omega(n \log n)$.

Beweis. Wir beschränken uns beim Beweis auf deterministische Algorithmen. Eine Verallgemeinerung auf randomisierte Algorithmen ist aber möglich.

Sei also A ein *deterministisches* Verfahren zur Lösung des Sortierproblems durch Vergleiche. Dann wird der erste Aufruf $K(i, j)$

der Vergleichsfunktion nach dem Start von A immer für dasselbe Indexpaar (i, j) ausgeführt, unabhängig von der Inputfolge $(q_1, \ldots, q_n)$, über die A ja noch keinerlei Information besitzt.

Wenn $K(i, j) = 1$ ist, das heißt für alle Eingaben, die der Menge

$$W(i, j) = \{(q_1, \ldots, q_n) \in Q^n; q_i < q_j\}$$

angehören, wird der zweite Funktionsaufruf *dasselbe* Indexpaar (k, l) als Parameter enthalten, denn A ist deterministisch und weiß zu diesem Zeitpunkt über die Inputfolge nur, daß $q_i < q_j$ gilt. Ebenso wird für alle Folgen aus $W(j, i)$ derselbe Funkionsaufruf als zweiter ausgeführt.

Entscheidungsbaum

Wenn wir mit dieser Überlegung fortfahren,[6] erhalten wir den vergleichsbasierten *Entscheidungsbaum* von Algorithmus A, einen binären Baum, dessen Knoten mit Vergleichen „$q_i < q_j$?“ beschriftet sind. An den Kanten steht entweder 1 oder 0; siehe Abbildung 1.17.

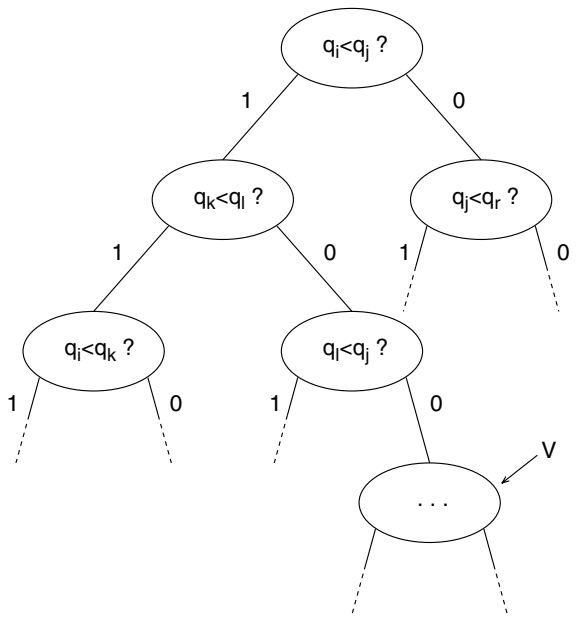


Abb. 1.17 Der Entscheidungsbaum eines vergleichsbasierten Sortierverfahrens.

Genau die Eingabetupel aus der Menge

$$W(i, j) \cap W(l, k) \cap W(j, l) = \{(q_1, \ldots, q_n) \in Q^n; q_i < q_j < q_l < q_k\}$$

[6] Wir interessieren uns nur für die Vergleiche und ignorieren alle „Nebenrechnungen“ von A.

führen zum Knoten v.

Weil A das Sortierproblem löst, gibt es zu jedem Blatt des Entscheidungsbaums eine Permutation π, so daß nur Inputfolgen mit

$$q_{\pi(1)} < q_{\pi(2)} < \ldots < q_{\pi(n)}$$

zu diesem Blatt führen. Für jede Permutation muß ein Blatt vorhanden sein. Der Entscheidungsbaum hat daher mindestens $n!$ Blätter, folglich mindestens die Höhe

$$\begin{aligned} \log_2(n!) &\geq \log_2\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) \\ &\geq \frac{n}{2}\log_2\left(\frac{n}{2}\right) \\ &\geq \frac{1}{3}n\log_2 n \text{ für alle } n \geq 8. \end{aligned}$$

Es gibt also eine Permutation, für die das Verfahren A mindestens $\frac{1}{3}n\log_2 n$ viele Vergleiche ausführt. □

Weil man mit Heapsort oder Mergesort in Zeit $O(n\log n)$ sortieren kann, hat das Sortierproblem die Komplexität $\Theta(n\log n)$.

Übungsaufgabe 1.12 Sei T ein Binärbaum mit m Blättern, und seien $l_1, \ldots, l_m$ die Länge der Pfade von der Wurzel zu den Blättern.

(i) Man beweise die Aussage

$$\sum_{i=1}^{m} 2^{-l_i} = 1.$$

(ii) Man benutze die Ungleichung vom geometrischen und arithmetischen Mittel, um aus (i)

$$\frac{1}{m}\sum_{i=1}^{m} l_i \geq \log_2 m$$

zu folgern.

(iii) Man beweise mit Hilfe von (ii), daß auch der mittlere Zeitaufwand für das Sortieren durch Vergleiche $\Omega(n\log n)$ beträgt, wenn alle Permutationen des Inputs gleich wahrscheinlich sind.

Theorem 1.4 ist für uns ein wenig unbefriedigend, denn wir haben es meist nicht mit abstrakten Objekten q zu tun, sondern mit *reellen Zahlen*, und die kann man nicht nur miteinander vergleichen – man kann mit ihnen auch rechnen.

Zur Verschärfung von Theorem 1.4 betrachten wir das *lineare Modell*, in dem wir das Vorzeichen von *linearen Ausdrücken* der

lineares Modell

reellen Eingabezahlen $x_1, \ldots, x_n$ testen können. Erlaubt sind jetzt also Tests der Art

$$h(x_1, \ldots, x_n) < 0?$$

für affin-lineare Abbildungen

$$h(X_1, \ldots, X_n) = c_1 X_1 + \ldots + c_n X_n + d.$$

Hierbei sind $c_1, \ldots, c_n$, d reelle Koeffizienten, die nichts mit den Eingabewerten x_i zu tun haben. Durch $h(X_1, \ldots, X_n) = 0$ wird eine Hyperebene im $\mathbb{R}^n$ definiert, falls nicht alle c_i gleich Null sind. Der Test „$h(x_1, \ldots, x_n) < 0?$“ entscheidet, in welchem der beiden offenen Teilräume, die von der Hyperebene getrennt werden, der Punkt $(x_1, \ldots, x_n)$ liegt. Vergleiche $x_i < x_j$ sind damit natürlich immer noch möglich. Als Hardwarebasis können wir uns eine REAL RAM vorstellen, die keinen direkten Zugriff auf die Zahlenfolge $(x_1, \ldots, x_n)$ hat. Sie kann lediglich auf beliebige Weise reelle Koeffizienten $(c_1, \ldots, c_n, d)$ bilden und in speziellen Speicherzellen ablegen. Nach Aufruf eines speziellen *Orakelbefehls* enthält der Akkumulator den Wert 1 oder 0, je nach Ausgang des Vorzeichentests

Orakel

$$c_1 x_1 + \ldots + c_n x_n + d < 0?$$

Hierfür wird nur ein Rechenschritt veranschlagt.

Folgender Satz erlaubt es, für verschiedene Probleme untere Schranken im linearen Modell anzugeben. Wir betrachten dazu zunächst einen neuen Problemtyp, den *Elementtest*. Gegeben[7] ist eine Menge W im $\mathbb{R}^n$. Es soll ein Verfahren angegeben werden, das für ein beliebiges $x \in \mathbb{R}^n$ entscheidet, ob x in W liegt.

Elementtest

Theorem 1.5 *Angenommen, die Menge W hat m Zusammenhangskomponenten. Dann benötigt jeder Algorithmus für den Elementtest für W im linearen Modell im schlimmsten Fall mindestens* $\log_2 m$ *viele Schritte.*

Beweis. Sei A ein Algorithmus für den Elementtest für W. Wie im Beweis von Theorem 1.4 betrachten wir den linearen *Entscheidungsbaum* E von A, dessen Knoten die Tests

$$h(x_1, \ldots, x_n) < 0$$

in allen möglichen Rechenabläufen von A darstellen.

[7] Bei den Anwendungen des folgenden Satzes wird die Menge W stets durch eine *endliche Beschreibung* gegeben sein.

Es genügt zu zeigen, daß der Entscheidungsbaum E mindestens so viele Blätter hat wie die Menge W Zusammenhangskomponenten. Für jedes Blatt b des Entscheidungsbaums E sei

$$A_b = \{x \in \mathbb{R}^n;\ A \text{ terminiert bei Eingabe von } x \text{ im Blatt } b\}$$

Diese Menge A_b ist – nach Definition des linearen Modells – Durchschnitt von offenen und abgeschlossenen affinen Teilräumen

$$\{(x_1, \ldots, x_n) \in \mathbb{R}^n; h(x_1, \ldots, x_n) < 0\}$$
$$\{(x_1, \ldots, x_n) \in \mathbb{R}^n; h(x_1, \ldots, x_n) \geq 0\}.$$

Weil diese Teilräume *konvex* sind, ist auch A_b konvex, insbesondere also *zusammenhängend*. Für alle Punkte x in A_b trifft Algorithmus A dieselbe Entscheidung; folglich gilt entweder $A_b \subset W$ oder $A_b \cap W = \emptyset$. Außerdem ist

$$\mathbb{R}^n = \bigcup_{b \text{ Blatt}} A_b,$$

da der Algorithmus ja für *jede* Eingabe x eine Entscheidung treffen muß. Daraus folgt:

$$\begin{aligned} W &= \mathbb{R}^n \cap W \\ &= \bigcup_{b \text{ Blatt}} A_b \cap W \\ &= \bigcup_{b \in B} A_b \end{aligned}$$

für eine bestimmte Teilmenge B aller Blätter des Entscheidungsbaums E. Die Anzahl der Zusammenhangskomponenten dieser Menge kann höchstens so groß sein wie die Anzahl $|B|$ der zusammenhängenden Teilmengen A_b, denn jedes A_b ist vollständig in einer Zusammenhangskomponente enthalten. Andererseits ist $|B|$ kleiner gleich der Anzahl *aller* Blätter von E. □

Für $n = 1$ bedeutet Theorem 1.5, daß man $\log n$ viele Schritte benötigt um festzustellen, ob eine reelle Zahl in einem von n disjunkten Intervallen liegt; durch Anwendung von binärem Suchen läßt sich das Problem auch in dieser Zeit lösen.

Als erste Anwendung von Theorem 1.5 betrachten wir das Problem *ε-closeness*. Gegeben sind n reelle Zahlen $x_1, \ldots, x_n$ und ein $\varepsilon > 0$. Gefragt ist, ob es Indizes $i \neq j$ gibt, so daß $|x_i - x_j| < \varepsilon$ gilt. *ε-closeness*

Hat man die n Zahlen erst ihrer Größe nach sortiert, so läßt sich die Frage leicht in linearer Zeit beantworten: Es genügt, jeweils die Nachbarn in der aufsteigenden Folge

$$x_{\pi(1)} \leq x_{\pi(2)} \leq \ldots \leq x_{\pi(n)}$$

zu testen. Folglich hat *ε-closeness* die Zeitkomplexität $O(n \log n)$ im linearen Modell.[8]

Wir können nun zeigen, daß es keine schnellere Lösung gibt.

Korollar 1.6 *Das Problem ε-closeness hat im linearen Modell die Zeitkomplexität* $\Theta(n \log n)$.

Beweis. Offenbar ist *ε-closeness* ein Elementtestproblem für die Menge

$$W = \{(x_1, \ldots, x_n) \in \mathbb{R}^n;\ \text{für alle } i \neq j \text{ ist } |x_i - x_j| \geq \varepsilon\}.$$

Wir zeigen, daß die Zusammenhangskomponenten von W genau die $n!$ vielen Mengen

$$W_\pi = \{(x_1, \ldots, x_n) \in W; x_{\pi(1)} < x_{\pi(2)} < \ldots < x_{\pi(n)}\}$$

sind, wobei π alle Permutationen der n Indizes durchläuft.

Seien $p, q \in W_\pi$, und gelte z. B. $p_i \geq p_j + \varepsilon$. Dann muß auch $q_i \geq q_j + \varepsilon$ sein, und für jeden Parameter $a \in [0, 1]$ folgt ebenfalls

$$\begin{aligned} (1-a)p_i + aq_i &\geq (1-a)(p_j + \varepsilon) + a(q_j + \varepsilon) \\ &= (1-a)p_j + aq_j + \varepsilon; \end{aligned}$$

deshalb liegt das ganze Liniensegment pq in W_π. Die Menge W_π ist also konvex und insbesondere zusammenhängend.

Für zwei verschiedene Permutationen $\pi \neq \sigma$ seien $p \in W_\pi$ und $q \in W_\sigma$. Dann muß es ein Indexpaar (i, j) geben mit

$$p_i < p_j \text{ und } q_i > q_j$$

Ist nun $f(a) = (f_1(a), \ldots, f_n(a))$ eine Parametrisierung eines Weges von p nach q, so muß es wegen

$$\begin{aligned} f_i(0) - f_j(0) &= p_i - p_j < 0 \\ f_i(1) - f_j(1) &= q_i - q_j > 0 \end{aligned}$$

nach dem Zwischenwertsatz ein $a \in (0, 1)$ geben mit $f_i(a) = f_j(a)$. Der Punkt

$$f(a) = (f_1(a), \ldots, f_i(a), \ldots, f_j(a), \ldots, f_n(a))$$

gehört dann aber nicht zur Menge W.

Also sind die Mengen W_π die paarweise disjunkten Zusammenhangskomponenten von W, und die Behauptung folgt aus Theorem 1.5. □

element uniqueness

Ganz analog läßt sich ein ähnliches Problem behandeln, das den Namen *element uniqueness* trägt. Gegeben sind wieder n reelle Zahlen $x_1, \ldots, x_n$, gefragt ist, ob es Indizes $i \neq j$ mit $x_i = x_j$ gibt.

[8] Sortieren, z. B. mit Heapsort, benötigt nur Vergleiche. Tests der Art $x_i - x_j - \varepsilon < 0$ sind im linearen Modell erlaubt.

Korollar 1.7 *Das Problem element uniqueness hat die Zeitkomplexität* $\Theta(n \log n)$ *im linearen Modell.*

Nun können wir auch die gewünschte Verschärfung von Theorem 1.4 beweisen:

Korollar 1.8 *Auch im linearen Modell hat Sortieren die Zeitkomplexität* $\Theta(n \log n)$.

Beweis. Könnten wir schneller sortieren, so könnten wir damit auch das Problem *ε-closeness* schneller lösen, was wegen Korollar 1.6 unmöglich ist. □

Prinzip der Reduktion

Hier haben wir ein Beispiel, wie man ein Problem auf ein anderes reduziert, um neue untere Schranken zu beweisen.

Im linearen Modell haben wir erlaubt, Linearkombinationen der reellen Eingabezahlen $x_1, \ldots, x_n$ auf ihr Vorzeichen zu testen, und gesehen, daß sich dadurch keine schnelleren Sortierverfahren für reelle Zahlen ergeben. Was passiert, wenn wir auch Multiplikation und Division der Eingabezahlen erlauben, damit also auch Tests der Art

$$h(x_1, \ldots, x_n) > 0?,$$

in denen $h(x_1, \ldots, x_n)$ ein *Polynom* vom Grad $d \geq 1$ ist? Wenn wir versuchen, den Beweis von Theorem 1.5 auf solche Tests zu verallgemeinern, stoßen wir auf ein Problem: Die Mengen A_b aller Punkte im $\mathbb{R}^n$, für die ein Algorithmus A in demselben Blatt b seines Entscheidungsbaums terminiert, ist nicht mehr Schnitt von linearen Halbräumen und auch nicht mehr konvex! Wir haben es vielmehr nun mit *algebraischen (Pseudo-) Mannigfaltigkeiten* zu tun, einem klassischen Gegenstand der Algebraischen Geometrie.

Mannigfaltigkeit

algebraisches Modell

Eine andere Beobachtung zeigt, daß wir in diesem *algebraischen Modell* die Kosten eines Tests $h(x_1, \ldots, x_n) > 0?$ nicht länger mit nur *einem* Schritt veranschlagen dürfen. Sonst könnte nämlich das Problem *element uniqueness* plötzlich in Zeit $O(1)$ gelöst werden, durch den Test

$$\prod_{0 \leq i < j \leq n} (x_i - x_j)^2 > 0?$$

Vielmehr muß auch jede Ausführung einer Grundoperation $+, -, \cdot, /$ bei der Berechnung eines Polynomwertes $h(x_1 \ldots x_n)$ mit einer Kosteneinheit berechnet werden.

Unter Verwendung tiefliegender Resultate aus der Algebraischen Geometrie läßt sich eine Verallgemeinerung von Theorem 1.5 für das algebraische Modell beweisen. Damit bleiben alle unsere Folgerungen auch in diesem Modell gültig.

Der Vollständigkeit halber sei bemerkt, daß die Hinzunahme der *trunc*-Funktion zum Befehlsvorrat der REAL RAM gestattet, n reelle Zahlen, die jeweils gleichverteilt im Intervall $[0,1]$ gewählt wurden, *im Mittel* in Zeit $O(n)$ zu sortieren; im algebraischen Modell ist das nicht möglich. Man kann sich eine Vorstellung von der Macht der *trunc*-Funktion machen, wenn man das Elementtestproblem für die reellen Intervalle $[i, i+\varepsilon], 1 \leq i \leq n$, betrachtet. Wenn man zur nächsten ganzen Zahl i abrunden darf, läßt es sich in konstanter Zeit lösen — Theorem 1.5 verliert also seine Gültigkeit!

all nearest neighbors

Zum Schluß dieses ersten Kapitels kommen wir noch einmal auf das Problem *all nearest neighbors* zurück, mit dem wir anfangs gestartet waren, und beweisen eine untere Schranke für die Zeitkomplexität.

nächster Nachbar

Korollar 1.9 *Für jeden Punkt einer n-elementigen Punktmenge $S \subset \mathbb{R}^d$ einen nächsten Nachbarn in S zu bestimmen, hat die Zeitkomplexität $\Omega(n \log n)$.*

Beweis. Könnten wir die nächsten Nachbarn schneller berechnen, hätten wir auch eine schnellere Lösung für das Problem ε-*closeness* (im Widerspruch zu Korollar 1.6). Sind nämlich n reelle Zahlen $x_1, \ldots, x_n$ und $\varepsilon > 0$ gegeben, so können wir die Punkte

$$p_i = (x_i, 0, \ldots, 0) \in \mathbb{R}^d$$

bilden und für jeden von ihnen einen nächsten Nachbarn bestimmen. Unter diesen n Punktepaaren befindet sich auch ein

dichtestes Paar

dichtestes Paar (p_i, p_j) mit

$$|p_i p_j| = \min_{1 \leq k \neq l \leq n} |p_k p_l|,$$

und wir können dieses dichteste Paar in Zeit $O(n)$ finden. Nun brauchen wir nur noch zu testen, ob $|x_i - x_j| < \varepsilon$ gilt. □

Dabei haben wir folgendes Resultat gleich mitbewiesen:

closest pair

Korollar 1.10 *Ein dichtestes Paar von n Punkten im $\mathbb{R}^d$ zu finden, hat die Zeitkomplexität $\Omega(n \log n)$.*

Später werden wir sehen, wie man diese Probleme in Zeit $O(n \log n)$ lösen kann.

Lösungen der Übungsaufgaben

Übungsaufgabe 1.1 Ja! Angenommen, die Feder hat k Windungen, und nach der Belastung hat der Zylinder die Höhe h und den Radius r. Dann ist die Federkurve parametrisiert durch

$$f(t) = (r\cos 2k\pi t, r\sin 2k\pi t, ht), \quad 0 \le t \le 1.$$

Weil die Koordinatenfunktionen stetig differenzierbar sind, gilt nach der Integralformel für die Länge der Feder

$$\begin{aligned} \textit{Länge} &= \int_0^1 \sqrt{(r\cos 2k\pi t)'^2 + (r\sin 2k\pi t)'^2 + (ht)'^2}\, dt \\ &= \int_0^1 \sqrt{4k^2r^2\pi^2(\sin^2 2k\pi t + \cos^2 2k\pi t) + h^2}\, dt \\ &= \int_0^1 \sqrt{4r^2k^2\pi^2 + h^2}\, dt \\ &= \sqrt{4r^2k^2\pi^2 + h^2}. \end{aligned}$$

Nun hat sich die Länge der Feder durch die Belastung nicht verändert, da der elastische Bereich nicht überschritten wurde. Also muß

$$\sqrt{4k^2\pi^2 + 1} = \sqrt{4r^2k^2\pi^2 + h^2}$$

gelten, folglich

$$r^2 = 1 - \frac{h^2 - 1}{4k^2\pi^2} < 1.$$

Übungsaufgabe 1.2

(i) Reflexivität: Für jedes $a \in A$ gilt $a \sim a$, weil sich a über den konstanten Weg mit Parametrisierung $f(t) = a$ mit a verbinden läßt.

Symmetrie: Aus $a \sim b$ folgt $b \sim a$, weil sich die Durchlaufrichtung eines durch f parametrisierten Weges von a nach b durch die Parametrisierung $g(t) = f(1-t)$ umkehren läßt.

Transitivität: Gelte $a \sim b$ und $b \sim c$ mit Wegen w und v, die durch

$$\begin{aligned} f &: [0,1] \longrightarrow M, \quad f(0) = a, \quad f(1) = b \\ g &: [0,1] \longrightarrow M, \quad g(0) = b, \quad g(1) = c \end{aligned}$$

parametrisiert sind. Dann ist die Verkettung von w und v ein Weg von a nach c, der ganz in A verläuft. Er wird parametrisiert durch die stetige Abbildung

$$h(t) = \begin{cases} f(2t) & \text{für} \quad 0 \le t \le \frac{1}{2} \\ g(2t-1) & \text{für} \quad \frac{1}{2} \le t \le 1. \end{cases}$$

(ii) und (iii) Offenbar ist $Z(a)$ gerade die Äquivalenzklasse von a, d. h. die Menge der zu a bezüglich $\sim$ äquivalenten Elemente von A. Für Äquivalenzklassen sind die Behauptungen (ii) und (iii) immer richtig.

Übungsaufgabe 1.3

(i) Nein! Für eine kreuzungsfreie geometrische Einbettung in der Ebene oder auf der Kugeloberfläche würde nach der Eulerschen Formel gelten

$$v - e + f = c + 1.$$

Nun ist $v = 5$ und $e = 10$, weil von jedem der 5 Knoten genau 4 Kanten ausgehen. Außerdem ist K_5 zusammenhängend, also $c = 1$. Daraus folgt $f = 7$. Weil K_5 außerdem schlicht ist, folgt $3f \leq 2e$ aus Korollar 1.3, und wir erhalten den Widerspruch

$$20 = 2e \geq 3f = 21.$$

(ii) Ja, wie Abbildung 1.18 zeigt.

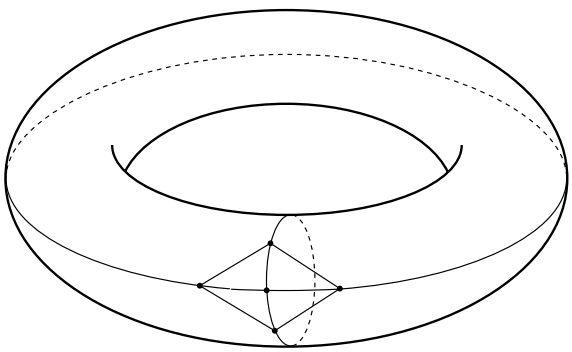


Abb. 1.18 Eine kreuzungsfreie Realisierung von K_5 auf dem Torus.

Übungsaufgabe 1.4 Nein! Für den Graphen $K_{3,3}$ gilt offenbar $v = 6, e = 9$ und $c = 1$. Wäre er planar, würde mit der Eulerschen Formel $f = 5$ folgen. Das ist aber noch kein Widerspruch, denn es gibt ja einen kreuzungsfreien Graphen mit diesen Werten: ein Quadrat mit zwei an gegenüberliegenden Kanten aufgesetzten Dreiecken, deren äußere Knoten miteinander verbunden sind, siehe Abbildung 1.19. Um zu einem Widerspruch zu gelangen, nutzen wir aus, daß der Graph $K_{3,3}$ schlicht ist und keine Kreise ungerader Länge enthalten kann; denn wenn man von einem Knoten a der Menge A startet, muß man eine gerade Anzahl von Kanten besuchen, bevor man wieder in a ankommt.

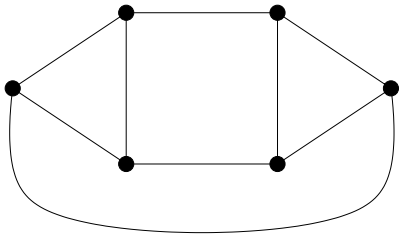


Abb. 1.19 Ein Graph mit $v = 6, e = 9, c = 1$ und $f = 5$.

Also folgt

$$20 = 4f \leq \sum_{i=1}^{f} m_i = 2e = 18\,,$$

ein Widerspruch!

Übungsaufgabe 1.5

(i) Sei e eine Kante von G, die den Punkt p mit seinem nächsten Nachbarn q verbindet; dann kann der Kreis durch q mit Mittelpunkt p keinen anderen Punkt von S im Innern enthalten, siehe Abbildung 1.20. Erst recht kann der Kreis $K(e)$ durch p und q mit Mittelpunkt im mittleren Punkt von e außer p und q keinen anderen Punkt im Innern oder auf dem Rand enthalten. Angenommen, e würde von einer anderen Kante e' von G gekreuzt, die zwei Punkte r, s miteinander verbindet. Dann lägen r und s außerhalb von $K(e)$, und ebenso lägen p und q außerhalb von $K(e')$. Das könnte nur gelten, wenn die beiden Kreise $K(e)$ und $K(e')$ sich (mindestens) viermal kreuzen. Zwei nicht zusammenfallende Kreise haben aber höchstens zwei Schnittpunkte!

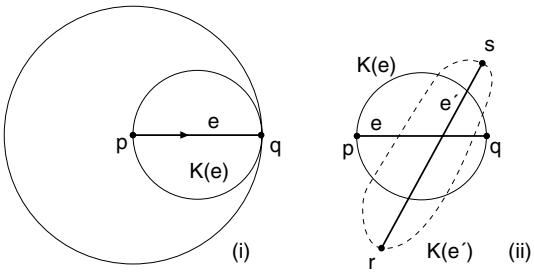


Abb. 1.20 (i) Wenn q nächster Nachbar von p ist, enthält der Kreis $K(e)$ keinen Punkt aus S im Innern. (ii) Die Kreise $K(e)$ und $K(e')$ können sich nicht viermal kreuzen!

(ii) Ein solcher geschlossener Weg würde eine Folge $p_0, p_1, \ldots p_{n-1}, p_n = p_0$ von Punkten aus S durchlaufen, bei der für jedes i, $0 \leq i \leq n-1$, gilt:

p_{i+1} ist der nächste Nachbar von p_i oder
p_i ist der nächste Nachbar von p_{i+1}.

Angenommen, es gilt für jedes i die erste Eigenschaft. Dann hat unser Weg die in Abbildung 1.21 (i) gezeigte Struktur; dabei weisen die Pfeile jeweils zum nächsten Nachbarn.

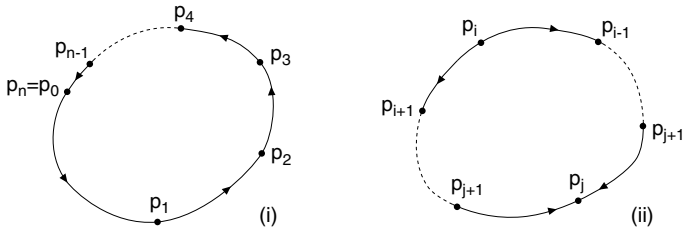


Abb. 1.21 Wenn in (i) stets $|p_i p_{i+1}| > |p_{i+1} p_{i+2}|$ gilt, kann nicht gleichzeitig $p_n = p_0$ sein. Wenn in (ii) ein Punkt existiert, zu dem beide Kanten hinführen, so gibt es auch einen, von dem aus beide Kanten wegführen.

Weil p_2 der nach Voraussetzung eindeutig bestimmte nächste Nachbar von p_1 ist, muß $|p_0 p_1| > |p_1 p_2|$ gelten, ebenso $|p_1 p_2| > |p_2 p_3|$ und so fort. Dann kann aber nicht außerdem $|p_{n-1} p_0| > |p_0 p_1|$ sein. Aus demselben Grund kann nicht für jedes i die zweite Bedingung gelten. Wenn aber nicht alle Kanten des geschlossenen Weges gleich orientiert sind, muß es einen Punkt p_i geben, von dem aus beide Kanten wegführen, wie in Abbildung 1.21 (ii) gezeigt. Dieser Punkt p_i hätte dann *zwei* nächste Nachbarn – ein Widerspruch zur Voraussetzung. Wenn also G keinen geschlossenen Weg enthält, so erst recht keine seiner Zusammenhangskomponenten. Jede von ihnen ist folglich ein Baum.

(iii) Nein. Wenn die nächsten Nachbarn eindeutig sind, besteht der Graph G wegen Teilaufgabe (ii) aus lauter Bäumen. Sei T einer dieser Bäume. Aus jedem seiner v Knoten führt genau eine Kante heraus; bei dieser Betrachtung werden genau die d Kanten doppelt erfaßt, die in beide Richtungen orientiert sind. Also ist $e = v - d$, und durch Einsetzen dieser Gleichung und $f = 1$, $c = 1$ in die Eulersche Formel $v - e + f = c + 1$ ergibt sich $d = 1$. Jeder Teilbaum T von G enthält also genau eine doppelt orientierte Kante.

Übungsaufgabe 1.6

(i) Wie im Beweis von Korollar 1.2 bezeichne m_i die Anzahl der Kanten auf dem Rand der i-ten Fläche. Dann muß stets $m_i \geq 3$ gelten, weil G schlicht und nicht leer ist. Also ist

$$3f \leq \sum_{i=1}^{f} m_i \leq 2e.$$

(ii) Wir nehmen an, daß der Graph G zusammenhängend ist; andernfalls werden seine Zusammenhangskomponenten einzeln betrachtet. Der duale Graph G^* ist kreuzungsfrei. Weil in G alle Flächen von mindestens 3 Kanten berandet sind, hat jeder Knoten von G^* einen Grad ≥ 3. Also folgt $e^* < 3f^*$ nach Korollar 1.2. Wegen $e^* = e$ und $f^* = v$ gilt für G daher $e < 3v$. Mit (i) oder direkt aus $v^* < 2f^*$ folgt $f < 2v$.

(iii) Nein! Der Graph in Abbildung 1.22 hat nur einen Knoten, aber e Kanten und $e + 1$ Flächen.

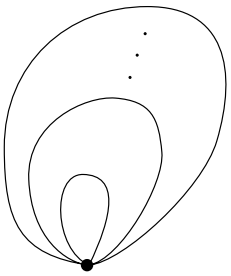


Abb. 1.22 Ein Graph, der nicht schlicht ist, kann viele Flächen und Kanten und wenige Knoten besitzen.

Übungsaufgabe 1.7 Nein! Gegenbeispiel:

$$K_1 = \{(x, y); x < 0 \text{ oder } (x = 0 \text{ und } y > 0)\}$$

$$K_2 = \{(x, y); x > 0 \text{ oder } (x = 0 \text{ und } y < 0)\}.$$

Übungsaufgabe 1.8

(i) Ist das Polygon P konvex, ist mit p und x stets auch das Liniensegment px in P enthalten. Wenn umgekehrt jeder Punkt in P jeden anderen sehen kann, können keine spitzen Ecken existieren. Also ist P konvex.

(ii) Jede Kante von P kann *höchstens eine* Kante zu $vis(p)$ beitragen. Sind nämlich die Punkte q und q' einer Kante e von P vom

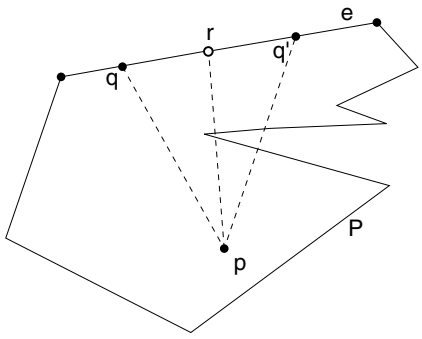


Abb. 1.23 Würde ∂P die Sicht von p auf r blockieren, wäre einer der Punkte q, q' von p aus ebenfalls nicht sichtbar.

Punkt p aus sichtbar, muß das ganze Segment qq' von p aus sichtbar sein, denn ∂P kann weder pq noch pq' kreuzen, um die Sicht auf einen inneren Punkt von qq' zu stören; siehe Abbildung 1.23.

Jede künstliche Kante von $vis(p)$ rührt von einer spitzen Ecke von P her; eine der beiden Kanten von P, die sich in dieser Ecke treffen, ist von p aus vollständig unsichtbar, kann also nicht zu $vis(p)$ beitragen. Insgesamt kann daher $vis(p)$ nicht mehr Kanten besitzen als P.

Übungsaufgabe 1.9 Höchstens $m + n + 2\min(m, n)$ viele. Sei $m \leq n$. Weil Q konvex ist, kann jede der m Kanten von P höchstens zweimal ∂Q schneiden. Zu diesen maximal $2m$ vielen Ecken kommen höchstens noch die $m+n$ Originalecken von P und Q hinzu. Daß diese Schranke scharf ist, zeigt das Beispiel von zwei konzentrischen regelmäßigen n-Ecken, von denen das eine gegenüber dem anderen um den Winkel π/n verdreht ist.

Übungsaufgabe 1.10 Zunächst wenden wir den Halbebenentest an und bestimmen, ob r auch auf der Geraden g liegt. Falls ja, stellen wir fest, ob r hinter oder vor dem Punkt q liegt; im ersten Fall ist $\alpha = 0$, im zweiten können wir nur $\alpha = \pm\pi$ berichten. Liegt aber r nicht auf der Geraden g, läßt sich der Betrag von α durch das Skalarprodukt bestimmen:

$$|\alpha| = \arccos \frac{(q-p)\cdot(r-q)}{|pq|\cdot|qr|}.$$

Übungsaufgabe 1.11

(i) In $O(1)$ liegen genau diejenigen Funktionen von den natürlichen in die nicht-negativen reellen Zahlen, die nach oben, also

insgesamt beschränkt sind. Die Klasse $\Omega(1)$ besteht aus denjenigen Funktionen mit $f(n) \geq c$ für alle $n \geq n_0$, bei geeigneter Wahl von $c > 0$ und n_0. Folglich besteht $\Theta(1)$ aus allen Funktionen f, für die Konstanten $0 < c \leq b$ existieren, so daß fast alle Werte von f in $[c, b]$ liegen.

(ii) Falls $f(n) \geq 1$ ist, gilt die Abschätzung

$$f(n) < \lfloor f(n) \rfloor + 1 \leq 2\lfloor f(n) \rfloor.$$

Ist dagegen $f(n) \in (0, 1)$, so ist $\lfloor f(n) \rfloor = 0$. Folglich ist genau dann f in $O(\lfloor f \rfloor)$, falls f nur endlich oft Werte in $(0, 1)$ annimmt.

(iii) Weil für alle $n \geq \max(c, a + 1)$

$$\log_2(an + c) \leq \log_2((a + 1)n) = \log_2(a + 1) + \log_2 n \leq 2 \log_2 n$$

ist, gilt für diese n die Abschätzung

$$\begin{aligned}(\log_b 2) \log_2 n = \log_b n &\leq \log_b(an + c) = (\log_b 2) \log_2(an + c) \\ &\leq 2(\log_b 2) \log_2 n,\end{aligned}$$

aus der die Behauptung folgt. Dabei haben wir die nützliche Gleichung

$$(\log_b c) \log_c x = \log_b x$$

verwendet, die für $1 < b, c$ erfüllt ist.

(iv) Nein! Zum Beispiel liegt jede Funktion f_i mit $f_i(n) = i$ für alle n in $O(1)$, aber es ist

$$h(n) = \sum_{i=1}^{n} i = \frac{n(n + 1)}{2} \in \Theta(n^2)$$

und nicht $h(n) \in O(n)$.

Übungsaufgabe 1.12

(i) Beweis durch Induktion über m. Für $m = 1$ stimmt die Behauptung. Ein Binärbaum T mit $m \geq 2$ Blättern hat zwei Teilbäume mit m_1 und m_2 Blättern, wobei $m = m_1 + m_2$ ist. Für die Pfadlängen h_i, k_j der Teilbäume gilt nach Induktionsvoraussetzung

$$\sum_{i=1}^{m_1} 2^{-h_i} = 1 \text{ und } \sum_{j=1}^{m_2} 2^{-k_j} = 1.$$

Weil in T alle Pfade um 1 länger sind, folgt

$$\sum_{v=1}^{m} 2^{-l_v} = 2^{-1}(\sum_{i=1}^{m_1} 2^{-h_i} + \sum_{j=1}^{m_2} 2^{-k_j}) = 1.$$

(ii) Die Ungleichung vom geometrischen und arithmetischen Mittel besagt für die Zahlen 2^{-l_i}

$$\frac{1}{m}\sum_{i=1}^{m} 2^{-l_i} \geq \sqrt[m]{\prod_{i=1}^{m} 2^{-l_i}}.$$

Die linke Seite ist nach (i) gleich

$$\frac{1}{m}\sum_{i=1}^{m} 2^{-l_i} = \frac{1}{m},$$

für die rechte ergibt sich

$$\prod_{i=1}^{m} 2^{-\frac{l_i}{m}} = 2^{-\frac{1}{m}\sum_{i=1}^{m} l_i},$$

und es folgt

$$m \leq 2^{\frac{1}{m}\sum_{i=1}^{m} l_i}.$$

(iii) Setzt man in (ii) $m = n!$ ein, so folgt, daß nicht nur die maximale, sondern schon die *mittlere* Länge eines Pfades von der Wurzel zu einem Blatt im Entscheidungsbaum in $\Omega(n \log n)$ liegt. Deshalb braucht jedes Sortierverfahren auch im Mittel $\Omega(n \log n)$ Vergleiche, wenn jede Permutation der Eingabefolge gleich wahrscheinlich ist.

2

Das Sweep-Verfahren

2.1 Einführung

In diesem Kapitel geht es um eine der vielseitigsten Techniken der Algorithmischen Geometrie: das *Sweep-Verfahren*, manchmal auch *Scan-Verfahren* genannt. Es handelt sich hierbei um ein *Paradigma*, also eine algorithmische Technik, mit deren Hilfe man viele Probleme lösen kann.

sweep als Paradigma

Ein anderes wichtiges Paradigma ist nicht nur bei geometrischen Algorithmen bekannt: *divide and conquer*. Hierbei versucht man, ein Problembeispiel P der Größe n in zwei Teile P_1 und P_2 zu zerlegen (*divide*-Schritt). Diese beiden Teilprobleme werden „direkt“ gelöst, wenn sie nur noch $O(1)$ groß sind, ansonsten durch Rekursion. Die eigentliche Aufgabe besteht darin, die Teillösungen für P_1 und P_2 zu einer Lösung für das ganze Problem P zusammenzusetzen (*conquer*-Schritt). Wenn das stets in Zeit $O(|P|)$ möglich ist, ergibt sich insgesamt eine Laufzeit von $O(n \log n)$, falls bei jedem Teilungsschritt etwa gleich große Teilprobleme P_1, P_2 entstehen. Nach diesem Prinzip funktionieren zum Beispiel Mergesort und Quicksort.

divide and conquer als Paradigma

Man sieht: Das Wesentliche am Prinzip *divide and conquer* läßt sich ziemlich knapp und präzise beschreiben. Beim Sweep-Verfahren ist eine abstrakte Definition nicht ganz so einfach, weil zwischen den Anwendungen etwas größere Unterschiede bestehen. Man könnte sagen: *sweep* verwandelt eine räumliche Dimension in eine zeitliche, indem es aus einem statischen d-dimensionalen Problem ein dynamisches $(d-1)$-dimensionales Problem macht. Wir werden gleich sehen, daß diese Formulierung zutreffend ist. Sie verrät uns aber wenig über die Arbeitsweise des Sweep-Verfahrens.

Deshalb beginnen wir nicht mit einer abstrakten Definition, sondern untersuchen typische *Beispiele* von Sweep-Algorithmen. Dabei halten wir die Augen nach Gemeinsamkeiten offen und wer-

den im Laufe dieses Kapitels eine Reihe von typischen Merkmalen dieses Verfahrens erkennen.

2.2 Sweep im Eindimensionalen

2.2.1 Das Maximum einer Menge von Objekten

Bestimmung des Maximums

Die denkbar einfachste Anwendung des Sweep-Verfahrens ist aus der Programmierung wohlvertraut: Gegeben sind n Objekte $q_1, \ldots, q_n$ aus einer geordneten Menge Q, gesucht ist ihr *Maximum*, also dasjenige $q \in Q$ mit

$$\begin{aligned} q &\geq q_i \text{ für } i = 1, \ldots, n, \\ q &= q_j \text{ für ein } j \text{ mit } 1 \leq j \leq n. \end{aligned}$$

Um das Maximum zu bestimmen, genügt es, jedes Objekt einmal „in die Hand zu nehmen“ und zu testen, ob es größer ist als das größte der bisher betrachteten Objekte. Hierdurch ergibt sich ein optimaler Algorithmus mit Laufzeit $\Theta(n)$:

```
MaxSoFar := q[1];
for j := 2 to n do
    if MaxSoFar < q[j]
    then MaxSoFar := q[j];
write("Das Maximum ist ", MaxSoFar)
```

Diese Vorgehensweise ist typisch für die Sweep-Technik. Man will eine bestimmte Eigenschaft einer Menge von Objekten ermitteln. Dazu besucht man die Objekte der Reihe nach und führt über die schon besuchten Objekte eine geeignete Information mit (hier die Variable *MaxSoFar*), aus der sich am Schluß die gesuchte Eigenschaft ergibt.

Bei diesem Beispiel ist das Problem räumlich eindimensional; wir haben ja lediglich mit einer Folge von Objekten zu tun. Der Sweep-Algorithmus macht daraus eine zeitliche Folge von räumlich nulldimensionalen Problemen: der Bestimmung des Maximums zweier Zahlen. Im Algorithmus wird die Zeitdimension durch die **for**-Schleife dargestellt.

Bei der Bestimmung des Maximums kommt es überhaupt nicht auf die Reihenfolge an, in der die Objekte besucht werden. Im allgemeinen ist aber die *Besuchsreihenfolge* für das Gelingen des *sweep* wesentlich.

Besuchsreihenfolge

2.2.2 Das dichteste Paar einer Menge von Zahlen

Betrachten wir zum Beispiel das Problem *closest pair* (vgl. Korollar 1.10) im $\mathbb{R}^1$ für n reelle Zahlen $x_1, \ldots, x_n$. Um zwei Zahlen x_i, x_j mit minimalem Abstand $d(x_i, x_j) = |x_i - x_j|$ zu bestimmen, müssen wir nur solche Paare betrachten, die der Größe nach benachbart sind. Wir *sortieren* die x_i deshalb zunächst nach wachsender Größe; die sortierte Folge

$$x'_1 \leq x'_2 \leq \ldots \leq x'_n$$

mit $x'_j = x_{\pi_{(j)}}$ ist die richtige Besuchsreihenfolge für einen *sweep*.

Während wir die x'_j in dieser Reihenfolge durchlaufen, merken wir uns in der Variablen *ClosPos* den kleinsten Index i, für den x'_{i-1} und x'_i ein dichtestes Paar unter den bisher besuchten Zahlen $x'_1, \ldots, x'_j$ bilden. Ihr Abstand heißt *MinDistSoFar*.[1]

```
MinDistSoFar := x'[2] − x'[1];
ClosPos := 2;
for j := 3 to n do
    if MinDistSoFar > x'[j] − x'[j − 1]
    then
        MinDistSoFar := x'[j] − x'[j − 1];
        ClosPos := j;
write("Ein dichtestes Paar bilden ",
                    x'[ClosPos − 1], x'[ClosPos])
```

Nach dem vorausgehenden Sortiervorgang läßt sich ein dichtestes Paar von n reellen Zahlen mit diesem *sweep* in linearer Zeit bestimmen. Die Gesamtlaufzeit der Lösung ist also in $O(n \log n)$. Wegen der unteren Schranke aus Korollar 1.10 haben wir damit folgende Aussage:

Korollar 2.1 *Ein dichtestes Paar von n reellen Zahlen zu bestimmen, hat die Zeitkomplexität $\Theta(n \log n)$.*

Übungsaufgabe 2.1 Gegeben sind n Punkte $p_i = (x_i, \sin x_i)$ auf der Sinuskurve im $\mathbb{R}^2$. Man bestimme ein dichtestes Paar.

mitzuführende Information

Bei unseren ersten beiden Beispielen war es ziemlich einfach zu entscheiden, welche Information während des *sweep* mitgeführt werden muß: Bei der Bestimmung des Maximums war es das Maximum der bisher betrachteten Objekte, bei der Berechnung des dichtesten Paars war es das dichteste Paar der bisher betrachteten Zahlen und seine Position.

[1] Oft wird man sich nur für den Abstand eines dichtesten Paars interessieren. Dann kann auf die Positionsinformation *ClosPos* verzichtet werden.

Wir werden nun ein Beispiel für eine Anwendung der Sweep-Technik kennenlernen, das ein klein wenig mehr Überlegung erfordert, aber dafür um so verblüffender ist.

2.2.3 Die maximale Teilsumme

Aktienkurs

Angenommen, jemand führt über die Kursschwankungen einer Aktie Buch. Das Auf oder Ab am i-ten Tag wird als reelle Zahl an der i-ten Stelle eines Arrays *Variation* gespeichert, für $1 \leq i \leq n$.

Eine interessante Frage lautet: Wieviel hätte man verdienen können, wenn man diese Aktie am richtigen Tag gekauft und am richtigen Tag wieder verkauft hätte? Der maximale Gewinn wird durch die maximale Summe

$$\mathit{Variation}[i] + \mathit{Variation}[i+1] + \ldots + \mathit{Variation}[j]$$

beschrieben, die sich ergibt, wenn i von 1 bis n und j von i bis n laufen, falls wenigstens eine solche Summe nicht-negativ ist; wenn dagegen eine Aktie stets fällt, kauft man sie am besten gar nicht und macht den Gewinn Null.

Ein Beispiel ist in Abbildung 2.1 zu sehen. Vom dritten bis zum siebten Tag steigt der Kurs der Aktie insgesamt um den Wert 10. Durch Ausprobieren der anderen Möglichkeiten kann man sich davon überzeugen, daß es keine lückenlose Teilfolge von Zahlen gibt, die eine größere Summe hätte. So lohnt es sich zum Beispiel nicht, die Aktie auch am achten Tag noch zu behalten, weil der Verlust von 9 später nicht mehr ausgeglichen wird.

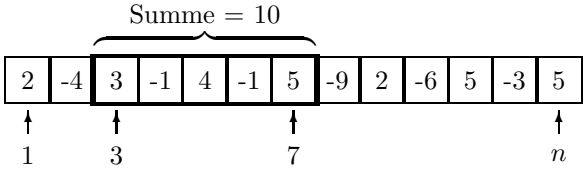


Abb. 2.1 Die maximale Teilsumme aufeinanderfolgender Zahlen hat hier den Wert 10.

maximum subvector

Wie läßt sich die maximale Teilsumme effizient berechnen? Dieses hübsche Problem ist unter dem Namen *maximum subvector* bekannt. Es findet sich in der lesenswerten Sammlung *Programming Pearls* von J. Bentley [15].

kubisch

quadratisch

Die naive Berechnung der maximalen Teilsumme führt zu einem Algorithmus mit drei geschachtelten **for**-Schleifen für i, j und einen Summationsindex k; seine Laufzeit ist in $\Theta(n^3)$. Er läßt sich zu einem $\Theta(n^2)$-Verfahren verbessern, indem man nach jeder Erhöhung von j nicht die ganze Summe von i bis $j+1$ neu

berechnet, sondern lediglich zur alten Summe den neuen Summanden *Variation*[j+1] hinzuaddiert. Auch ein Ansatz nach dem Verfahren *divide and conquer* ist möglich. Er liefert eine Laufzeit in $\Theta(n \log n)$.

Man kann aber die gesuchte maximale Teilsumme sogar in optimaler Zeit $\Theta(n)$ berechnen, wenn man das Sweep-Verfahren richtig anwendet! Die Objekte sind die Einträge

linear

$$\mathit{Variation}[1], \mathit{Variation}[2], \ldots, \mathit{Variation}[n],$$

die wir in dieser Reihenfolge besuchen. In der Variablen $\mathit{MaxSoFar}_{j-1}$ merken wir uns die bisher ermittelte maximale Teilsumme der ersten $j-1$ Zahlen.

Wodurch könnte $\mathit{MaxSoFar}_{j-1}$ übertroffen werden, wenn im nächsten Schritt nun auch *Variation*[j] besucht wird? Doch nur von einer Teilsumme der ersten j Einträge, *die den j-ten Eintrag einschließt!* Diese Situation ist in Abbildung 2.2 dargestellt. Die bisherige maximale Teilsumme der Positionen 1 bis $j-1$ hat den Wert 10; gebildet wird sie von den Einträgen an den Positionen 4 bis 7. Dieses alte Maximum wird nun „entthront“ von der Summe der Einträge an den Positionen $j-5$ bis j, die den Wert 11 hat.

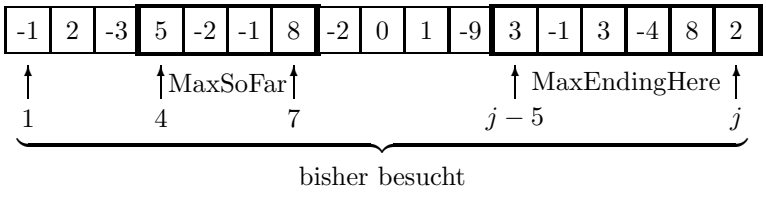


Abb. 2.2 Die bisher ermittelte maximale Teilsumme *MaxSoFar* kann nur von einer Teilsumme übertroffen werden, die sich bis zum neu hinzukommenden Eintrag *Variation*[j] erstreckt.

Um die Variable *MaxSoFar* in dieser Situation korrekt aktualisieren zu können, müssen wir während des *sweep* mehr Information mitführen als nur *MaxSoFar* selbst. Wir benötigen auch die maximale Teilsumme der schon betrachteten Einträge, die in dem zuletzt neu hinzugekommenen Eintrag endet. Mit $\mathit{MaxEndingHere}_j$ bezeichnen wir das Maximum aller Summen

mitzuführende Information

$$\mathit{Variation}[h] + \mathit{Variation}[h+1] + \ldots + \mathit{Variation}[j],$$

wobei h zwischen 1 und j läuft, falls wenigstens eine dieser Summen nicht-negativ ist; andernfalls soll $\mathit{MaxEndingHere}_j$ den Wert Null haben. Im Beispiel von Abbildung 2.2 ist $\mathit{MaxEndingHere}_j$ die Summe der letzten sechs Einträge. Dagegen wäre $\mathit{MaxEndingHere}_3 = 0$.

Wir müssen uns nun noch überlegen, wie *MaxEndingHere* aktualisiert werden soll, wenn der *sweep* von Position $j-1$ zur Position j vorrückt. Dabei hilft uns folgende einfache Beobachtung: Für festes j wird das Maximum aller Summen

$$\mathit{Variation}[h]+\mathit{Variation}[h+1]+\ldots+\mathit{Variation}[j-1]+\mathit{Variation}[j],$$

für dasselbe $h \leq j-1$ angenommen, das auch schon die Summe

$$\mathit{Variation}[h] + \mathit{Variation}[h+1] + \ldots + \mathit{Variation}[j-1]$$

maximiert hatte, es sei denn, es gibt keine positive Teilsumme, die in Position $j-1$ endet, und $\mathit{MaxEndingHere}_{j-1}$ hatte den Wert Null. In beiden Fällen gilt

$$\mathit{MaxEndingHere}_j = \max(0, \mathit{MaxEndingHere}_{j-1} + \mathit{Variation}[j]).$$

Damit haben wir folgenden Sweep-Algorithmus zur Berechnung der maximalen Teilsumme:

```
MaxSoFar := 0;
MaxEndingHere := 0;
for j := 1 to n do
    MaxEndingHere :=
                max(0, MaxEndingHere + Variation[j]);
    MaxSoFar := max(MaxSoFar, MaxEndingHere);
write("Die maximale Teilsumme beträgt ", MaxSoFar)
```

Theorem 2.2 *In einer Folge von n Zahlen die maximale Teilsumme konsekutiver Zahlen zu bestimmen, hat die Zeitkomplexität $\Theta(n)$.*

schnell und elegant

Dieser Algorithmus ist nicht nur der schnellste unter allen Mitbewerbern, sondern auch der kürzeste! Darüber hinaus benötigt er keinen wahlfreien Zugriff auf das ganze Array *Variation*, denn die Kursschwankung des j-ten Tages wird ja nur einmal angeschaut und danach nie wieder benötigt.

Man könnte deshalb dieses Verfahren als nicht-terminierenden Algorithmus implementieren, der am j-ten Tag die Schwankung $\mathit{Variation}[j]$ als Eingabe erhält und als Antwort den bis jetzt maximal erzielbaren Gewinn ausgibt; schade, daß dieser erst im nachhinein bekannt wird. Bei diesem *on-line-Betrieb* wären der Speicherplatzbedarf und die tägliche Antwortzeit konstant.

on-line-Betrieb

Übungsaufgabe 2.2 Man formuliere den oben vorgestellten Sweep-Algorithmus zur Bestimmung der maximalen Teilsumme als *on-line*-Algorithmus.

Der Sweep-Algorithmus zur Bestimmung der maximalen Teilsumme wird uns später in einem überraschenden Zusammenhang gute Dienste leisten: bei der Berechnung des *Kerns* eines Polygons, der Menge aller Punkte im Polygon, von denen aus das ganze Polygon sichtbar ist.

Kern

2.3 Sweep in der Ebene

Ihre wahre Stärke zeigt die Sweep-Technik in der Ebene. Die Objekte, die in unseren Problemen vorkommen, werden zunächst Punkte sein, später Liniensegmente und Kurven. Wir besuchen sie in der Reihenfolge, in der sie von einer senkrechten Geraden angetroffen werden, die von links nach rechts über die Ebene wandert.

Die Vorstellung dieser wandernden Geraden (*sweep line*), die die Ebene von links nach rechts „ausfegt“ und dabei keine Stelle ausläßt, hat dem Sweep-Verfahren seinen Namen gegeben.

sweep line

2.3.1 Das dichteste Punktepaar in der Ebene

Wir beginnen mit dem Problem *closest pair*, das uns aus dem Eindimensionalen schon bekannt ist. Gegeben sind n Punkte $p_1, \ldots, p_n$ in der Ebene, gesucht ist ein Paar mit minimalem euklidischem Abstand. Wir sind zunächst etwas bescheidener und bestimmen nur den minimalen Abstand

closest pair

$$\min_{1 \le i < j \le n} |p_i p_j|$$

selbst. Der Algorithmus läßt sich später leicht so erweitern, daß er auch ein dichtestes Paar ausgibt, bei dem dieser Abstand auftritt.

Erinnern wir uns an unser Vorgehen im Eindimensionalen: Die Punkte waren dort Zahlen auf der X-Achse. Wir hatten sie von links nach rechts besucht und dabei für jede Zahl den Abstand zu ihrer unmittelbaren Vorgängerin betrachtet. War er kleiner als *MinSoFar*, der bisher ermittelte minimale Abstand, mußte *MinSoFar* aktualisiert werden.

In der Ebene können wir nicht ganz so einfach verfahren. Wenn wir mit der *sweep line* auf einen neuen Punkt r treffen, kann der Abstand zu dessen direktem Vorgänger q, d. h. zu dem Punkt, der als letzter vor r von der *sweep line* erreicht wurde, viel größer sein als der Abstand von r zu noch weiter links liegenden Punkten; siehe Abbildung 2.3.

Unterschied zum Eindimensionalen

Eines aber ist klar: Wenn r mit einem Punkt p links von r ein Paar bilden will, dessen Abstand kleiner ist als *MinSoFar*, so

muß p im Innern des senkrechten Streifens der Breite *MinSoFar* liegen, dessen rechter Rand durch r läuft. Dieser Streifen ist in Abbildung 2.3 grau eingezeichnet. Während die *sweep line* vorrückt, zieht sie den senkrechten Streifen hinter sich her. Nur die Punkte im Streifen brauchen wir uns während des *sweep* zu merken; jeder von ihnen könnte zu einem Paar mit Abstand $<$ *MinSoFar* gehören, wenn die *sweep line* auf einen passenden Partner stößt.

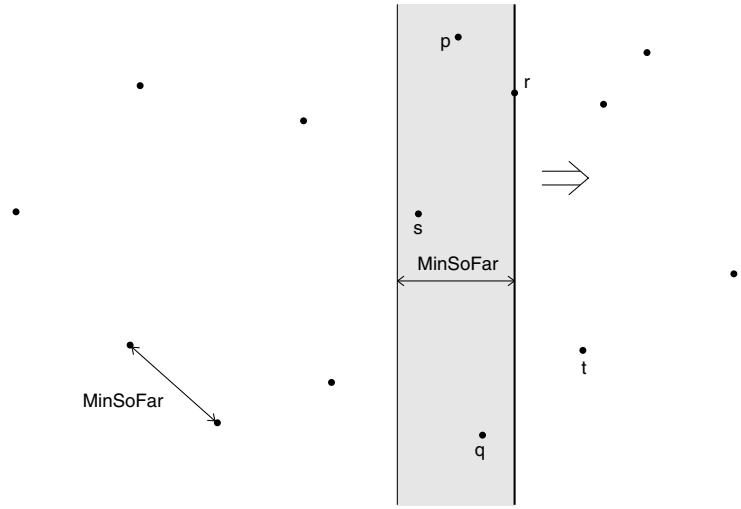


Abb. 2.3 Die *sweep line* erreicht Punkt r. Der Abstand $|pr|$ ist hier kleiner als *MinSoFar*.

Wir führen deshalb ein Verzeichnis mit, in dem zu jedem Zeitpunkt genau die Punkte im senkrechten Streifen der Breite *MinSoFar* links von der *sweep line* stehen;[2] hierdurch ist der Inhalt des Verzeichnisses eindeutig definiert.

Sweep-Status-Struktur

Weil dieses Verzeichnis die Information über „die Situation dicht hinter der *sweep line*“ enthält, trägt es den Namen *Sweep-Status-Struktur*, abgekürzt *SSS*.[3]

Ereignis

Folgende Ereignisse erfordern eine Aktualisierung der *SSS*:

(1) Der linke Streifenrand wandert über einen Punkt hinweg.

(2) Der rechte Streifenrand, also die *sweep line*, stößt auf einen neuen Punkt.

[2] Der Streifen ist links offen, d. h. der linke Rand gehört nicht dazu.

[3] Manche Autoren sagen auch *Y-Liste* dazu, weil die Objekte darin meist nach Y-Koordinaten sortiert sind. Auf solche Implementierungsfragen werden wir später noch eingehen.

Bei einem Ereignis des ersten Typs muß der betroffene Punkt aus der *SSS* entfernt werden. Findet ein Ereignis vom zweiten Typ statt, gehört der neue Punkt zum Streifen und muß in die *SSS* aufgenommen werden. Außerdem ist zu testen, ob er zu irgendeinem anderen Punkt im Streifen einen Abstand $< MinSoFar$ hat. Falls ja, müssen *MinSoFar* und die Streifenbreite auf den neuen Wert verringert werden. Der linke Streifenrand kann dabei über einen oder mehrere Punkte hinwegspringen und so eine Folge von „gleichzeitigen“ Ereignissen des ersten Typs nach sich ziehen. Wir bearbeiten sie, bevor die *sweep line* sich weiterbewegt.

Aktualisierung der *SSS*

In Abbildung 2.3 schrumpft der Streifen auf die Breite $|pr|$, wenn die *sweep line* den Punkt r erreicht; dabei springt der linke Streifenrand über den Punkt s hinweg. Während die *sweep line* danach zum Punkt t vorrückt, wandert der linke Streifenrand erst über p und dann über q hinweg.

Offenbar betreten und verlassen die Punkte den Streifen in der Reihenfolge von links nach rechts. Wir sortieren sie deshalb nach aufsteigenden X-Koordinaten und legen sie in einem Array ab. Dabei kann man testen, ob bei Punkten mit identischen X-Koordinaten auch die Y-Koordinaten übereinstimmen; in diesem Fall terminiert das Verfahren sofort und liefert den kleinsten Abstand 0. Andernfalls wissen wir, daß die n Punkte paarweise verschieden sind.

Die Indizes *links* und *rechts* verweisen jeweils auf die am weitesten links gelegenen Punkte *im* Streifen und *rechts* vom Streifen.[4] Kurz bevor die *sweep line* in Abbildung 2.3 den Punkt r erreicht, haben wir die in Abbildung 2.4 gezeigte Situation.

links und *rechts*

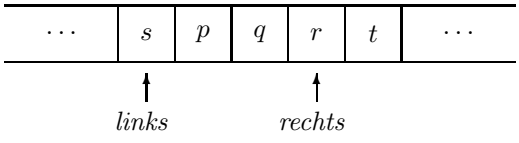


Abb. 2.4 $P[links]$ ist der am weitesten links gelegene Punkt im senkrechten Streifen, und $P[rechts]$ ist der nächste Punkt, auf den die *sweep line* treffen wird.

Ob zuerst $P[links]$ den Streifen verläßt oder $P[rechts]$ den Streifen betritt, können wir leicht feststellen. Wenn wir mit $p.x$ die X-Koordinate eines Punktes p bezeichnen, so müssen wir nur testen, ob

$$P[links].x + MinSoFar \leq P[rechts].x$$

ist. Falls ja, kommt zuerst $P[links]$ an die Reihe. Damit läßt sich der Sweep-Algorithmus wie folgt beschreiben:

[4] Sollte der Streifen gerade keinen Punkt enthalten, ist *links* gleich *rechts*.

```
(* Initialisierung *)
sortiere die n Punkte nach aufsteigenden X-Koordinaten
    und füge sie ins Array P ein;
(* alle Punkte sind paarweise verschieden *)
füge P[1], P[2] in SSS ein;
MinSoFar := |P[1]P[2]|;
links := 1;
rechts := 3;

(* sweep *)
while (* I_1 gilt *) rechts ≤ n do
    if P[links].x + MinSoFar ≤ P[rechts].x
    then (* alter Punkt verläßt Streifen *)
        entferne P[links] aus SSS;
        links := links + 1
    else (* I_2 gilt; neuer Punkt betritt Streifen *)
        MinSoFar := MinDist(SSS, P[rechts], MinSoFar);
        füge P[rechts] in SSS ein;
        rechts := rechts + 1;
(* Ausgabe *)
write("Der kleinste Abstand ist ", MinSoFar)
```

Zu spezifizieren ist noch die Funktion *MinDist*: Bei einem Aufruf

$$MinDist(SSS, r, MinSoFar)$$

wird das Minimum von *MinSoFar* und der minimalen Distanz $\min\{|pr|; p \in SSS\}$ vom neuen Punkt r auf der *sweep line* zu allen übrigen Punkten im senkrechten Streifen zurückgegeben. Wie diese Funktion implementiert wird, beschreiben wir gleich.

Korrektheit

Zunächst vergewissern wir uns, daß der Sweep-Algorithmus korrekt ist.

Übungsaufgabe 2.3 Man zeige, daß an den mit I_1 und I_2 markierten Stellen im Programm jeweils die folgenden Invarianten gelten:

I_1 : Es ist *MinSoFar* der minimale Abstand unter den Punkten $P[1], \ldots, P[rechts - 1]$.

I_2 : Die *SSS* enthält genau die Punkte $P[i]$, $1 \leq i \leq rechts - 1$, mit $P[i].x > P[rechts].x - MinSoFar$.

Implementierung von *MinDist*

Um beim Erreichen eines neuen Punktes r das Minimum $\min\{|pr|; p \in SSS\}$ zu bestimmen, genügt es, diejenigen Punkte p im Streifen zu inspizieren, deren Y-Koordinate um höchstens

MinSoFar oberhalb oder unterhalb von $r.y$ liegen, denn kein anderer Punkt kann zu r einen Abstand $< MinSoFar$ haben; siehe Abbildung 2.5.[5]

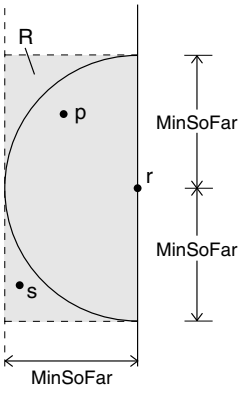


Abb. 2.5 Nur ein Punkt, der im Rechteck R liegt, kann zum Punkt r einen Abstand kleiner als *MinSoFar* haben.

Die Sweep-Status-Struktur *SSS* muß es also gestatten, Punkte einzufügen und zu entfernen und beim Aufruf von *MinDist* für beliebige Zahlen y_0 und M alle gespeicherten Punkte p mit

$$y_0 - M \leq p.y \leq y_0 + M$$

zu berichten; für $y_0 = r.y$ und $M = MinSoFar$ sind das genau die Punkte im Rechteck R in Abbildung 2.5. Diese Anfrage heißt *Bereichsanfrage*.

Bereichsanfrage

Offenbar kommt es hierbei auf die X-Koordinaten der Punkte gar nicht an! Wir können deshalb zur Implementierung der *SSS* eine eindimensionale Datenstruktur verwenden, die nur die Y-Koordinaten unserer Punkte „kennt“. Dazu nehmen wir einen nach Y-Koordinaten geordneten balancierten Binärbaum, dessen Blätter zu einer Liste verkettet sind[6] und die zu speichernden Punkte enthalten. Man kann zum Beispiel AVL-Bäume verwenden, wie sie in Büchern über Datenstrukturen vorgestellt werden. Damit lassen sich die Operationen Einfügen und Entfernen jeweils in Zeit $O(\log n)$ ausführen, und die Bereichsanfrage erfolgt in Zeit $O(\log n + k)$, wobei k die Anzahl der berichteten Punkte bedeutet, die sogenannte *Größe der Antwort*.

Implementierung der *SSS*

Laufzeitananalyse

Größe der Antwort

[5] Eigentlich würde es genügen, den Halbkreis um r mit Radius *MinSoFar* abzusuchen, aber dazu fehlt uns eine effiziente Methode. Statt dessen inspizieren wir das den Halbkreis umschließende Rechteck R und nehmen in Kauf, dabei auch auf Punkte s mit $|sr| > MinSoFar$ zu stoßen.

[6] Daß es auch ohne eine Verkettung der Blätter geht, wird sich in Abschnitt 3.3.2 auf Seite 136 zeigen.

Weil die **while**-Schleife nur $O(n)$-mal durchlaufen wird, ergibt sich eine Gesamtlaufzeit in

$$O\left(n \log n + \sum_{i=3}^{n} k_i\right);$$

hierbei ist k_i die Größe der Antwort der i-ten Bereichsanfrage beim Einfügen des i-ten Punkts.

Wie groß kann die Summe der k_i werden? Natürlich ist $k_i \leq i$, aber diese Abschätzung liefert uns nur eine obere Schranke in $O(n^2)$. Wäre das Laufzeitverhalten tatsächlich so schlecht, so hätten wir ebensogut auf den *sweep* verzichten und gleich alle Paare von Punkten inspizieren können!

strukturelle Eigenschaft

Bei der Verbesserung dieser Abschätzung kommt uns folgende strukturelle geometrische Überlegung zu Hilfe: Wenn ein Aufruf *MinDist*$(SSS, r, MinSoFar)$ eine Bereichsanfrage für das in Abbildung 2.5 gezeigte Rechteck R auslöst, so haben alle Punkte links von der *sweep line*, also insbesondere diejenigen im Rechteck R, mindestens den Abstand *MinSoFar* voneinander. Wenn aber die Punkte so „lose gepackt“ sind, passen nicht allzu viele in das Rechteck R hinein!

Lemma 2.3 *Sei $M > 0$ und P eine Menge von Punkten in der Ebene, von denen je zwei mindestens den Abstand M voneinander haben. Dann enthält ein Rechteck mit den Kantenlängen M und $2M$ höchstens 10 Punkte aus P.*

Beweis. Nach Voraussetzung sind die Kreisumgebungen $U_{M/2}(p)$ der Punkte p aus P paarweise disjunkt. Liegt p im Rechteck R, so ist mindestens ein Viertel der offenen Kreisscheibe in R enthalten. Durch Flächenberechnung ergibt sich, daß R höchstens

$$\frac{\mathit{Fläche(R)}}{\mathit{Fläche(Viertelkreis)}} = \frac{2M^2}{\frac{1}{4}\pi(\frac{M}{2})^2} = \frac{32}{\pi} < 11$$

viele Punkte von P enthalten kann. □

Folglich ist jedes k_i kleiner gleich 10, und wir erhalten folgenden Satz:

Theorem 2.4 *Der minimale Abstand aller Paare einer n-elementigen Punktmenge in der Ebene läßt sich in Zeit $O(n \log n)$ bestimmen, und das ist optimal.*

Die Optimalität folgt wie beim Beweis von Korollar 1.9. Indem wir uns außer dem Wert von *MinSoFar* auch merken, zwischen welchen beiden bisher betrachteten Punkten der minimale Abstand aufgetreten ist, erhalten wir als Folgerung:

Korollar 2.5 *Ein dichtestes Paar von n Punkten in der Ebene zu bestimmen, hat die Zeitkomplexität $\Theta(n \log n)$.*

Damit ist eines der in Kapitel 1 vorgestellten Probleme gelöst. Wir haben uns dabei auf zwei geometrische Sachverhalte gestützt. Der erste ist trivial: Wenn der minimale Abstand M in einer Punktmenge P durch Hinzufügen eines neuen Punktes r abnimmt, so müssen die hierfür verantwortlichen Punkte von P nah (d. h. im Abstand $< M$) bei r liegen.

strukturelle Eigenschaft

Unsere zweite geometrische Stütze ist Lemma 2.3. Es besagt im wesentlichen: Eine beschränkte Menge kann nicht beliebig viele Punkte enthalten, die zueinander einen festen Mindestabstand haben.

Die genaue Anzahl von Punkten, die der rechteckige Anfragebereich R höchstens enthalten kann, geht natürlich in den Faktor in der O-Notation für die Laufzeitabschätzung ein. Dies ist Gegenstand folgender Übungsaufgabe.

Übungsaufgabe 2.4 Man bestimme die kleinste Konstante ≤ 10, für die Lemma 2.3 richtig bleibt.

Daß man das *closest-pair*-Problem mit einem derart einfachen Sweep-Algorithmus lösen kann, wurde von Hinrichs et al. [74] beobachtet.

Nebenbei haben wir eine Reihe weiterer Eigenschaften kennengelernt, die für Sweep-Algorithmen charakteristisch sind: Während des *sweep* wird eine Sweep-Status-Struktur mitgeführt, die die „Situation dicht hinter der *sweep line*“ beschreibt. Der Inhalt dieser Struktur muß eine bestimmte *Invariante* erfüllen.

charakteristische Sweep-Eigenschaften

Invariante

Jeder Anlaß für eine Veränderung der *SSS* heißt ein *Ereignis*. Obwohl wir uns vorstellen, daß die *sweep line* sich stetig mit der Zeit vorwärtsbewegt, braucht der Algorithmus nur die diskrete Menge der Ereignisse zu bearbeiten – dazwischen gibt es nichts zu tun! Es kommt darauf an, die Ereignisse vollständig und in der richtigen Reihenfolge zu behandeln und die *SSS* jedesmal korrekt zu aktualisieren. Will man die Korrektheit eines Sweep-Algorithmus formal beweisen, muß man zeigen, daß die Invariante nach der Bearbeitung von Ereignissen wieder erfüllt ist.

Ereignis

Die Objekte, mit denen ein Sweep-Algorithmus arbeitet, durchlaufen während des *sweep* eine bestimmte Entwicklung:

Solange ein Objekt noch rechts von der *sweep line* liegt, ist es ein *schlafendes Objekt*. Sobald es von der *sweep line* berührt wird, *wacht es auf* und wird ein *aktives Objekt*. Genau die aktiven Objekte sind in der *SSS* gespeichert. Wenn feststeht, daß ein Objekt

schlafende, aktive und tote Objekte

nie wieder gebraucht wird, kann es aus der *SSS* entfernt werden. Danach ist es ein *totes Objekt*.

Raum vs. Zeit

Im $\mathbb{R}^2$ verwandelt *sweep* ein zweidimensionales Problem in eine Folge von eindimensionalen Problemen: die Verwaltung der Sweep-Status-Struktur. Hierzu wird meist eine dynamische eindimensionale Datenstruktur eingesetzt.

Obwohl es in diesem Kapitel um den *sweep* geht, wollen wir andere Techniken nicht ganz aus dem Auge verlieren:

Übungsaufgabe 2.5 Man skizziere, wie sich der kleinste Abstand zwischen n Punkten in der Ebene in Zeit $O(n \log n)$ mit dem Verfahren *divide and conquer* bestimmen läßt.

2.3.2 Schnittpunkte von Liniensegmenten

ein Klassiker

In diesem Abschnitt kommen wir zu dem Klassiker unter den Sweep-Algorithmen, der ganz am Anfang der Entwicklung dieser Technik in der Algorithmischen Geometrie stand; siehe Bentley und Ottmann [16]. Gegeben sind n Liniensegmente $s_i = l_i r_i, 1 \leq i \leq n$, in der Ebene. Gemäß unserer Definition in Abschnitt 1.2.3 gehören die Endpunkte zum Liniensegment dazu; zwei Liniensegmente können deshalb auf verschiedene Weisen einen Punkt als Durchschnitt haben, wie Abbildung 2.6 zeigt. Uns interessiert nur der Fall (i), wo der Durchschnitt nur aus einem Punkt p besteht,

echte Schnittpunkte

der zum relativen Inneren beider Segmente gehört. Wir nennen solche Punkte p *echte Schnittpunkte*.[7]

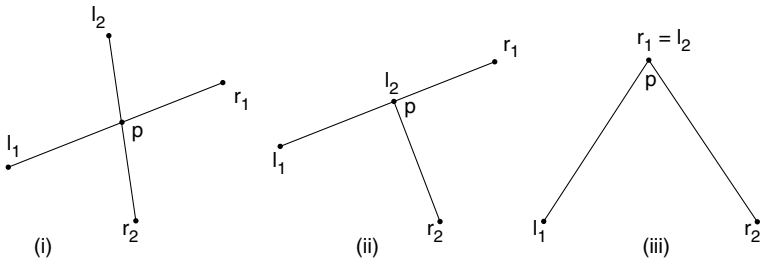


Abb. 2.6 Nur in (i) ist p ein echter Schnittpunkt.

Existenzproblem

Zwei Probleme sollen untersucht werden: Beim *Existenzproblem* besteht die Aufgabe darin, herauszufinden, ob es unter

[7] Wir können einen echten Schnittpunkt p von zwei Liniensegmenten auch dadurch charakterisieren, daß sich die Segmente in p *kreuzen*, das heißt, daß jede hinreichend kleine ε-Umgebung von p durch die beiden Segmente in vier Gebiete zerteilt wird. Diese Definition läßt sich von Liniensegmenten auf Wege verallgemeinern.

den n Segmenten zwei gibt, die einen echten Schnittpunkt haben; beim *Aufzählungsproblem* sollen *alle* echten Schnittpunkte bestimmt werden.

Aufzählungsproblem

Untere Schranken für diese Probleme können wir leicht herleiten.

Lemma 2.6 *Herauszufinden, ob zwischen n Liniensegmenten in der Ebene ein echter Schnittpunkt existiert, hat die Zeitkomplexität $\Omega(n \log n)$. Alle k Schnittpunkte aufzuzählen, hat die Zeitkomplexität $\Omega(n \log n + k)$.*

Beweis. Wir zeigen, daß man das Problem *ε-closeness* (siehe Korollar 1.6 auf Seite 40) in linearer Zeit auf das Existenzproblem reduzieren kann. Seien also n reelle Zahlen $x_1, \ldots, x_n$ und ein $\varepsilon > 0$ gegeben. Wir bilden die Liniensegmente s_i von $(x_i, 0)$ nach $(x_i + \frac{\varepsilon}{2}, \frac{\varepsilon}{2})$ und t_i von $(x_i, 0)$ nach $(x_i - \frac{\varepsilon}{2}, \frac{\varepsilon}{2})$, siehe Abbildung 2.7.

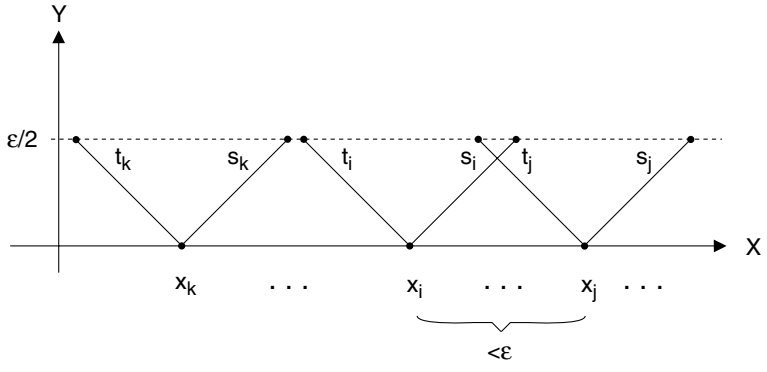


Abb. 2.7 Die Liniensegmente s_i und t_j haben nur dann einen echten Schnittpunkt, wenn $0 < |x_i - x_j| < \varepsilon$ ist.

Parallele Liniensegmente können keinen echten Schnittpunkt miteinander haben, selbst dann nicht, wenn sie übereinander liegen. Ein echter Schnittpunkt kann nur von zwei Segmenten s_i und t_j mit $i \neq j$ herrühren, für die dann $0 < |x_i - x_j| < \varepsilon$ gelten muß. Umgekehrt verursacht jedes solche Zahlenpaar einen echten Schnittpunkt von s_i mit t_j oder von s_j mit t_i.

Um das *ε-closeness*-Problem zu lösen, testen wir also zunächst, ob bei diesen Segmenten ein echter Schnittpunkt vorliegt. Falls ja, sind wir fertig. Falls nein, wissen wir, daß alle Zahlen x_i, x_j, die voneinander verschieden sind, mindestens den Abstand ε voneinander haben. Es könnte aber $i \neq j$ geben mit $x_i = x_j$.

erster Schnittpunkttest

zweiter Schnittpunkttest

Um das herauszufinden, wird ein zweiter Schnittpunkttest durchgeführt. Zu jedem x_i konstruieren wir ein Liniensegment u_i der Länge ε mit Mittelpunkt in $(x_i, \frac{\varepsilon}{2})$. Jedes u_i erhält einen anderen Winkel in bezug auf die X-Achse, der nur von seinem Index i abhängt, zum Beispiel den Winkel $\frac{i}{2n}\pi$ wie in Abbildung 2.8. Weil verschiedene Punkte $x_i \neq x_j$ mindestens den Abstand ε voneinander haben, können sich ihre Segmente u_i, u_j nicht schneiden. Genau dann haben also u_i und u_j mit $i \neq j$ einen echten Schnittpunkt, wenn $x_i = x_j$ gilt.

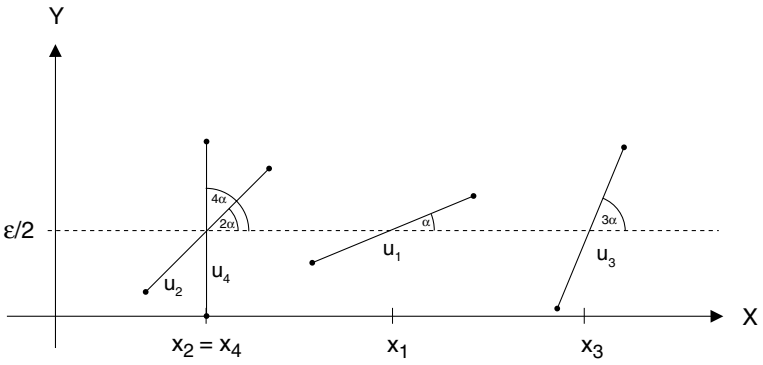


Abb. 2.8 Die Segmente u_2 und u_4 haben einen echten Schnittpunkt, weil $x_2 = x_4$ gilt. Bei vier Segmenten hat α den Wert $\pi/8$.

Die Antwort auf den zweiten Test ist nun entscheidend. Lautet sie ebenfalls „nein“, so gibt es keine zwei Zahlen x_i, x_j mit $i \neq j$ und $|x_i - x_j| < \varepsilon$.

Hieraus folgt, daß das Existenzproblem für echte Schnittpunkte mindestens so schwierig ist wie *ε-closeness*, d. h. seine Zeitkomplexität ist $\Omega(n \log n)$. Mit der Beantwortung des Aufzählungsproblems hat man insbesondere das Existenzproblem gelöst. Außerdem müssen alle k Schnittpunkte berichtet werden. Insgesamt wird also $\Omega(n \log n + k)$ Zeit benötigt. □

Jedes der beiden in Lemma 2.6 angesprochenen Probleme kann man naiv in Zeit $O(n^2)$ lösen, indem man je zwei Liniensegmente hernimmt und feststellt, ob sie einen echten Schnittpunkt haben. Im schlimmsten Fall kann jedes Segment jedes andere schneiden, so daß $k = \frac{n(n-1)}{2}$ ist. Bedeutet das etwa, daß der naive Algorithmus zur Lösung des Aufzählungsproblems optimal ist?

Wenn man nur die Anzahl n der Segmente als Problemparameter (siehe Abschnitt 1.2.4) zuläßt, ist diese Frage zu bejahen. Wir erwarten aber doch, daß die Rechenzeit eines Lösungsverfahrens nur dann groß wird, wenn auch k groß ist. Dieses erstrebenswerte Verhalten eines Algorithmus nennt man *Output-Sensitivität*. Mit

Output-Sensitivität

der Einführung von k als zweitem Problemparameter haben wir ein Kriterium für die Output-Sensitivität von guten Algorithmen. Das naive Verfahren hat diese Eigenschaft nicht, denn es benötigt immer $\Theta(n^2)$ viele Schritte, auch für $k = 0$.

Wir wollen nun effiziente Sweep-Algorithmen zur Lösung des Existenz- und des Aufzählungsproblems angeben. Unsere Objekte sind die n Liniensegmente s_i. Wir nehmen an, daß keines von ihnen senkrecht ist; der linke Endpunkt ist l_i, der rechte r_i. Außerdem sei der Durchschnitt von zwei Liniensegmenten entweder leer oder ein einzelner Punkt. Ferner sollen sich nie mehr als zwei Segmente in einem Punkt schneiden, und alle Endpunkte und Schnittpunkte sollen paarweise verschiedene X-Koordinaten haben.[8]

vereinfachende Annahmen

Die *sweep line* bewegt sich, wie üblich, von links nach rechts über die Ebene. Den Augenblick, in dem sie die X-Achse in x_0 schneidet, nennen wir den *Zeitpunkt* x_0.

Zeitpunkt

Auf denjenigen Liniensegmenten, die die *sweep line* zum Zeitpunkt x_0 gerade schneidet, definiert sie eine Ordnung entsprechend den aufsteigenden Y-Koordinaten der Schnittpunkte. In Abbildung 2.9 haben wir zum Beispiel zur Zeit x_1 die Ordnung $s_3 < s_1 < s_4 < s_2$. Diese Ordnung wird in der Sweep-Status-Struktur *SSS* mitgeführt.

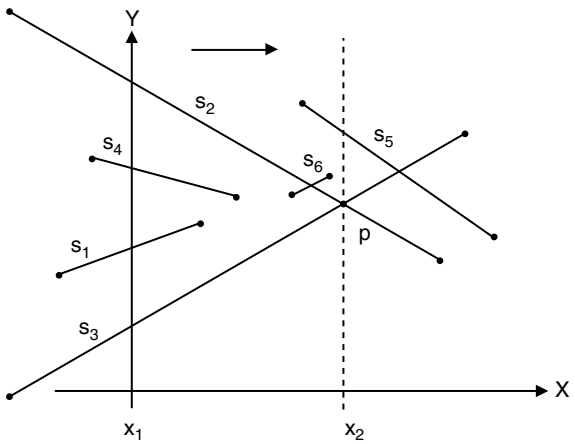


Abb. 2.9 Die von der *sweep line* geschnittenen Liniensegmente sind nach den y-Werten der Schnittpunkte geordnet.

Folgende *Ereignisse* verändern die Ordnung der Segmente längs der *sweep line* und erfordern eine Aktualisierung der *SSS*:

Ereignisse

[8] Später machen wir uns von diesen Annahmen frei.

(1) Die *sweep line* stößt auf den linken Endpunkt eines Liniensegments,

(2) die *sweep line* erreicht den rechten Endpunkt eines Liniensegments,

(3) die *sweep line* erreicht einen echten Schnittpunkt von zwei Liniensegmenten.

Um die Ereignisse (1) und (2) in der richtigen Reihenfolge bearbeiten zu können, sortieren wir vor Beginn des *sweep* alle Endpunkte l_i, r_i nach ihren X-Koordinaten. Wie aber sollen wir die Ereignisse (3) rechtzeitig erkennen? (Die Schnittpunkte sollen ja erst noch bestimmt werden!)

strukturelle Eigenschaft

Zum Glück können wir uns auf folgende geometrische Eigenschaft stützen:

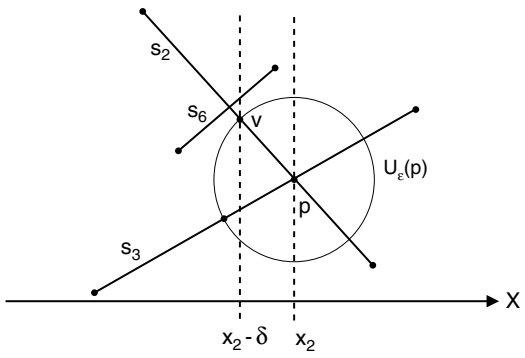


Abb. 2.10 Kurz vor ihrem Schnittpunkt p sind s_2 und s_3 benachbart.

Lemma 2.7 *Wenn zwei Liniensegmente einen echten Schnittpunkt haben, so sind sie unmittelbar vorher direkte Nachbarn in der Ordnung längs der sweep line.*

Beweis. Sei p echter Schnittpunkt der beiden Segmente s_2 und s_3 mit X-Koordinate x_2; siehe Abbildung 2.9. Weil nach unserer Annahme keines der anderen $n-2$ Liniensegmente den Punkt p enthält, gibt es ein $\varepsilon > 0$, so daß die Umgebung $U_\varepsilon(p)$ von s_2 und s_3 durchschnitten, aber von keinem anderen Segment geschnitten wird. Die Segmente s_2 und s_3 schneiden den Rand von $U_\varepsilon(p)$ links von p. Sei v der weiter rechts gelegene der beiden Schnittpunkte, und sei $x_2 - \delta$ seine X-Koordinate; siehe Abbildung 2.10. Dann sind für jeden Zeitpunkt $x \in (x_2-\delta, x_2)$ die Segmente s_2, s_3 direkte

Nachbarn in der Ordnung längs der *sweep line*, also auch Nachbarn in der *SSS* spätestens nach dem letzten Ereignis vor Erreichen von p. □

Wir werden also keinen echten Schnittpunkt verpassen, wenn wir dafür sorgen, daß zwei Liniensegmente sofort auf Schnitt getestet werden, sobald sie in der Ordnung längs der *sweep line* zu direkten Nachbarn werden!

direkte Nachbarn testen

Bei der Behandlung eines Ereignisses (1) muß zunächst das neue Liniensegment s an der richtigen Stelle in die aktuelle Ordnung eingefügt werden. Sind danach die Segmente s_i und s_j der direkte Vorgänger und der direkte Nachfolger von s längs der *sweep line*, so muß s mit s_i und mit s_j auf echten Schnitt getestet werden. In Abbildung 2.9 hat zum Beispiel s_4 den Vorgänger s_1 und den Nachfolger s_2, wenn die *sweep line* seinen linken Endpunkt erreicht hat.

Wenn die *sweep line* am rechten Endpunkt eines Segments s angekommen ist (Ereignis (2)), wird s aus der Sweep-Status-Struktur entfernt. Sein früherer direkter Vorgänger und sein früherer direkter Nachfolger sind danach selbst direkte Nachbarn und müssen auf echten Schnitt getestet werden. Beim Entfernen von s_1 in Abbildung 2.9 sind das die Segmente s_3 und s_4.

Besitzt ein Segment keinen Vorgänger oder Nachfolger in der aktuellen Ordnung, fällt der entsprechende Schnittest weg.

Wenn wir uns nur für die Existenz eines echten Schnittpunkts interessieren, ist unser Sweep-Algorithmus sehr einfach. Wir sortieren anfangs die $2n$ Endpunkte der Liniensegmente nach aufsteigenden X-Koordinaten und speichern sie in einem Array. Während des *sweep* bearbeiten wir die zugehörigen Ereignisse (1) oder (2) in dieser Reihenfolge, wie oben beschrieben. Sobald der erste Schnittest ein positives Ergebnis liefert, können wir den *sweep* beenden. Ereignisse (3) brauchen hierbei nicht behandelt zu werden. Die Sweep-Status-Struktur muß folgende Operationen erlauben:

Lösung des Existenzproblems

FügeEin(*SSS,Seg,x*):	Fügt Segment *Seg* entsprechend der Ordnung zur Zeit x in die *SSS* ein.
Entferne(*SSS,Seg,x*):	Entfernt Segment *Seg* aus der *SSS* zum Zeitpunkt x.
Vorg(*SSS,Seg,x*):	Bestimmt den Vorgänger von *Seg* in der *SSS* zum Zeitpunkt x.
Nachf(*SSS,Seg,x*):	Bestimmt den Nachfolger von *Seg* in der *SSS* zum Zeitpunkt x.

Auch hier können wir zur Implementierung der *SSS* einen balancierten Binärbaum mit verketteten Blättern verwenden. Ge-

speichert werden darin – durch Angabe der Endpunkte – die Liniensegmente entsprechend ihrer Ordnung längs der *sweep line*.

Um die bei jedem Zugriff erforderlichen Größenvergleiche zwischen Liniensegmenten auszuführen, werden die zum momentanen X-Wert zugehörigen Y-Koordinaten berechnet und miteinander verglichen. Aus diesem Grund ist es notwendig, bei jeder Operation auf der *SSS* den Zeitparameter x zu übergeben.

Jede der Operation ist in Zeit $O(\log n)$ ausführbar, weil höchstens n Liniensegmente gleichzeitig in der *SSS* gespeichert sind. Da die Bearbeitung von jedem der (höchstens) $2n$ vielen Ereignisse nur drei solche Operationen erfordert, ergibt sich folgende Aussage:

Theorem 2.8 *Man kann in optimaler Zeit $O(n \log n)$ und mit Speicherplatz $O(n)$ herausfinden, ob von n Liniensegmenten in der Ebene mindestens zwei einen echten Schnittpunkt haben.*

Beweis. Daß $O(n \log n)$ als Zeitschranke optimal ist, folgt aus Lemma 2.6. □

Wenn man davon ausgeht, daß die n Liniensegmente ohnehin gleichzeitig im Speicher gehalten werden müssen, ist auch die lineare Schranke für den Speicherplatzbedarf optimal.

Übungsaufgabe 2.6 Entdeckt der Sweep-Algorithmus den am weitesten links liegenden echten Schnittpunkt von n Liniensegmenten immer als ersten?

Lösung des Aufzählungsproblems

Alle echten Schnittpunkte zu berichten, ist ein etwas anspruchsvolleres Problem. Der gravierende Unterschied wird schon bei der Bearbeitung der Ereignisse (1) und (2) deutlich: Wenn ein Schnittest von zwei direkten Nachbarn positiv verlaufen ist, müssen wir uns diesen echten Schnittpunkt als zukünftiges Ereignis (3) vormerken und zum richtigen Zeitpunkt behandeln! Dort wechseln die beiden sich schneidenden Liniensegmente in der Ordnung längs der *sweep line* ihre Plätze; siehe zum Beispiel s_3 und s_2 zur Zeit x_2 in Abbildung 2.9. Zur Ausführung der Vertauschung zweier benachbarter Segmente *USeg* und *OSeg* zur Zeit x dient die Operation *Vertausche*(*SSS,USeg,OSeg,x*).

Schnittereignisse vormerken

Danach hat jedes von beiden Segmenten einen neuen nächsten Nachbarn, mit dem auf Schnitt zu testen ist. So hat zum Beispiel s_3 nach der Vertauschung mit s_2 den direkten Nachfolger s_5. Wird ein echter Schnittpunkt festgestellt, wie hier zwischen s_3 und s_5, muß er als zukünftiges Ereignis (3) vorgemerkt werden.

Beim Aufzählungsproblem sind also die Ereignisse, die während des *sweep* zu behandeln sind, nicht alle schon am Anfang bekannt; sie ergeben sich vielmehr erst nach und nach zur

dynamische Ereignisstruktur

Laufzeit. Zu ihrer Verwaltung benötigen wir eine *dynamische Ereignisstruktur ES*, in der zukünftige Ereignisse in ihrer zeitlichen Reihenfolge verzeichnet sind. Außer der Initialisierung und dem Test, ob *ES* leer ist, sind folgende Operationen möglich:

FügeEin(*ES,Ereignis*):	Fügt Ereignis *Ereignis* entsprechend seiner Eintrittszeit in die *ES* ein.
NächstesEreignis(*ES*):	Liefert das zeitlich erste in *ES* gespeicherte Ereignis und entfernt es aus der *ES*.

Ereignisse

Ereignisse stellen wir uns dabei als Objekte folgenden Typs vor:

```
type tEreignis = record
    Zeit: real;
    case Typ: (LinkerEndpunkt,
                   RechterEndpunkt, Schnittpunkt) of
        LinkerEndpunkt, RechterEndpunkt:
            Seg: tSegment;
        Schnittpunkt:
            USeg, OSeg: tSegment;
```

Für ein solches *Ereignis* gibt *Ereignis.Typ* an, ob es sich um den linken oder rechten Endpunkt des Liniensegments *Ereignis.Seg* handelt, oder um einen Schnittpunkt; in diesem Fall bezeichnet *Ereignis.USeg* das untere und *Ereignis.OSeg* das obere der beiden beteiligten Liniensegmente unmittelbar links vom Schnittpunkt.

Die Komponente *Ereignis.Zeit* bezeichnet schließlich den Zeitpunkt, zu dem das Ereignis eintreten wird, also die X-Koordinate des linken oder rechten Endpunkts von *Ereignis.Seg* oder die X-Koordinate des Schnittpunkts von *Ereignis.USeg* und *Ereignis.OSeg*.

Aufzählen aller Schnittpunkte

Einen Sweep-Algorithmus zum Aufzählen aller echten Schnittpunkte von n Liniensegmenten können wir nun folgendermaßen skizzieren:

```
(* Initialisierung *)
initialisiere die Strukturen SSS und ES;
sortiere die 2n Endpunkte nach aufsteigenden X-Koordinaten;
erzeuge daraus Ereignisse;
füge diese Ereignisse in die ES ein;
```

```
(* sweep und Ausgabe *)
while ES ≠ ∅ do
    Ereignis := NächstesEreignis(ES);
    with Ereignis do
        case Typ of
            LinkerEndpunkt:
                FügeEin(SSS, Seg, Zeit);
                VSeg := Vorg(SSS, Seg, Zeit);
                TesteSchnittErzeugeEreignis(VSeg, Seg);
                NSeg := Nachf(SSS, Seg, Zeit);
                TesteSchnittErzeugeEreignis(Seg, NSeg);
            RechterEndpunkt:
                VSeg := Vorg(SSS, Seg, Zeit);
                NSeg := Nachf(SSS, Seg, Zeit);
                Entferne(SSS, Seg, Zeit);
                TesteSchnittErzeugeEreignis(VSeg, NSeg);
            Schnittpunkt:
                Berichte (USeg, OSeg) als Paar mit Schnitt;
                Vertausche(SSS, USeg, OSeg, Zeit);
                VSeg := Vorg(SSS, OSeg, Zeit);
                TesteSchnittErzeugeEreignis(VSeg, OSeg);
                NSeg := Nachf(SSS, USeg, Zeit);
                TesteSchnittErzeugeEreignis(USeg, NSeg)
```

Beim Aufruf der Prozedur *TesteSchnittErzeugeEreignis*(S, T) wird zunächst getestet, ob die Liniensegmente S und T einen echten Schnittpunkt besitzen. Falls ja, wird ein neues Ereignis vom Typ *Schnittpunkt* erzeugt, mit $USeg = S$, $OSeg = T$ und der X-Koordinate des Schnittpunkts als *Zeit*. Dieses Ereignis wird dann in die Ereignisstruktur *ES* eingefügt.

Ereignisstruktur

Zur Implementierung der *ES* können wir eine *Warteschlange* (*priority queue*) verwenden, auf der jede der Operationen *FügeEin* und *NächstesEreignis* in Zeit $O(\log h)$ ausführbar ist, wobei h die Anzahl der gespeicherten Ereignisse bedeutet (wer mag, kann auch hierfür einen balancierten Binärbaum nehmen).

Übungsaufgabe 2.7 Kann es vorkommen, daß derselbe Schnittpunkt während des *sweep* mehrere Male entdeckt wird?

Theorem 2.9 *Mit dem oben angegebenen Sweep-Verfahren kann man die k echten Schnittpunkte von n Liniensegmenten in Zeit $O((n+k)\log n)$ und Speicherplatz $O(n+k)$ bestimmen.*

Laufzeit $O((n+k)\log n)$

Beweis. Insgesamt gibt es $2n+k$ Ereignisse. Weil keines von ihnen mehrfach in der Ereignisstruktur vorkommt[9], sind niemals mehr

[9] Jeder Versuch, ein schon vorhandenes Objekt erneut einzufügen, wird von der Struktur abgewiesen.

als $O(n^2)$ Ereignisse in der *ES* gespeichert. Jeder Zugriff auf die *ES* ist also in Zeit $O(\log n^2) = O(\log n)$ ausführbar.

Die **while**-Schleife wird $(2n + k)$-mal durchlaufen; bei jedem Durchlauf werden höchstens sieben Operationen auf der *SSS* oder der *ES* ausgeführt. Daraus folgt die Behauptung. □

In Wahrheit ist die Schranke $O(n+k)$ für den Speicherplatzbedarf der Ereignisstruktur zu grob! Zwar können für ein einzelnes Liniensegment $n-1$ zukünftige Schnittereignisse bekannt sein, an denen das Segment beteiligt ist, aber diese Situation kann nicht für alle Liniensegmente gleichzeitig eintreten. Pach und Sharir [119] konnten zeigen, daß der Umfang der Ereignisstruktur, und damit der gesamte Speicherplatzbedarf des oben beschriebenen Sweep-Algorithmus, durch $O(n(\log n)^2)$ beschränkt ist. In der folgenden Übungsaufgabe 2.8 wird ein Beispiel aus der zitierten Arbeit behandelt, bei dem für jedes aktive Segment immerhin $\Omega(\log n)$ viele künftige Ereignisse bekannt sind, insgesamt also $\Omega(n \log n)$ viele.

zu viele künftige Ereignisse

Übungsaufgabe 2.8 Es sei L eine Menge von n Liniensegmenten, die sich von der Senkrechten S_0 nach rechts über die Senkrechte S_1 hinaus erstrecken; siehe die symbolische Darstellung in Abbildung 2.11.[10] Angenommen, zwischen S_0 und S_1 entdeckt der Sweep-Algorithmus k echte Schnittpunkte zwischen Segmenten aus L, welche rechts von S_1 liegen. Wir konstruieren nun aus L eine zweite Menge L' in folgender Weise: Zuerst fixieren wir die Schnittpunkte der Segmente aus L mit S_1 und verschieben ihre Anfangspunkte auf S_0 alle um den Betrag C nach oben (Scherung). Dann werden die so gescherten Segmente durch eine Translation um den Betrag ε nach oben verschoben.

(i) Man zeige, daß bei passender Wahl von C und ε der Sweep-Algorithmus zwischen S_0 und S_1 mindestens $2k+n$ echte Schnittpunkte zwischen Segmenten der Menge $L \cup L'$ entdeckt, die rechts von S_1 liegen.

(ii) Man folgere, daß es Konfigurationen von n Liniensegmenten gibt, bei denen die Ereignisstruktur während des *sweep* $\Omega(n \log n)$ viele künftige Ereignisse enthält.

Es gibt aber eine ganz einfache Möglichkeit, den Speicherplatzbedarf der Ereignisstruktur zu reduzieren: Man merkt sich nur die Schnittpunkte zwischen solchen Liniensegmenten, die gegenwärtig in der *SSS* benachbart sind; davon kann es zu jedem Zeitpunkt höchstens $n-1$ viele geben. Lemma 2.7 stellt sicher, daß dabei kein Schnittereignis übersehen wird: Dicht vor dem Schnittpunkt müssen die beteiligten Segmente ja benachbart sein!

Speicherersparnisregel

[10] Die Segmente sollen nach rechts genügend lang sein, so daß wir sie im Prinzip wie Halbgeraden behandeln können.

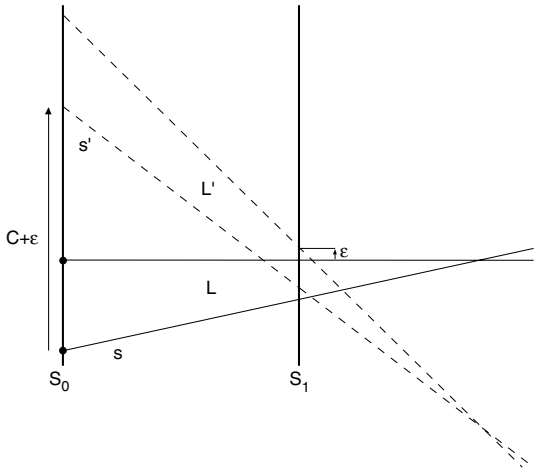


Abb. 2.11 Die Segmentmenge L' entsteht aus L durch Scherung um die Distanz C und Verschiebung um ε nach oben.

Übungsaufgabe 2.9 Was ist zu tun, um diesen Ansatz zu implementieren?

Damit haben wir folgende Verbesserung von Theorem 2.9:

Speicherbedarf $O(n)$

Theorem 2.10 *Die k echten Schnittpunkte von n Liniensegmenten in der Ebene lassen sich in Zeit $O((n+k)\log n)$ bestimmen, unter Benutzung von $O(n)$ Speicherplatz.*

Im Hinblick auf Lemma 2.6 ist Theorem 2.9 etwas unbefriedigend, weil zwischen unterer und oberer Schranke für die Zeitkomplexität eine Lücke klafft. Die Frage nach der genauen Komplexität dieses Problems erwies sich als schwieriger und wurde erst später endgültig beantwortet. Chazelle und Edelsbrunner [27] gaben 1992 einen Algorithmus an, der die k Schnittpunkte von n Liniensegmenten in optimaler Zeit $O(n \log n + k)$ berichtet, allerdings mit Speicherplatzbedarf $O(n + k)$. Ein einfacher, randomisierter Algorithmus mit gleicher Laufzeit findet sich im Buch von Mulmuley [107]. Balaban [12] schlug 1995 ein Verfahren vor, bei dem die optimale Laufzeit bei einem Speicherverbrauch von $O(n)$ erreicht wird.

optimale Resultate

Der Bequemlichkeit halber hatten wir eine Reihe von Annahmen über die Lage unserer Liniensegmente gemacht: Zum Beispiel sollten die X-Koordinaten der Endpunkte paarweise verschieden sein. Insbesondere waren damit senkrechte Segmente ausgeschlossen.

Praxistaugliche Algorithmen müssen ohne solche Einschränkungen auskommen. Im Prinzip gibt es zwei Möglichkeiten, um dieses Ziel zu erreichen: Man kann die Algorithmen so erweitern, daß sie auch mit „degenerierten“ Inputs fertig werden, oder man kann den Input so vorbehandeln, daß die Degenerationserscheinungen verschwinden. Wir wollen jetzt exemplarisch mögliche Maßnahmen vorstellen, um die Einschränkungen beim Liniensegment-Schnittproblem zu beseitigen.

Beseitigung der vereinfachenden Annahmen

Daß alle Endpunkte unterschiedliche X-Koordinaten haben, läßt sich zum Beispiel durch *Wahl eines geeigneten Koordinatensystems* erreichen. Wenn wir beim Sortieren nach X-Koordinaten vor Beginn des *sweep* feststellen, daß es Punkte mit gleichen X-Koordinaten gibt, so sortieren wir diese Gruppen von Punkten nach ihren Y-Koordinaten; das geht in demselben Sortiervorgang.[11] Als Ergebnis erhalten wir eine Folge vertikaler Punktgruppen, wie in Abbildung 2.12 gezeigt. Nun verbinden wir jeweils den obersten Punkt einer Gruppe mit dem untersten Punkt der rechten Nachbargruppe.

Endpunkte mit gleichen X-Koordinaten

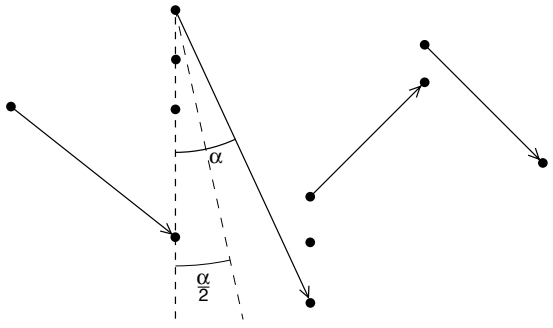


Abb. 2.12 Bestimmung eines neuen Koordinatensystems.

Für jedes dieser Liniensegmente bestimmen wir den Winkel mit der negativen Y-Achse. Sei α das Minimum dieser Winkel. Dann rotieren wir das alte XY-Koordinatenkreuz um den Winkel $\frac{\alpha}{2}$ entgegen dem Uhrzeigersinn und erhalten ein neues $X'Y'$-Koordinatensystem.

Übungsaufgabe 2.10

(i) Man beweise, daß nun alle Endpunkte paarweise verschiedene X'-Koordinaten haben.

(ii) Ist ein neuer Sortiervorgang nach X'-Koordinaten notwendig, bevor der *sweep* beginnen kann?

[11] Wir können von vornherein lexikographisch sortieren.

Schnittpunkte mit gleichen X-Koordinaten

Dieser Wechsel des Koordinatensystems verursacht nur $O(n)$ zusätzlichen Zeitaufwand. Es kann aber immer noch zu „gleichzeitigen“ Ereignissen kommen, wenn Schnittpunkte entdeckt werden, deren X-Koordinaten untereinander gleich sind oder mit denen von Endpunkten übereinstimmen. Wir vereinbaren, daß beim Auftreten von Ereignissen mit gleichen X-Koordinaten in der Ereignisstruktur erst die linken Endpunkte, dann die Schnittpunkte und danach die rechten Endpunkte an die Reihe kommen.

tiebreak in der Ereignisstruktur

Innerhalb dieser Gruppen werden die Ereignisse nach aufsteigenden Y-Koordinaten sortiert. Fallen mehrere linke Endpunkte zusammen, zählt die Ordnung der anhängenden Segmente; dasselbe gilt für mehrfache rechte Endpunkte.

vielfache Schnittpunkte

Etwas schwieriger ist der Fall von mehrfachen Schnittpunkten. Wir hatten ja bisher vorausgesetzt, daß sich höchstens zwei Liniensegmente in einem Punkt echt schneiden. Jetzt wollen wir den Sweep-Algorithmus so erweitern, daß er auch mit solchen Punkten zurechtkommt wie dem in Abbildung 2.13 gezeigten Punkt p, bei dem sich m Segmente schneiden.

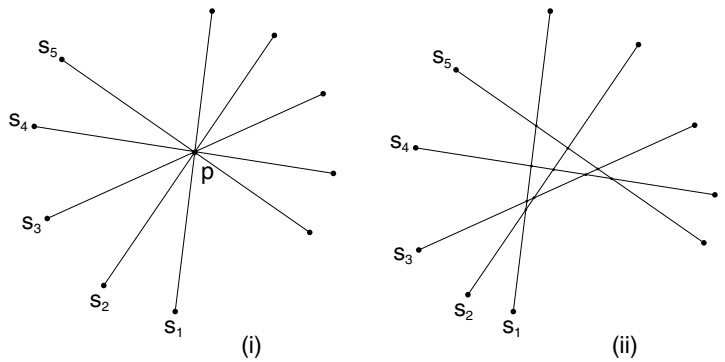


Abb. 2.13 Ein mehrfacher Schnittpunkt kann durch viele einfache „ersetzt“ werden.

Wir könnten so tun, als befänden sich die Liniensegmente in der in Abbildung 2.13 (ii) dargestellten Konfiguration (ohne aber an den Segmenten irgendwelche Änderungen vorzunehmen!). Das Sweep-Verfahren würde die einfachen Schnittpunkte in lexikographischer Reihenfolge $(s_1, s_2), (s_1, s_3), \ldots, (s_4, s_5)$ abarbeiten und dabei schrittweise die invertierte Anordnung der Segmente berechnen.

Entzerrung

Die folgende Übungsaufgabe zeigt, daß solch eine „Entzerrung“ immer möglich ist. Es wäre aber ineffizient, den mehrfachen Schnittpunkt p wie $\Theta(m^2)$ viele einfache Schnittpunkte zu behandeln, weil dabei entsprechend viele Zugriffe auf die *SSS* auszuführen wären.

Übungsaufgabe 2.11 Man beweise, daß sich jede Menge von m Liniensegmenten, die einen gemeinsamen echten Schnittpunkt besitzen, durch Translation ihrer Elemente in ein Arrangement[12] überführen läßt, in dem folgende Eigenschaft erfüllt ist: Jedes Segment s_i schneidet die übrigen in der Reihenfolge

$$s_1, s_2, \ldots, s_{i-1}, s_{i+1}, \ldots, s_m.$$

Dabei ist $s_1, s_2, \ldots, s_m$ die Ordnung der Segmente links vom Schnittpunkt.

Statt dessen warten wir mit der Bearbeitung von Schnittereignissen, bis die *sweep line* den Punkt p erreicht. Bis dahin verfahren wir wie bisher und testen jedes neu entdeckte Liniensegment auf Schnitt mit seinen beiden Nachbarn. Die dabei entstehenden Schnittereignisse ordnen wir in der Ereignisstruktur lexikographisch an. Außerdem wenden wir die Ersparnisregel an, derzufolge nur die Schnittpunkte zwischen direkt benachbarten Liniensegmenten gespeichert werden. Unabhängig von der Reihenfolge, in der die linken Endpunkte entdeckt werden, enthält dann die Ereignisstruktur im Beispiel von Abbildung 2.13 kurz vor p die Folge $(s_1, s_2), (s_2, s_3), (s_3, s_4), (s_4, s_5)$.

mehrfache Schnitte *en bloc* behandeln

Wir können daran erkennen, daß es sich um einen mehrfachen Schnittpunkt handelt, ihn berichten und dann *en bloc* die Reihenfolge der beteiligten m Liniensegmente in der Sweep-Status-Struktur invertieren. Hierfür wird nur $O(m)$ Zeit benötigt.

In Schorn [126] werden auch die numerischen Probleme diskutiert, die bei diesem Sweep-Algorithmus auftreten. Burnikel et al. [23] haben gezeigt, wie man sogar das zeitoptimale Verfahren so implementieren kann, daß alle möglichen degenerierten Situationen korrekt behandelt werden.

Im folgenden werden wir einige Anwendungen und Varianten des Sweep-Verfahrens besprechen.

sweep circle?

Wer sagt eigentlich, daß wir für den *sweep* unbedingt eine Gerade verwenden müssen? Könnte man nicht ebensogut einen Kreis von einem Punkt aus expandieren lassen und damit die Ebene ausfegen?

Angenommen, wir wollten mit diesem Ansatz alle Schnittpunkte berichten. Wir wählen einen Punkt z als Zentrum, der auf keinem Liniensegment liegt, und lassen von dort aus einen Kreis expandieren. Zuvor werden alle Endpunkte nach ihrem Abstand zu z sortiert. Alle Segmente, die vom Kreis geschnitten werden,

[12] Unter einem *Arrangement* oder einer *Konfiguration* geometrischer Objekte verstehen wir einfach eine Anordnung der Objekte.

sind aktiv. Wir merken uns ihre Reihenfolge entlang der Kreislinie. Alle Segmente, die ganz im Innern des Kreises liegen, sind tot, die draußen liegenden sind schlafend; siehe Abbildung 2.14.

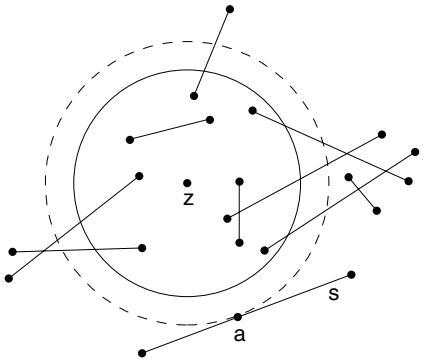


Abb. 2.14 Was passiert, wenn der *sweep circle* das Segment s erreicht?

Auf den ersten Blick könnte man meinen, dieses Verfahren funktioniere genausogut wie der *sweep* mit einer Geraden. Es tritt hier aber ein Phänomen auf, das eine Komplikation mit sich bringt: Ein Liniensegment kann aktiv werden, bevor der Kreis einen seiner Endpunkte erreicht hat! Dies wird z. B. für Segment s in Abbildung 2.14 eintreten. Die beiden Teile von s links und rechts vom Auftreffpunkt a sind wie zwei neue Segmente zu behandeln, die zufällig in a ihren gemeinsamen Anfangspunkt haben. Daher genügt es nicht, anfangs nur die Endpunkte der Liniensegmente, nach ihrem Abstand zu z sortiert, in die Ereigniswarteschlange einzufügen. Man muß außerdem für jedes Segment den ersten Auftreffpunkt des *sweep circle* bestimmen. Wenn es kein Endpunkt ist, gibt auch er zu einem Ereignis Anlaß.

2.3.3 Die untere Kontur – das Minimum von Funktionen

Minimum

Gegeben seien n reellwertige Funktionen f_i, die auf einem gemeinsamen Intervall I definiert sind. Zu bestimmen ist ihr *Minimum*

$$f(x) := \min_{1 \le i \le n} f_i(x).$$

untere Kontur

Offenbar besteht der Graph des Minimums aus denjenigen Teilen der Funktionsgraphen, die für einen bei $y = -\infty$ stehenden Beobachter sichtbar wären; siehe Abbildung 2.15. Man nennt ihn deshalb auch die *untere Kontur* K_u des Arrangements der

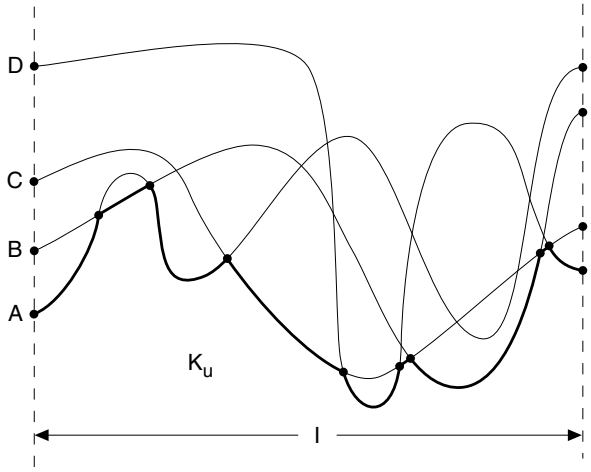


Abb. 2.15 Die untere Kontur K_u eines Arrangements von vier Funktionsgraphen über dem Intervall I.

Graphen. Im Englischen ist die Bezeichnung *lower envelope* gebräuchlich.

Ein Spezialfall liegt vor, wenn alle Funktionen f_i linear sind. Ihre Graphen sind dann Liniensegmente; siehe Abbildung 2.16. Offenbar können wir zur Berechnung von K_u unseren Sweep-Algorithmus aus Abschnitt 2.3.2 anwenden. Er kann die untere Kontur leicht aufbauen, indem er diejenigen Stücke der Liniensegmente verkettet, die bezüglich der Ordnung längs der *sweep line* gerade ganz unten sind.

Können wir die Sweep-Technik auch auf Graphen nichtlinearer Funktionen $f_i(x)$ anwenden? Als Wege betrachtet, haben Funktionsgraphen eine nützliche Eigenschaft: Sie sind *monoton* in bezug auf die X-Achse, d. h. jede senkrechte Gerade schneidet den Weg in höchstens einem Punkt. Diese Eigenschaft erlaubt es, vom *ersten gemeinsamen Schnittpunkt* zweier auf der *sweep line* benachbarter Wege zu reden. Abbildung 2.17 zeigt, warum dies bei beliebigen Wegen heikel ist.

monotoner Weg

Wir setzen voraus, daß Funktionsauswertung und Schnittpunktbestimmung in Zeit $O(1)$ ausführbar sind, wie bei Liniensegmenten. Auch zur Speicherung der Beschreibung einer Funktion f_i soll $O(1)$ Speicherplatz genügen.

Dann können wir zunächst folgende Verallgemeinerung von Theorem 2.10 formulieren:

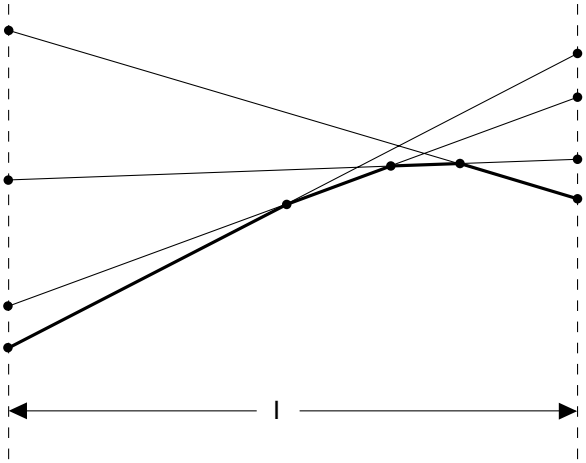


Abb. 2.16 Die untere Kontur K_u eines Arrangements von Liniensegmenten, die ein gemeinsames Intervall I überdecken.

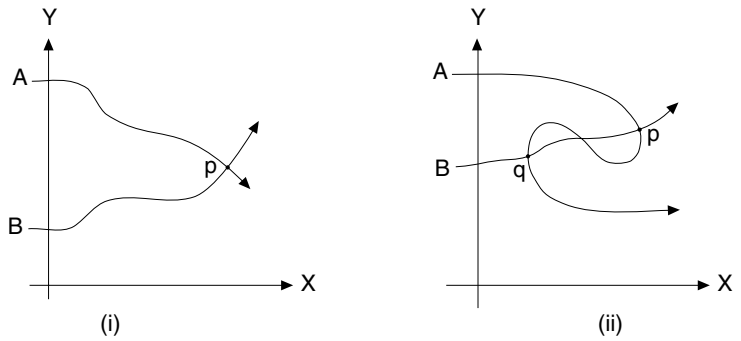


Abb. 2.17 (i) Für X-monotone Wege ist der erste gemeinsame Schnittpunkt wohldefiniert, falls es ihn gibt. (ii) Bei beliebigen Wegen ist das nicht so: p ist der erste gemeinsame Schnittpunkt von A und B auf A, aber q ist der erste gemeinsame Schnittpunkt auf B.

Theorem 2.11 *Mit dem Sweep-Verfahren kann man die k echten Schnittpunkte von n verschiedenen X-monotonen Wegen in Zeit $O((n+k)\log n)$ und Speicherplatz $O(n)$ bestimmen.*

Nebenbei läßt sich die untere Kontur der n Wege bestimmen. Aber ist dieses Vorgehen effizient? Schließlich hängt seine Laufzeit von der Gesamtzahl k *aller* Schnittpunkte ab – auch der von unten nicht sichtbaren! –, während uns eigentlich nur die untere Kontur interessiert.

Angenommen, je zwei Wege schneiden sich in höchstens s Punkten. Diese Annahme ist zum Beispiel erfüllt, wenn es sich um Graphen von Polynomen vom Grad $\leq s$ handelt. Dann ergibt sich

$$k \leq s\frac{n(n-1)}{2}.$$

Der Standard-Sweep-Algorithmus hat daher eine Laufzeit in $O(sn^2 \log n)$. Im Beispiel von Abbildung 2.15 wird für $s = 3$ und $n = 4$ die Maximalzahl $k = 18$ erreicht, aber nur 8 Schnittpunkte sind von unten sichtbar. Zur Berechnung der unteren Kontur würden wir uns deshalb ein Output-sensitives Verfahren wünschen.

untere Kontur effizienter berechnen

Der folgende, sehr einfache Algorithmus kommt diesem Ziel recht nahe, wie wir sehen werden. Er stellt eine Mischung aus *divide and conquer* und *sweep* dar. Im *divide*-Schritt wird die Menge der Wege in zwei gleich große Teilmengen zerlegt, ohne dabei auf geometrische Gegebenheiten Rücksicht zu nehmen. Dann werden rekursiv die unteren Konturen K_u^1 und K_u^2 der beiden Teilmengen berechnet. Im *conquer*-Schritt müssen sie nun zur unteren Kontur der Gesamtmenge zusammengesetzt werden. Abbildung 2.18 zeigt, welche Situation sich im Beispiel von Abbildung 2.15 ergibt, nachdem wir die untere Kontur K_u^1 der Wege A, D und die untere Kontur K_u^2 der Wege B, C berechnet haben.

Zum Zusammensetzen der beiden unteren Konturen führen wir einen *sweep* von links nach rechts durch. Dabei treten als Ereignisse auf:

(1) alle Ecken[13] von K_u^1 und K_u^2;

(2) alle Schnittpunkte von K_u^1 mit K_u^2.

Unsere Ereignisstruktur enthält zu jeder Zeit höchstens drei Einträge: die rechten Ecken der aktuellen Kanten von K_u^1 und K_u^2 und den ersten gemeinsamen Schnittpunkt der beiden aktuellen Kanten, falls ein solcher existiert.

Die Sweep-Status-Struktur ist noch einfacher: Sie enthält die beiden aktuellen Kanten in der jeweiligen Reihenfolge.

Mit diesem einfachen *sweep* läßt sich die gemeinsame Kontur in Zeit proportional zur Anzahl der Ereignisse berechnen; der logarithmische Faktor entfällt hier, weil wir vorher nicht mehr zu sortieren brauchen und die Strukturen *ES* und *SSS* nur konstante Größe haben.

Um die Anzahl der Ereignisse abschätzen zu können, definieren

[13] Mit „Ecken“ meinen wir die Punkte, an denen zwei Stücke von verschiedenen Wegen zusammenstoßen. Die Wegstücke werden entsprechend auch „Kanten“ genannt, in Analogie zu polygonalen Ketten.

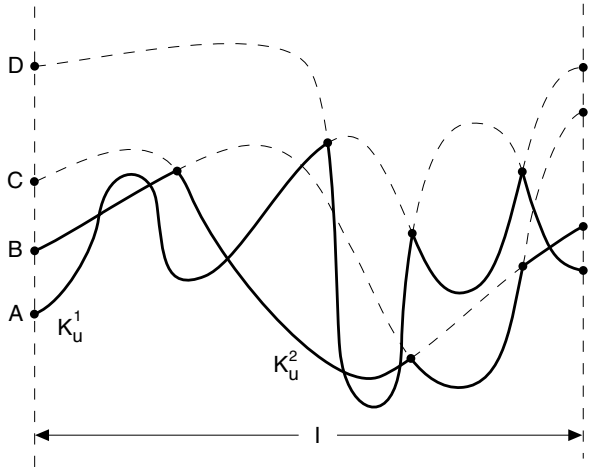


Abb. 2.18 Die unteren Konturen der Wege A, D und B, C aus Abbildung 2.15.

untere Kontur hat $\lambda_s(n)$ Kanten

wir als $\lambda_s(n)$ die maximale Kantenzahl der unteren Kontur eines Arrangements von n verschiedenen X-monotonen Wegen über einem gemeinsamen Intervall, von denen je 2 höchstens s Schnittpunkte haben. Klar ist, daß $\lambda_s(n)$ endlich ist und mit n monoton wächst.

Dann ist offenbar die Anzahl der Ereignisse vom Typ 1 beim Zusammensetzen der beiden unteren Konturen beschränkt durch

$$2\lambda_s\left(\left\lceil\frac{n}{2}\right\rceil\right) \leq 2\lambda_s(n),$$

denn K_u^1, K_u^2 sind ja untere Konturen von $\lceil n/2\rceil$ vielen Wegen.

conquer in Zeit $C\lambda_s(n)$

Jedes Ereignis vom Typ 2 gibt Anlaß zu einer Ecke der gemeinsamen Kontur K_u. Davon gibt es höchstens $\lambda_s(n)$ viele. Die Zeitkomplexität $T(n)$ unseres *divide and conquer*-Verfahrens genügt also der Rekursion

$$\begin{aligned} T(2) &= C \\ T(n) &\leq 2T(\frac{n}{2}) + C\lambda_s(n) \end{aligned}$$

für eine geeignete Konstante C. Um die Analyse zu Ende zu bringen, brauchen wir folgendes Lemma; den Beweis stellen wir hinter den Beweis von Theorem 2.14 zurück.

Lemma 2.12 *Für alle* $s, n \geq 1$ *ist* $2\lambda_s(n) \leq \lambda_s(2n)$.

Damit können wir der Rekursion genauso zu Leibe rücken, wie wir das von anderen Algorithmen kennen: Wir stellen uns

den Baum der Rekursionsaufrufe für eine Zweierpotenz n vor und beschriften jeden Knoten mit den Kosten des *divide*- und des *conquer*-Schritts, die an der Stelle entstehen. Die Wurzel trägt Kosten in Höhe von $C\lambda_s(n)$ für das Zusammensetzen von zwei Konturen, die aus je $n/2$ vielen Wegen bestehen. In jedem ihrer beiden Nachfolgerknoten entstehen Kosten von $C\lambda_s(n/2)$. Wegen Lemma 2.12 ergibt das Gesamtkosten in Höhe von

Rekursionsbaum

$$2C\lambda_s(\frac{n}{2}) \leq C\lambda_s(n)$$

auf der Ebene der Knoten mit Abstand 1 zur Wurzel. Per Induktion sieht man leicht, daß dieselben Kosten auf *jeder* Ebene im Baum entstehen, und weil es $\log_2(n)$ viele Ebenen gibt, erhalten wir folgendes Resultat:

Theorem 2.13 *Die untere Kontur von n verschiedenen X-monotonen Wegen über einem gemeinsamen Intervall, von denen sich je zwei in höchstens s Punkten schneiden, läßt sich in Zeit $O(\lambda_s(n)\log n)$ berechnen.*

Unser neuer Algorithmus ist also schon viel Output-sensitiver als die Berechnung aller Schnittpunkte: Von den $\Omega(sn^2)$ vielen möglichen Schnittpunkten gehen nur so viele in die Zeitkomplexität ein, wie schlimmstenfalls in der unteren Kontur auftreten könnten.[14]

Spätestens hier stellt sich die Frage nach der Natur der Funktion $\lambda_s(n)$. Haben wir gegenüber sn^2 wirklich eine Verbesserung erzielt?

wie groß ist $\lambda_s(n)$?

Überraschenderweise läßt sich $\lambda_s(n)$ auch rein kombinatorisch definieren, ohne dabei geometrische Begriffe wie untere Konturen zu verwenden. Gegeben seien n Buchstaben $A, B, C, \ldots$ Wir betrachten Wörter über dem Alphabet $\{A, B, C, \ldots\}$ und interessieren uns dafür, wie oft zwei Buchstaben einander abwechseln, auch auf nicht konsekutiven Positionen.

Ein Wort heißt *Davenport-Schinzel-Sequenz* der Ordnung s, wenn darin kein Buchstabe mehrfach direkt hintereinander vorkommt, und wenn je zwei Buchstaben höchstens s mal abwechselnd auftreten.

Davenport-Schinzel-Sequenz

Danach ist $ABBA$ keine Davenport-Schinzel-Sequenz (wegen BB), $ABRAKADABRA$ hat *nicht* die Ordnung 3 (wegen des *vier*fachen Wechsels in $AB.A\ldots.B.A$), wohl aber die Ordnung 4 und damit automatisch jede höhere Ordnung.

[14] Ideale Output-Sensitivität würde bedeuten: Es gehen nur so viele Schnittpunkte in die Laufzeit ein, wie im konkreten Eingabearrangement tatsächlich auf der unteren Kontur vorhanden sind.

Man kann sich vorstellen, daß solche Worte nicht beliebig lang sein können. Diese Vorstellung wird durch folgenden Satz erhärtet.

Theorem 2.14 *Die maximale Länge einer Davenport-Schinzel-Sequenz der Ordnung s über n Buchstaben beträgt $\lambda_s(n)$.*

Beweis. Wir zeigen zuerst, daß es zu jeder unteren Kontur eine gleich lange Davenport-Schinzel-Sequenz gibt. Sei also K_u untere Kontur eines Arrangements von n X-monotonen Wegen $A, B, C, \ldots$ über einem gemeinsamen Intervall, von denen sich je zwei höchstens s mal schneiden. Wir beschriften jede „Kante“ von K_u mit dem Index des Weges, aus dem sie stammt. In Abbildung 2.15 ergibt sich dabei das Wort $ABACDCBCD$. Weil direkt aufeinander folgende Kanten der Kontur aus verschiedenen Wegen kommen, tritt nie derselbe Index zweimal hintereinander auf. Außerdem können zwei Indizes A, B einander nicht mehr als s-mal abwechseln! Denn bei jedem Wechsel $A \ldots B$ muß der Weg B durch den Weg A nach unten stoßen, um Teil der unteren Kontur werden zu können. Es sind aber nur s Schnittpunkte erlaubt.

Umgekehrt sei eine Davenport-Schinzel-Sequenz über n Buchstaben gegeben, die die Ordnung s hat. Wir konstruieren ein Arrangement von Wegen, das diese Sequenz als Indexfolge seiner unteren Kontur hat. Sei A der erste Buchstabe der Sequenz, B der erste von A verschiedene, und so fort. Das Konstruktionsverfahren ist sehr einfach: Am linken Intervallende beginnen die Wege in der Reihenfolge $A, B, C, \ldots$ von unten nach oben; siehe Abbildung 2.19. Im Prinzip laufen alle Wege in konstanter Höhe nach rechts. Nur wenn der Index eines Weges in der Sequenz gerade an der Reihe ist, darf dieser Weg durch die anderen hindurch ganz nach unten stoßen und dort weiter nach rechts laufen.

Abbildung 2.19 zeigt, welches Arrangement sich nach diesem Konstruktionsverfahren für die Sequenz $ABACACBC$ ergibt.

Daß die entstehenden Wege X-monoton sind und die vorgegebene Sequenz gerade die Indexfolge längs ihrer unteren Kontur ist, folgt aus der Konstruktion. Wie sieht es mit der Anzahl der Schnittpunkte aus? Betrachten wir zwei beliebige Indizes B und C.

Weil B zuerst in der Sequenz vorkommt, verläuft am Anfang der Weg B unterhalb des Weges C. Der erste gemeinsame Schnittpunkt ergibt sich erst, wenn C nach unten stößt, um Teil der unteren Kontur zu werden. Der nächste Schnittpunkt ergibt sich, wenn wieder B nach unten stößt, und so fort. Dies ist jedesmal durch einen Wechsel der Indizes $\ldots B \ldots C \ldots B \ldots$ in der Sequenz verursacht. Weil es höchstens s Wechsel zwischen B und C gibt, können die beiden Wege sich höchstens s-mal schneiden. □

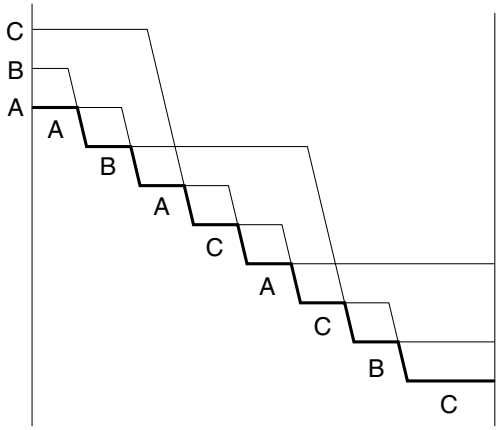


Abb. 2.19 Konstruktion eines Arrangements von Wegen mit vorgegebener unterer Kontur.

Beweis von Lemma 2.12

Die kombinatorische Aussage von Theorem 2.14 macht es leichter, Eigenschaften von $\lambda_s(n)$ herzuleiten. Zum Beispiel ist jetzt offensichtlich, warum Lemma 2.12 gilt: Wenn wir mit n Buchstaben eine Davenport-Schinzel-Sequenz der Ordnung s und der Länge l bilden können, dann läßt sich mit $2n$ Buchstaben leicht eine doppelt so lange Sequenz der Ordnung s konstruieren. Wir schreiben die Sequenz zweimal hintereinander, verwenden aber beim zweiten Mal neue Buchstaben.

Auch einige andere Eigenschaften von $\lambda_s(n)$ sind nun leicht zu zeigen.

Übungsaufgabe 2.12

(i) Man zeige, daß $\lambda_1(n) = n$ gilt.

(ii) Man beweise $\lambda_2(n) = 2n - 1$.

(iii) Man zeige, daß $\lambda_s(n) \leq \frac{s(n-1)n}{2} + 1$ gilt.

$\lambda_s(n)$ ist fast linear in n

Man könnte durch die Aussagen (i) und (ii) in Übungsaufgabe 2.12 zu der Vermutung gelangen, daß für jede feste Ordnung s die Funktion $\lambda_s(n)$ linear in n ist. Diese Vermutung ist zwar falsch, kommt der Wahrheit aber ziemlich nahe: Man kann zeigen, daß für festes s

$$\lambda_s(n) \in O(n \log^* n)$$

gilt. Dabei bezeichnet $\log^* n$ die kleinste Zahl m, für die die m-fache Iteration des Logarithmus

$$\underbrace{\log_2\big(\log_2(\ldots(\log_2(n))\ldots)\big)}_{m\text{-mal}}$$

einen Wert ≤ 1 ergibt, oder äquivalent, für die der Turm von m Zweierpotenzen

$$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \Big\} m\text{-mal}$$

größer gleich n wird. Die Funktion $\log^* n$ kann für „praktische“ Werte von n als Konstante angesehen werden. So ist zum Beispiel $\log^* n \leq 5$ für alle $n \leq 10^{20000}$. Die Aussage von Übungsaufgabe 2.12 (iii) ist also *viel* zu grob!

Wer sich für dieses Gebiet interessiert, sei auf das Buch von Sharir und Agarwal [136] verwiesen, in dem es ausschließlich um Davenport-Schinzel-Sequenzen und um ihre Bedeutung in der Geometrie geht.

Schauen wir uns noch einmal ein Arrangement von Liniensegmenten über einem Intervall an, wie es in Abbildung 2.16 dargestellt ist. Weil zwei verschiedene Geraden sich höchstens einmal schneiden, ist $s = 1$; nach Übungsaufgabe 2.12 (i) und Theorem 2.13 folgt:

Korollar 2.15 *Die untere Kontur von n Liniensegmenten über einem gemeinsamen Intervall läßt sich in Zeit $O(n \log n)$ und linearem Speicherplatz berechnen.*

untere Kontur beliebiger Liniensegmente

Man könnte glauben, daß Davenport-Schinzel-Sequenzen höherer Ordnung bei Liniensegmenten nicht auftreten. Das stimmt aber nicht. Sobald die Endpunkte nicht mehr dieselben X-Koordinaten haben, wird die untere Kontur komplizierter; siehe Abbildung 2.20. Wir nehmen im folgenden an, daß keines der Liniensegmente senkrecht ist.

Korollar 2.16 *Die untere Kontur von n Liniensegmenten in beliebiger Lage enthält $O(\lambda_3(n))$ viele Segmente. Sie läßt sich in Zeit $O(\lambda_3(n) \log n)$ berechnen.*

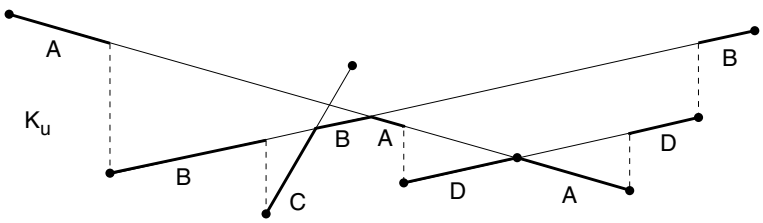


Abb. 2.20 Die untere Kontur K_u eines Arrangements von beliebigen Liniensegmenten.

Beweis. Wir können uns auf den Fall von X-monotonen Wegen über einem gemeinsamen Intervall zurückziehen. Dazu fügen wir am linken Endpunkt eines jeden Segments ein (langes) Liniensegment negativer Steigung an. Diese neuen Segmente sind parallel und haben ihre linken Endpunkte auf einer gemeinsamen Senkrechten. Entsprechend werden die vorhandenen Liniensegmente nach rechts durch parallele Segmente positiver Steigung verlängert; siehe Abbildung 2.21. Bei passender Wahl der Steigungen hat das neue Arrangement in der unteren Kontur dieselbe Indexfolge wie das alte. Wegen der Parallelität der äußeren Segmente können sich zwei Wege höchstens dreimal schneiden, wie zum Beispiel A und B in Abbildung 2.21. Die untere Kontur hat deshalb die Komplexität $O(\lambda_3(n))$. Aus Theorem 2.13 folgt die Abschätzung für die Rechenzeit. □

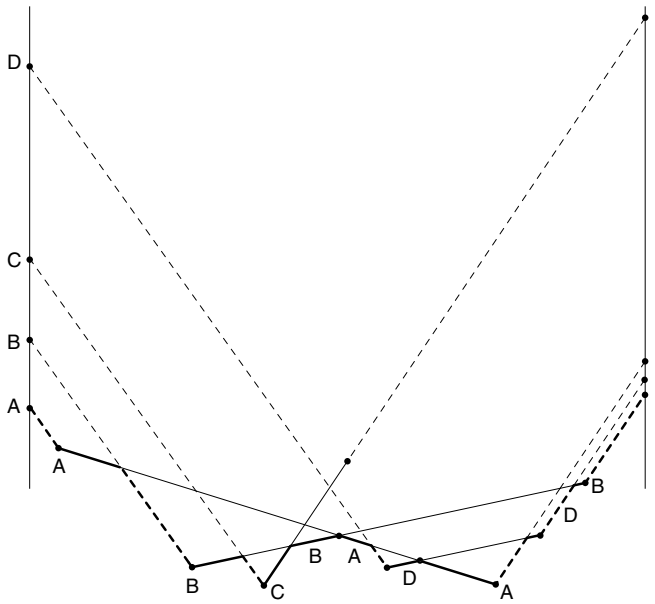


Abb. 2.21 Reduktion auf den Fall X-monotoner Wege über demselben Intervall.

In [136] wird gezeigt, daß es Arrangements von n Liniensegmenten gibt, deren untere Kontur tatsächlich die Komplexität $\Omega(\lambda_3(n))$ besitzt. Außerdem kennt man die Funktion λ_3 recht gut; es ist nämlich

$$\lambda_3(n) \in \Theta(n\alpha(n)),$$

wobei $\alpha(n)$ die Inverse der *Ackermann-Funktion* bedeutet, jener extrem schnell wachsenden Funktion, mit deren Hilfe in der Rekur- Ackermann-Funktion

sionstheorie gezeigt wird, daß die primitiv rekursiven Funktionen eine echte Teilmenge der μ-rekursiven Funktionen bilden.

Zur Erinnerung: Man definiert zuerst rekursiv die Funktion A durch

$$\begin{array}{lcll} A(1,n) & = & 2n & \text{für } n \geq 1 \\ A(k,1) & = & A(k-1,1) & \text{für } k \geq 2 \\ A(k,n) & = & A(k-1,A(k,n-1)) & \text{für } k \geq 2, n \geq 2. \end{array}$$

Dann ist

$$\begin{array}{rcl} A(2,n) & = & 2^n \\ A(3,n) & = & 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \Big\} n\text{-mal} \end{array}$$

Durch Diagonalisierung erhält man die Ackermann-Funktion

$$a(n) = A(n,n),$$

und ihre Inverse $\alpha(m)$ ist definiert durch

$$\alpha(m) = \min\{n; a(n) \geq m\}.$$

Entsprechend langsam wächst $\alpha(n)$, so daß $\lambda_3(n)$ fast linear ist. Da ist es eher von theoretischem Interesse, daß man die untere Kontur von n Liniensegmenten in beliebiger Lage auch in Zeit $O(n \log n)$ berechnen kann, wobei gegenüber Korollar 2.16 der Faktor $\alpha(n)$ eingespart wird [136]. Trotzdem ist es schon bemerkenswert, daß die von unten sichtbare Kontur von n Liniensegmenten eine überlineare strukturelle Komplexität haben kann und daß die Funktion $\alpha(n)$ an dieser Stelle „in der Natur vorkommt“.

Übungsaufgabe 2.13 Bisher haben wir stets angenommen, daß der Beobachter sich das Arrangement der Liniensegmente von $y = -\infty$ aus anschaut. Können wir unsere Algorithmen zur Berechnung der sichtbaren Kontur an die Situation anpassen, wo der Beobachter mitten in der Ebene steht?

2.3.4 Der Durchschnitt von zwei Polygonen

Zu den Grundaufgaben der algorithmischen Geometrie gehört es, Durchschnitt und Vereinigung von geometrischen Objekten zu berechnen.

Abbildung 2.22 zeigt den Durchschnitt von zwei Polygonen P und Q. Er ist nicht zusammenhängend, sondern besteht aus mehreren Polygonen D_i. Wir werden uns in diesem Abschnitt zunächst ein Verfahren überlegen, mit dem man testen kann, ob der Durchschnitt des Inneren von zwei Polygonen nicht leer ist. Dann geben wir einen Algorithmus zur Berechnung des Durchschnitts an.

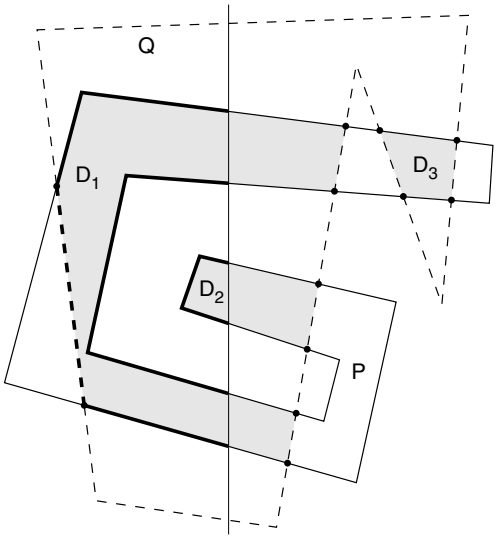


Abb. 2.22 Der Durchschnitt von zwei einfachen Polygonen.

Annahmen

Der Einfachheit halber setzen wir wieder voraus, daß es keine senkrechten Kanten gibt und daß keine Ecke des einen Polygons auf einer Kante des anderen liegt.[15]

Durchschnitt leer?

Offenbar haben P und Q genau dann einen nichtleeren Durchschnitt, wenn es eine Kante von P und eine Kante von Q gibt, die sich echt schneiden, oder wenn ein Polygon das andere ganz enthält.

Die erste Bedingung läßt sich mit dem einfachen Sweep-Verfahren aus Abschnitt 2.3.2 testen: Wir prüfen, ob es in der Menge aller Kanten von P und Q einen echten Schnittpunkt gibt.[16] Wenn dieser Test negativ ausgeht, bleibt noch festzustellen, ob P in Q enthalten ist (oder umgekehrt). Dazu genügt es, für einen *beliebigen* Eckpunkt p von P zu testen, ob er in Q liegt. Falls ja, liegt ein Schnitt vor. Wir verwenden folgende Methode: Man zieht eine Halbgerade von p z. B. in Richtung der positiven X-Achse. Dann testet man für jede Kante von Q, ob sie von der Halbgeraden geschnitten wird, und bestimmt die Gesamtzahl der geschnittenen Kanten. Ist sie ungerade, so liegt p in Q, andernfalls außerhalb. Der Aufwand ist linear in der Anzahl der Kanten von Q.

p in Q?

[15] Man kann den im folgenden beschriebenen Algorithmus aber so erweitern, daß er ohne diese Annahmen auskommt und selbst im Fall $P = Q$ funktioniert.

[16] Die Ecken der Polygone gelten nicht als echte Schnittpunkte der beiden angrenzenden Kanten.

Als Folgerung aus dieser Beobachtung und Theorem 2.8 erhalten wir:

Korollar 2.17 *Seien P und Q zwei Polygone mit insgesamt n Eckpunkten. Dann läßt sich in Zeit $O(n \log n)$ und linearem Speicherplatz feststellen, ob P und Q sich schneiden.*

Berechnung des Durchschnitts

Jetzt betrachten wir das Problem, den Durchschnitt von P und Q zu berechnen. Für jedes Polygon D_i in

$$P \cap Q = \bigcup_{i=1}^{r} D_i$$

soll dabei die Folge seiner Eckpunkte berichtet werden.

Intervalle auf der *sweep line*

Wir lassen eine *sweep line* von links nach rechts über die Polygone laufen. Sie wird durch P und Q in Intervalle zerlegt, deren Inneres zu P und zu Q, zu einem der beiden Polygone oder zu keinem von ihnen gehören kann. Uns interessieren nur die Intervalle in $P \cap Q$. Jedes von ihnen ist in einem Polygon D_i enthalten.

Die Endpunkte dieser Intervalle werden links von der *sweep line* durch Teile des Randes von $P \cap Q$, d. h. durch Kantenfolgen der Polygone D_i, miteinander verbunden. Sie sind in Abbildung 2.22 hervorgehoben. Dabei braucht eine Kantenfolge, die im oberen Punkt eines Intervalls startet, nicht an dessen unterem Punkt zu enden. Das ist zwar für den Durchschnitt der *sweep line* mit D_2 so, nicht aber für die beiden Intervalle von D_1.[17]

Inhalt der *SSS*

Wir verwalten in der Sweep-Status-Struktur

- das Verzeichnis der Kanten von P und Q, die gerade von der *sweep line* geschnitten werden,
- für jedes Intervall der *sweep line* die Information, ob es zu $P \cap Q$, $P \cap Q^C$, $P^C \cap Q$ oder $P^C \cap Q^C$ gehört,[18]
- die Kantenfolgen der D_i links von der *sweep line*, die die Endpunkte der Intervalle in $P \cap Q$ miteinander verbinden.

Die Kantenfolgen werden als verkettete Listen implementiert, und für jeden Endpunkt v eines Intervalls in $P \cap Q$ wird ein Verweis auf das Endstück der zugehörigen Kantenfolge verwaltet, auf dem v liegt.

Ereignisse

Ereignisse finden statt, wenn die *sweep line* auf Eckpunkte von P oder Q stößt oder wenn sie Schnittpunkte zwischen Kanten aus

[17] Allgemein gilt: Wenn man den oberen bzw. den unteren Punkt jeder solchen Kantenfolge auf der *sweep line* mit einer öffnenden bzw. schließenden Klammer markiert, entsteht ein korrekter Klammerausdruck.

[18] Dabei bezeichnet A^C das Komplement von A.

P und Q erreicht. Die Verwaltung der Ereignisse in der *ES* erfolgt wie bei der Bestimmung der Durchschnitte von Liniensegmenten in Abschnitt 2.3.2.

Die Verwaltung der *SSS* macht ein wenig mehr Mühe; schließlich müssen wir auch die Intervallinformationen und die Kantenfolgen aktualisieren. Es gibt mehrere Fälle. Bei Erreichen eines Eckpunkts v – sagen wir: von P – kommt es auf die Lage seiner beiden Kanten an: Zeigt eine nach links und die andere nach rechts, so ändert sich nichts an den Intervallen auf der *sweep line*, und alle aktiven Kantenfolgen werden verlängert. Wenn beide Kanten nach links zeigen, verschwindet ein Intervall; liegt v im Innern von Q, sind dort außerdem zwei Enden von Kantenfolgen miteinander zu verbinden. Wenn dagegen beide Kanten von v nach rechts weisen, entsteht ein neues Intervall und, falls v in Q liegt, auch eine neue Kantenfolge. Abbildung 2.23 zeigt drei typische Beispiele.

Eckpunkt

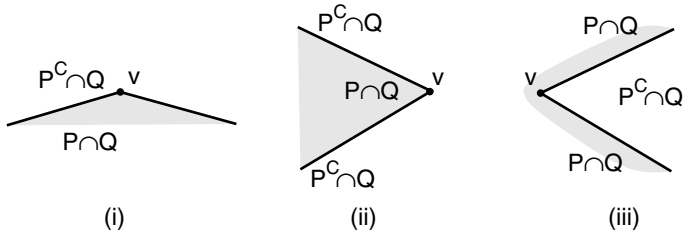


Abb. 2.23 Im Eckpunkt v kann eine aktive Kantenfolge von $P \cap Q$ verlängert werden (i), zwei Enden solcher Folgen können zusammenkommen (ii), oder eine neue Kantenfolge kann entstehen (iii).

Kreuzen sich in v zwei Kanten von P und Q, ändert sich nicht die Anzahl der Intervalle längs der *sweep line*, wohl aber ihre Zugehörigkeit zu den beiden Polygonen. Es kann sein, daß in v eine aktive Kantenfolge verlängert wird, daß zwei Enden von Kantenfolgen zusammenwachsen oder daß eine neue Kantenfolge entsteht; siehe Abbildung 2.24.

Schnittpunkt

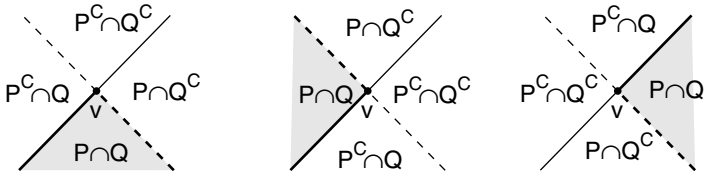


Abb. 2.24 Die möglichen Veränderungen bei der Bearbeitung eines Schnittpunkts.

Am Ende ist für jedes Polygon D_i seine geschlossene Kantenfolge bekannt. Sie kann in einem weiteren Durchlauf ausgegeben werden. Insgesamt ist dieses Verfahren nicht zeitaufwendiger als die Schnittbestimmung bei Liniensegmenten. Aus Theorem 2.10 folgt:

Theorem 2.18 *Der Durchschnitt von zwei einfachen Polygonen mit insgesamt n Kanten und k Kantenschnittpunkten kann in Zeit $O((n+k)\log n)$ und Speicherplatz $O(n)$ berechnet werden.*

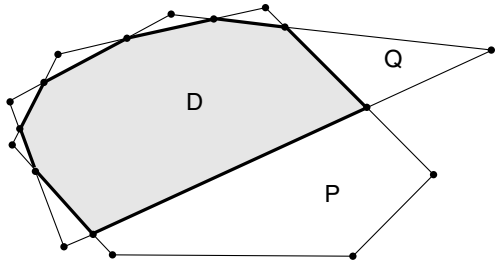


Abb. 2.25 Der Durchschnitt von zwei konvexen Polygonen mit n Ecken ist leer oder ein konvexes Polygon mit $O(n)$ Ecken.

einfacher: Durchschnitt konvexer Polygone

Ein interessanter Spezialfall tritt auf, wenn beide Polygone *konvex* sind. Dann ist auch ihr Durchschnitt konvex, kann also insbesondere nur aus einem einzelnen Polygon bestehen. Wenn P und Q maximal n Ecken haben, weist auch ihr Durchschnitt nur $O(n)$ viele Ecken auf, denn jede Kante von P kann den Rand von Q in höchstens zwei Punkten schneiden.

Übungsaufgabe 2.14 Gegeben seien zwei konvexe Polygone mit m bzw. n Ecken. Angenommen, ihr Durchschnitt ist nicht leer. Wie viele Ecken kann $P \cap Q$ höchstens haben? Man bestimme die genaue obere Schranke.

Wenn wir bei einem konvexen Polygon die obere Hälfte des Randes betrachten, also das Stück, das im Uhrzeigersinn von seiner am weitesten links gelegenen Ecke zur am weitesten rechts gelegenen Ecke läuft, so ist diese polygonale Kette X-monoton. Dasselbe gilt für die untere Hälfte des Randes.

Schnitt monotoner Ketten

Wir hatten schon im Abschnitt 2.3.3 bei der Diskussion des *sweep* im *divide and conquer*-Verfahren zur Berechnung der unteren Kontur gesehen, daß sich die Schnittpunkte zweier solcher monotonen Ketten mit einem Zeitaufwand berechnen läßt, der *linear* ist in der Gesamtzahl der Ecken und Schnittpunkte der beiden Ketten.

Wenn wir aber die Schnittpunkte der Randhälften von P und Q kennen, können wir daraus leicht in linearer Zeit den Rand von $P \cap Q$ rekonstruieren. Es folgt:

Theorem 2.19 *Der Durchschnitt von zwei konvexen Polygonen mit insgesamt n Ecken läßt sich in Zeit und Speicherplatz $O(n)$ berechnen, und das ist optimal.*

Finke und Hinrichs [56] haben kürzlich gezeigt, daß der Durchschnitt von zwei beliebigen Polygonen auch in Zeit $O(n + k)$ berechnet werden kann. Für dieses Ergebnis muß aber schweres Geschütz aufgefahren werden, auf das wir später bei der Diskussion der *Triangulation eines Polygons* in Abschnitt 4.2 zurückkommen werden.

2.4 Sweep im Raum

In den vorangegangenen Abschnitten haben wir an zahlreichen Beispielen gesehen, wie effizient und konzeptuell einfach das Sweep-Verfahren auf der Geraden und in der Ebene funktioniert. Auch in späteren Kapiteln wird uns der *sweep* in der Ebene wiederbegegnen.

Die Frage liegt nahe, ob sich diese nützliche Technik auf höhere Dimensionen verallgemeinern läßt. Wir sind zunächst bescheiden und betrachten erst einmal den dreidimensionalen Raum, den wir uns einigermaßen gut vorstellen können.

sweep im $\mathbb{R}^3$

Um den $\mathbb{R}^3$ zügig auszufegen, müssen wir schon eine Ebene nehmen, die *sweep plane*. Wir stellen uns vor, daß sie parallel zu den YZ-Koordinatenachsen ist und von links nach rechts, d. h. in Richtung steigender X-Koordinaten, durch den Raum geschoben wird.

2.4.1 Das dichteste Punktepaar im Raum

kleinster Abstand zwischen n Punkten im Raum

Wir diskutieren ein vertrautes Problem: die Bestimmung eines dichtesten Paars von n Punkten. Wie früher werden wir uns mit der Bestimmung des *kleinsten Abstands* zwischen den Punkten begnügen.

Eigentlich geht alles genauso wie in Abschnitt 2.3.1: Wir merken uns in *MinSoFar* den kleinsten Abstand aller Punkte links von der *sweep plane*. Die Sweep-Status-Struktur enthält alle Punkte in einem Streifen der Breite *MinSoFar* links von der *sweep plane*. Ein Ereignis findet immer dann statt, wenn ein neuer Punkt den Streifen betritt oder ein alter ihn verläßt; siehe Abbildung 2.26.

Unseren alten Algorithmus können wir wörtlich wiederverwenden! Nur die Funktion

$$MinDist(SSS,r,MinSoFar)$$

muß neu implementiert werden.

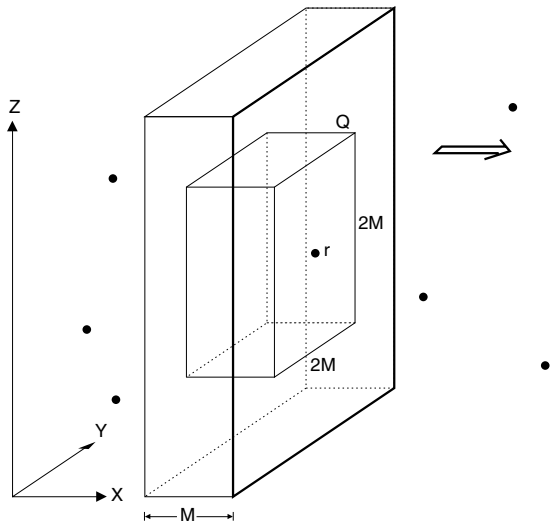


Abb. 2.26 Nur Punkte im Quader Q können zum neuen Punkt r einen Abstand $< M = MinSoFar$ haben.

Erinnern wir uns: Ein solcher Aufruf berechnet das Minimum vom alten Wert *MinSoFar* und dem kleinsten Abstand vom Punkt r, der soeben von der *sweep plane* getroffen wurde, zu den Punkten in der *SSS*.

Hier genügt es, alle Punkte im achsenparallelen Quader Q zu betrachten, der in Y- und Z-Richtung die Kantenlänge $2MinSoFar$ hat, in X-Richtung *MinSoFar* tief ist und den Punkt r in der Mitte seiner rechten Seite hat; siehe Abbildung 2.26. In Analogie zu Lemma 2.3 kann Q nur eine beschränkte Anzahl von Punkten enthalten.

Implementierung der *SSS*

Die Frage lautet: Wie implementieren wir die Sweep-Status-Struktur? Beim *sweep* in der Ebene haben uns bei der Frage nach den Punkten im Rechteck R in Abbildung 2.5 nur die *Projektionen* der Punkte im Streifen *auf die sweep line* interessiert. Deshalb konnten wir die X-Koordinaten der aktiven Punkte ignorieren und zur Speicherung der aktiven Punkte eine eindimensionale, nur nach den Y-Koordinaten sortierte Datenstruktur verwenden. Auf ihr waren – außer Einfügen und Entfernen von Punkten – *Bereichsanfragen* für Intervalle (der Länge $2MinSoFar$) auszuführen.

Entsprechend brauchen wir hier eine zweidimensionale Datenstruktur, in der Punkte im $\mathbb{R}^3$ nach ihren YZ-Koordinaten abgespeichert werden. Neben Einfügen und Entfernen müssen *Bereichsanfragen für achsenparallele Quadrate* (der Kantenlänge $2MinSoFar$) möglich sein.

rechteckige Bereichsanfragen

Geeignete Datenstrukturen werden wir im nächsten Kapitel kennenlernen. Angenommen, wir hätten schon eine solche Struktur, die es uns ermöglicht,

- Einfügen und Entfernen in Zeit $O\big(e(n)\big)$,
- rechteckige Bereichsanfragen in Zeit $O\big(b(n)\big)$

auszuführen, vorausgesetzt, die Größe der Antwort ist beschränkt, wie es bei uns ja der Fall ist. Dann könnten wir nach dem Sortieren der n Punkte entsprechend ihrer X-Koordinaten den kleinsten Abstand zwischen ihnen in Zeit

$$O\Big(n\big(e(n)+b(n)\big)\Big)$$

mit unserem alten Sweep-Verfahren bestimmen. Ein Vorteil gegenüber dem naiven $\Theta(n^2)$-Algorithmus, der jedes Punktepaar einzeln inspiziert, kann sich nur ergeben, falls $e(n)+b(n)$ sublinear ist.

Lösungen der Übungsaufgaben

Übungsaufgabe 2.1 Obwohl dieses Problem zweidimensional aussieht, läßt es sich auf den eindimensionalen Fall reduzieren. Denn auch auf der Sinuskurve können zwei Punkte nur dann ein dichtestes Paar bilden, wenn sie benachbart sind. Diese Eigenschaft hat die Sinuskurve mit der Geraden gemein.

Zum Beweis dieser Tatsache zeigen wir zunächst: Sind $p_1 = (x_1, \sin x_1)$ und $p_2 = (x_2, \sin x_2)$ zwei verschiedene Punkte auf der Sinuskurve, so schneidet ihr *Bisektor* (vgl. Abschnitt 1.2.3)

$$B(p_1, p_2) = \{\, q \in \mathbb{R}^2;\ |p_1 q| = |p_2 q| \,\}$$

die Sinuskurve genau einmal, und zwar zwischen p_1 und p_2; siehe Abbildung 2.27.

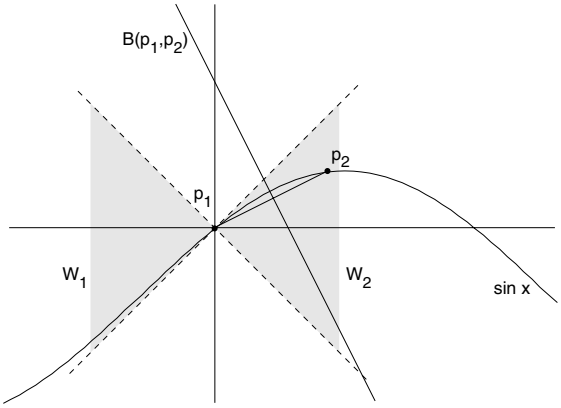


Abb. 2.27 Der Bisektor $B(p_1, p_2)$ von zwei Punkten auf der Sinuskurve schneidet sie genau einmal.

Die Steigung des Liniensegments $p_1 p_2$ hat den Betrag

$$\left|\frac{\sin x_2 - \sin x_1}{x_2 - x_1}\right| = |\sin' \xi| = |\cos \xi| \leq 1.$$

Solch ein ξ existiert nach dem Mittelwertsatz der Differentialrechnung. Weil diese Überlegung für p_1 und beliebige Kurvenpunkte p_2 gilt, ist die Sinuskurve in die in Abbildung 2.27 grau dargestellten Winkelbereiche W_1 und W_2 eingesperrt, die aus den beiden Nebendiagonalen durch p_1 gebildet werden.

Angenommen, p_2 liegt rechts von p_1. Weil p_2 den Bereich W_2 nicht verlassen kann, kann der Bisektor $B(p_1, p_2)$ den Bereich W_1

nicht schneiden! Ebensowenig kann er den rechten Winkelbereich an p_2 schneiden, der hier nicht eingezeichnet ist.

Alle Kurvenpunkte links von p_1 liegen also im Bereich W_1, der wiederum in der Halbebene links von $B(p_1, p_2)$ enthalten ist. Sie sind deshalb näher an p_1 als an p_2. Ebenso liegt jeder Punkt auf der Sinuskurve rechts von p_2 näher an p_2 als an p_1.

Das bedeutet: Ein dichtestes Paar von n Punkten $(x_i, \sin x_i)$ kann nur aus Punkten bestehen, deren X-Koordinaten benachbart sind in der sortierten Reihenfolge

$$x_{\pi(1)} \leq x_{\pi(2)} \leq \ldots \leq x_{\pi(n)}.$$

Wir sortieren also die Punkte p_i nach aufsteigenden X-Koordinaten (d. h. nach ihrer Reihenfolge auf der Sinuskurve) und betrachten in einem *sweep* alle Paare von konsekutiven Punkten. Dafür wird insgesamt $O(n \log n)$ Rechenzeit benötigt.

Übungsaufgabe 2.2

```
MaxSoFar := 0;
MaxEndingHere := 0;
while true do
    read(DailyVari);
    MaxEndingHere := max(0, MaxEndingHere + DailyVari);
    MaxSoFar := max(MaxSoFar,MaxEndingHere);
    write("Bis heute hätte man den maximalen Gewinn ",
        MaxSoFar, " erzielen können.")
```

Übungsaufgabe 2.3 Nach der Initialisierung ist zu jedem Zeitpunkt $MinSoFar > 0$, denn diese Variable hat stets den Abstand zweier verschiedener Punkte als Wert. Der Wert nimmt aber monoton ab.

Es folgt, daß immer $links \leq rechts$ gilt: Nur die Vergrößerung von *links* im **then**-Zweig ist kritisch. Sie wird aber nur ausgeführt, wenn vorher

$$P[links].x + MinSoFar \leq P[rechts].x$$

war, was wegen $MinSoFar > 0$ für $links = rechts$ unmöglich wäre; also galt vorher $links < rechts$. Außerdem ist leicht zu sehen, daß nach jeder Aktualisierung von *links* bzw. *rechts* diese Indizes zu einem Punkt mit minimaler X-Koordinate *in* der *SSS* gehören bzw. zu einem Punkt mit minimaler X-Koordinate, der *nicht* in der *SSS* enthalten ist.

Wir zeigen jetzt durch Induktion über die Anzahl der Durchläufe der **while**-Schleife, daß stets I_1 und I_2 erfüllt sind.

Beim ersten Durchlauf gilt I_1 aufgrund der Initialisierung. Falls jetzt der **else**-Zweig betreten wird, ist $rechts = 3$, und die *SSS* enthält genau $P[1]$ und $P[2]$. Für beide Punkte gilt

$$P[i].x > P[rechts].x - MinSoFar,$$

denn weil die Bedingung der **while**-Schleife falsch ist, haben wir

$$P[links].x > P[rechts].x - MinSoFar,$$

und jeder Punkt $P[i]$ in der *SSS* hat eine X-Koordinate $\geq P[links].x$. Also ist I_2 erfüllt.

Wenn die **while**-Schleife erneut betreten wird, müssen wir zunächst zeigen, daß die Invariante I_1 gilt. Dazu unterscheiden wir, welcher Zweig beim vorangehenden Durchlauf ausgeführt worden ist. War es der **then**-Zweig, so gilt I_1 unverändert. Lag der **else**-Fall vor, so hat sich die für I_1 relevante Punktmenge um den Punkt $P[rechts]$ vergrößert. Um den Wert von *MinSoFar* korrekt zu aktualisieren, wären eigentlich die Abstände von $P[rechts]$ zu *allen* Punkten mit kleinerem Index zu überprüfen gewesen. Die letzte Anweisung im **else**-Zweig hat diese Überprüfung nur für die Punkte in der *SSS* veranlaßt. Das genügt aber, weil nach I_2 die *SSS* zu dem Zeitpunkt alle Punkte enthielt, deren Abstand zu $P[rechts]$ überhaupt kleiner als *MinSoFar* sein kann. Auch in diesem Fall gilt also I_1 zu Beginn des neuen Schleifendurchlaufs.

Wenn bei diesem Durchlauf der **else**-Fall eintritt, müssen wir zeigen, daß I_2 erfüllt ist. Mit demselben Argument wie oben gilt

$$P[i].x > P[rechts].x - MinSoFar \tag{2.1}$$

für alle Punkte $P[i]$ in der *SSS*. Umgekehrt sei $P[i]$ mit $i \leq rechts - 1$ ein Punkt, der diese Ungleichung erfüllt. Dann ist er irgendwann in die *SSS* eingefügt worden. Wäre er inzwischen wieder entfernt worden, so hätten für die damaligen Werte *links'*, *rechts'* und *MinSoFar'* die Aussagen $i = links'$ und

$$P[i].x \leq P[rechts'].x - MinSoFar' \tag{2.2}$$

gelten müssen. Hieraus würde unter Benutzung der Ungleichungen 2.1 und 2.2 insgesamt der Widerspruch

$$0 \leq MinSoFar' - MinSoFar < P[rechts'].x - P[rechts].x \leq 0$$

folgen. Also ist $P[i]$ immer noch in der *SSS* enthalten, und I_2 gilt.

Übungsaufgabe 2.4 Wir wollen zeigen, daß es in einem Rechteck R mit den Kantenlängen M und $2M$ höchstens sechs Punkte geben kann, deren offene Umkreise vom Radius $M/2$ paarweise

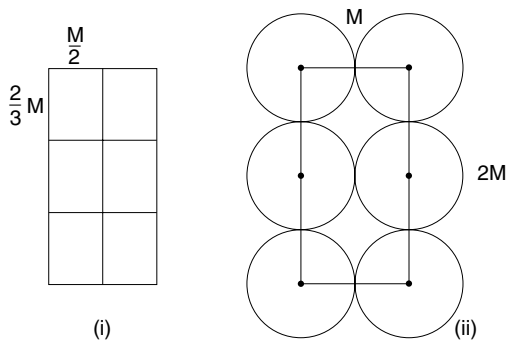


Abb. 2.28 Höchstens sechs Punkte mit gegenseitigem Mindestabstand M lassen sich im Rechteck R unterbringen.

disjunkt sind. Dazu wird R in sechs gleich große Rechtecke der Höhe $\frac{2}{3}M$ und der Breite $\frac{M}{2}$ geteilt, siehe Abbildung 2.28 (i).

Zwei Punkte in einem Rechteck können höchstens so weit voneinander entfernt sein wie seine Diagonale lang ist, hier also höchstens

$$\sqrt{\left(\frac{2}{3}M\right)^2 + \left(\frac{M}{2}\right)^2} = \frac{5}{6}M < M$$

Also kann in jedem der sechs kleinen Rechtecke höchstens ein Kreismittelpunkt sitzen. Eine – im übrigen auch die einzige! – solche Anordnung sehen wir in Abbildung 2.28 (ii).

Übungsaufgabe 2.5 Im Initialisierungsschritt werden die n Punkte nach ihren X- und nach ihren Y-Koordinaten sortiert (der Einfachheit halber nehmen wir an, daß keine zwei Punkte dieselben X- oder Y-Koordinaten haben). Nun beginnt der rekursive Hauptteil des Verfahrens. Wir benutzen die X-Sortierung, um die Punktmenge durch eine senkrechte Gerade G in eine linke und eine rechte Hälfte, L und R, zu teilen. Aus dem sortierten Y-Verzeichnis stellen wir zwei nach Y-Koordinaten sortierte Verzeichnisse Y_L und Y_R für die Punkte in L und R her (*divide*). Nun werden rekursiv die kleinsten Abstände M_L und M_R in L und R berechnet. Der kleinste Abstand in $L \cup R$ ist dann das Minimum der beiden Werte

$$\begin{aligned} \mathit{MinSoFar} &= \min(M_L, M_R) \\ M &= \min_{p \in L, q \in R} |pq|. \end{aligned}$$

Zur Berechnung von M kann man eine Art *sweep* verwenden: Wir schieben ein an der Geraden G zentriertes achsenparalleles Rechteck B der Höhe *MinSoFar* und Breite 2*MinSoFar* von oben nach

unten über die Ebene; siehe Abbildung 2.29. Wann immer ein neuer Punkt von B überdeckt wird, bestimmt man die Abstände zwischen ihm und allen Punkten im Rechteck, die auf der anderen Seite von G liegen. Wegen Übungsaufgabe 2.4 sind dabei höchstens sechs Vergleiche auszuführen. Das Minimum aller dieser Werte ist die gesuchte Zahl M. Damit ist der *conquer*-Schritt beendet.

Divide und *conquer* benötigen zusammen $O(n)$ Zeit. Insgesamt ergibt sich deshalb eine Laufzeit in $O(n \log n)$.

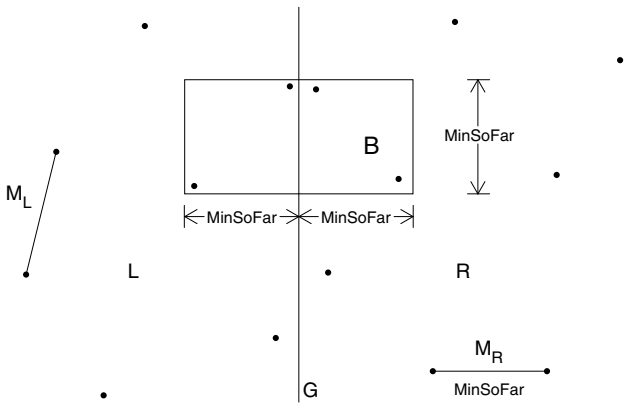


Abb. 2.29 Im *conquer*-Schritt wird das Rechteck B an der Geraden G hinabgeschoben. Der Abstand $|pq|$ ist kleiner als *MinSoFar*.

Übungsaufgabe 2.6 Nein. Zum Beispiel wird in Abbildung 2.30 der Schnittpunkt p zuerst entdeckt, aber jedes der beiden beteiligten Segmente hat einen weiter links gelegenen Schnittpunkt mit einem anderen Segment.

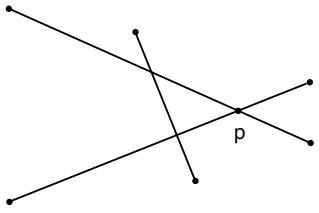


Abb. 2.30 Schnittpunkt p wird vom Sweep-Verfahren zuerst entdeckt.

Übungsaufgabe 2.7 Ja. In Abbildung 2.9 wird zum Beispiel der Schnittpunkt zwischen s_2 und s_3 gleich am Anfang entdeckt und ein weiteres Mal, nachdem s_4 gestorben ist.

Übungsaufgabe 2.8

(i) Durch die Scherung und die anschließende Translation ändert sich nicht die Ordnung der Segmente längs einer beliebigen Senkrechten zwischen S_0 und S_1. Außderdem bleiben die X-Koordinaten aller Schnittpunkte, die in L vorhanden sind, unverändert. Es entstehen auch keine neuen Schnittpunkte. Auch für die Menge L' allein stellt der Sweep-Algorithmus deshalb zwischen S_0 und S_1 fest, daß k Schnittpunkte rechts von S_1 liegen.

Wählt man C groß genug, so mischen sich die Segmente aus L und L' erst, nachdem die jeweils k vielen rechts von S_1 liegenden Schnittpunkte in L und L' entdeckt worden sind. Damit haben wir schon einmal $2k$ Schnittpunkte in $L \cup L'$ rechts von S_1 gefunden.

Wählt man, unabhängig von C, ε klein genug, so ist auf S_1 jedes Segment $s \in L$ zu „seinem“ gescherten $s' \in L'$ direkt benachbart. Da die beiden Segmente sich rechts von S_1 schneiden, ergibt das n weitere zukünftige Schnittereignisse, die zwischen S_0 und S_1 entdeckt werden, insgesamt also mindestens $2k + n$.

(ii) Wir starten nun mit einer zweielementigen Menge L und $k_1 = 1$. Bei Iteration dieser Konstruktion ist im i-ten Schritt $n_i = 2^i$ und

$$\begin{aligned} k_i &\geq 2k_{i-1} + 2^{i-1} \geq 2(2k_{i-2} + 2^{i-2}) + 2^{i-1} \\ &= 2^2 k_{i-2} + 2 \cdot 2^{i-1} \geq 2^2(2k_{i-3} + 2^{i-3}) + 2 \cdot 2^{i-1} \\ &= 2^3 k_{i-3} + 3 \cdot 2^{i-1} \\ &\vdots \\ &\geq 2^t k_{i-t} + t2^{i-1}, \end{aligned}$$

wie man leicht durch Induktion bestätigt. Für $t = i - 1$ ergibt sich wegen $k_1 = 1$ deshalb $k_i \geq i2^{i-1} = \frac{1}{2} n_i \log n_i$.

Übungsaufgabe 2.9 Wenn ein Schnittereignis E erzeugt wird, sind die beiden beteiligten Segmente in der *SSS* direkt benachbart. Das ändert sich aber, wenn eines von ihnen endet, sich mit seinem anderen Nachbarn schneidet oder wenn ein neues Segment entdeckt wird, das die beiden trennt. In jedem Fall ist E aus der Ereignisstruktur zu entfernen. Damit wir Ereignisse schnell auffinden können, verwalten wir für jedes Liniensegment Zeiger auf die höchstens zwei[19] Schnittereignisse in der *ES*, an denen es beteiligt ist.

[19] Es kommen ja nur die direkten Nachbarn als Partner in Frage.

Übungsaufgabe 2.10

(i) Es genügt zu zeigen, daß kein Liniensegment, das zwei Punkte aus verschiedenen vertikalen Gruppen miteinander verbindet, einen Winkel $< \alpha$ mit der negativen Y-Achse bilden kann. Für Punkte aus benachbarten Gruppen ist das nach Definition von α klar, für Punkte p, q aus nicht benachbarten Gruppen ergibt sich die Behauptung aus Abbildung 2.31. Hier ist r der unterste Punkt einer dazwischen liegenden Gruppe, falls r unterhalb von pq liegt, oder der oberste Punkt der Gruppe, falls dieser oberhalb von pq liegt. Mindestens einer der beiden Fälle muß eintreten.

(ii) Nein. In der neuen X'-Richtung erscheinen die Punkte in der lexikographischen XY-Reihenfolge.

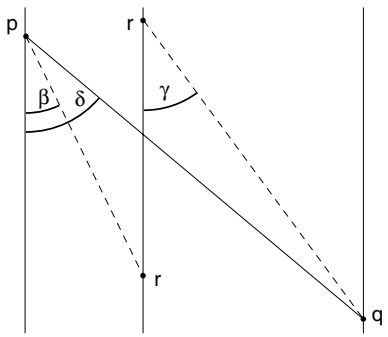


Abb. 2.31 Der Winkel δ ist größer als β und γ.

Übungsaufgabe 2.11

Beweis durch Induktion über m. Für $m = 2$ ist nichts zu zeigen. Angenommen, wir haben für $s_2, s_3, \ldots, s_m$ schon eine Anordnung mit der gewünschten Eigenschaft gefunden. Dann schieben wir das Segment s_1 in eine Position, in der es alle übrigen Segmente schneidet und alle Schnittpunkte auf s_1 links von allen übrigen Schnittpunkten liegen. Sollte eines der übrigen Segmente dafür zu kurz sein, muß die schon gefundene Anordnung vorher weiter zusammengeschoben werden.

Übungsaufgabe 2.12

(i) Daß die Ordnung s den Wert 1 hat, besagt, daß kein Buchstabe mehr als einmal verwendet werden darf. Würde er nämlich zweimal verwendet, müßte ein anderer Buchstabe dazwischen auftreten, und schon hätten wir zwei Wechsel.

(ii) Beweis durch Induktion über n. Für $n = 1$ stimmt die Behauptung, denn die einzige erlaubte Davenport-Schinzel-Sequenz, in der nur der Buchstabe A vorkommt, ist A selbst, weil $\ldots AA \ldots$ verboten ist. Sei also $n > 1$, und sei eine Sequenz der Ordnung 2 gegeben, in der n Buchstaben vorkommen. Sei Z derjenige Buchstabe in der Sequenz, dessen erstes Vorkommen am weitesten rechts liegt, und sei i die Position dieses ersten Vorkommens von Z; jeder Buchstabe $\neq Z$ an einer Position $> i$ muß also schon an einer Position $< i$ vorgekommen sein. Dann kann Z kein weiteres Mal in der Sequenz vorkommen: links von Position i nach Definition nicht, und rechts von Position i nicht, weil es sonst drei Wechsel in der Sequenz gäbe: Zwischen diesen beiden Vorkommen von Z müßte ja ein anderer Buchstabe U auftreten; U müßte dann auch links von Position i schon vorgekommen sein:

$$\begin{array}{c} U \ldots Z \ldots U \ldots Z \\ \uparrow \\ i \end{array}$$

Also kommt Z nur einmal vor. Wenn wir Z aus der Sequenz streichen, enthält das Resultat nur noch $n-1$ Buchstaben. Es muß sich aber um keine Davenport-Schinzel-Sequenz mehr handeln, weil der linke und rechte Nachbar von Z möglicherweise identisch waren und jetzt direkt benachbart sind. In dem Fall streichen wir außer Z auch einen von ihnen.

Wir haben jetzt eine Davenport-Schinzel-Sequenz der Ordnung 2 über $n-1$ Buchstaben. Nach Induktionsvoraussetzung hat sie höchstens die Länge $2(n-1)-1 = 2n-3$. Unsere Sequenz ist um höchstens zwei Stellen länger.

(iii) Die Anzahl der Kanten der unteren Kontur ist um höchstens 1 größer als die Anzahl der Ecken. Der Term $\frac{s(n-1)n}{2}$ ist sogar eine obere Schranke für die Anzahl *aller* Schnittpunkte im Arrangement.

Übungsaufgabe 2.13 Ja. Wir brauchen die *sweep line* nur durch eine Halbgerade zu ersetzen, die – wie ein Laserstrahl – um den Standpunkt p des Beobachters rotiert. Die Endpunkte und die Schnittpunkte der Liniensegmente beschreiben wir zweckmäßigerweise durch *Polarkoordinaten*. Dabei bedeutet $s = (a, \alpha)$, daß der Punkt s von p den Abstand a hat und mit der X-Achse den Winkel α einschließt, wie in Abbildung 2.32.

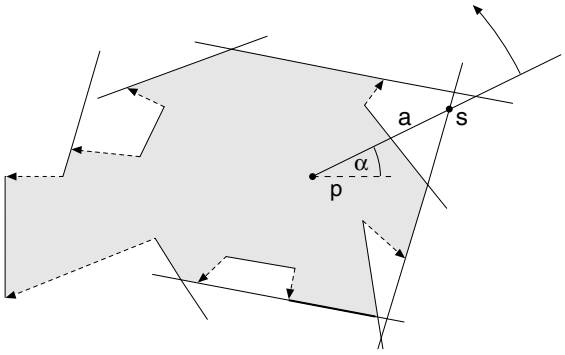


Abb. 2.32 Auch mit einer rotierenden Halbgeraden läßt sich die Ebene ausfegen.

Übungsaufgabe 2.14 Der Durchschnitt kann höchstens $m + n$ Ecken besitzen. Denn jede der Kanten von P und Q kann höchstens *eine* Kante zum Rand des Polygons $P \cap Q$ beitragen. Daß diese Schranke auch angenommen wird, zeigt das Beispiel von zwei konzentrischen regelmäßigen n-Ecken, von denen das eine gegenüber dem anderen um den Winkel $\frac{\pi}{n}$ verdreht ist.

3

Geometrische Datenstrukturen

3.1 Einführung

Beim Entwurf von Algorithmen steht man oft vor der Aufgabe, Mengen von Objekten so zu speichern, daß bestimmte Operationen auf diesen Mengen effizient ausführbar sind. Ein prominentes Beispiel stellt der abstrakte Datentyp *Verzeichnis* (oft auch *Wörterbuch* oder *dictionary* genannt) dar; hier sind die zu speichernden Objekte Elemente einer vollständig geordneten Grundmenge, und es sollen die Operationen Einfügen, Entfernen und Suchen ausführbar sein.

abstrakter Datentyp

Nun läßt sich zum Beispiel der abstrakte Datentyp *Verzeichnis* durch verschiedene *Datenstrukturen* implementieren, etwa durch lineare Listen oder durch AVL-Bäume. Für welche Datenstruktur man sich entscheidet, hat keinen Einfluß auf die Korrektheit des Verfahrens, wohl aber auf seine Effizienz! So kann man in linearen Listen zwar in Zeit $O(1)$ ein Objekt einfügen, aber Suchen und Entfernen kostet schlimmstenfalls $\Omega(n)$ viele Schritte, während beim AVL-Baum alle drei Operationen in Zeit $O(\log n)$ ausführbar sind (vorausgesetzt, ein Größenvergleich von zwei Objekten kann in Zeit $O(1)$ durchgeführt werden).

Datenstrukturen

Auch bei geometrischen Algorithmen spielen gute Datenstrukturen eine wichtige Rolle. Meistens nehmen die auszuführenden Operationen engen Bezug auf die geometrischen Eigenschaften der gespeicherten Objekte. Wir hatten zum Beispiel am Ende von Kapitel 2 vor dem Problem gestanden, eine Menge von Punkten in der Ebene so zu speichern, daß Einfügen und Entfernen von Punkten und Bereichsanfragen mit achsenparallelen Rechtecken („berichte alle gespeicherten Punkte, die im Rechteck liegen“) effizient ausführbar sind. Hieraus könnte sich eine Lösung des *closest pair*-Problems für n Punkte im $\mathbb{R}^3$ ergeben, die mit weniger als $O(n^2)$ Zeit auskommt.

Bereichsanfrage

Bereichsanfragen treten auch bei vielen anderen Problemen auf. Wir können zwei sehr allgemeine Grundtypen unterscheiden. In beiden Fällen sind n Datenobjekte $d_1, \ldots, d_n$ gleichen Typs zu speichern. Die Bereichsanfrage wird durch ein Anfrageobjekt q dargestellt. Meistens liegen die Objekte im $\mathbb{R}^d$.

Schnittanfrage

Beim Grundtyp *Schnittanfrage* müssen alle gespeicherten Datenobjekte d_i mit

$$d_i \cap q \neq \emptyset$$

Inklusionsanfrage

berichtet werden, bei der *Inklusionsanfrage* dagegen nur diejenigen d_i mit

$$d_i \subseteq q.$$

Das Anfrageobjekt q darf einen anderen Typ haben als die Datenobjekte. In der Ebene ist q oft ein achsenparalleles Rechteck,

Bildschirmausschnitt

das einen *Bildschirmausschnitt* darstellt. Alle gespeicherten Datenobjekte, die teilweise (Schnitt) oder ganz (Inklusion) in diesem Ausschnitt zu sehen sind, sollen angezeigt werden. Wenn die An-

Punkte als Anfrageobjekte

frageobjekte Punkte sind, ist die Inklusionsanfrage nur sinnvoll, wenn auch die Datenobjekte Punkte sind. Die Inklusionsanfrage degeneriert dann zur Suche nach einem vorgegebenen Punkt in einer Punktmenge. Dagegen ist die Schnittanfrage auch für „ausgedehnte“ Datenobjekte sinnvoll. Sie liefert alle Objekte, die einen vorgegebenen Punkt enthalten. In der Ebene wird dieser An-

Aufspießanfrage

fragetyp manchmal als *Aufspießanfrage* bezeichnet; dabei stellt man sich den Anfragepunkt als Heftzwecke vor, die die Datenobjekte aufspießt. Wenn es sich bei den Datenobjekten um Punkte

Punkte als Datenobjekte

handelt, fallen Schnitt- und Inklusionsanfrage zusammen. Dieser Fall ist besonders wichtig, denn oft lassen sich auch ausgedehnte geometrische Objekte durch Punkte in einem höherdimensionalen Raum darstellen.

So kann man zum Beispiel einen Kreis in der Ebene eindeutig durch die Koordinaten (x, y) seines Mittelpunkts und seinen Radius $r \geq 0$ beschreiben; das 3-Tupel (x, y, r) ist ein Punkt im $\mathbb{R}^3$.

Ebenso läßt sich ein achsenparalleles Rechteck als Punkt im $\mathbb{R}^4$ ansehen; man kann dazu die Koordinaten von zwei diagonalen Ecken verwenden, oder man nimmt die Koordinaten des Mittelpunkts und seine Entfernungen zu den Kanten.

Darstellung ausgedehnter Objekte durch Punkte

Bei einer solchen *Darstellung ausgedehnter Objekte durch Punkte* muß man überlegen, wie sich die geometrischen Relationen zwischen den Objekten auf die Punkte übertragen. Hierzu ein Beispiel:

Übungsaufgabe 3.1 Sei p ein Punkt in der Ebene. Man beschreibe die Menge aller Punkte im $\mathbb{R}^3$, die bei der oben angegebenen Darstellung denjenigen Kreisen in der Ebene entsprechen, die p enthalten.

Bereichsanfragen für Punkte als Datenobjekte spielen auch außerhalb der Geometrie eine wichtige Rolle. Angenommen, ein Betrieb hat für jeden Mitarbeiter einen Datensatz mit Eintragungen für die *Attribute* Name, Personalnummer, Lebensalter, Familienstand, Bruttogehalt und Anschrift angelegt. Ein solcher Datensatz ist ein Punkt im Raum

Attribute

$$\Sigma^* \times \mathbb{N} \times \mathbb{N} \times \{l, v, g, w\} \times \mathbb{R} \times \Sigma^*,$$

wobei Σ^* die Menge aller Wörter mit den Buchstaben $A, B, C, \ldots$ bedeutet und l, v, g, w Abkürzungen sind für *ledig*, *verheiratet* usw.

Die Frage nach allen ledigen Mitarbeitern in der Gruppe der 25- bis 30jährigen, die mindestens 2300 DM verdienen, läßt sich dann als Bereichsanfrage für das Anfrageobjekt

$$q = \Sigma^* \times \mathbb{N} \times [25, 30] \times \{l\} \times [2300, \infty) \times \Sigma^*$$

deuten. Man sieht, daß die Wertebereiche, aus denen die einzelnen Attribute stammen, nicht reellwertig zu sein brauchen.

Das Anfrageobjekt q enhält für jedes Attribut ein Intervall des Wertebereichs; im Extremfall kann es aus einem einzigen Element bestehen oder den gesamten Wertebereich einschließen. Solche Anfragen, die sich als Produkte von Intervallen der Wertebereiche schreiben lassen, nennt man *orthogonal*.

orthogonale Bereichsanfragen

In der Praxis werden Daten nicht im Hauptspeicher, sondern im Externspeicher, meist auf einer Festplatte, verwahrt, wenn ihr Volumen es erfordert, oder wenn sie auf Dauer benötigt werden. Hierzu zählen die Daten der Mitarbeiter eines Betriebs ebenso wie umfangreiche Mengen geometrischer Daten, wie sie etwa bei der Bearbeitung von Landkarten anfallen.

Auf Daten im Externspeicher kann nur *seitenweise* zugegriffen werden; dabei kann eine Datenseite dem Inhalt eines Sektors der Festplatte entsprechen, zum Beispiel 512 Byte. Für jeden Zugriff muß zunächst der Schreib-/Lesekopf positioniert werden. Bei heutigen Rechnern dauert das etwa 800mal länger als ein Zugriff auf ein Datenwort im Hauptspeicher. Man unterscheidet deshalb zwischen *internen* Datenstrukturen, die im Hauptspeicher leben, und *externen* Datenstrukturen, bei denen die Daten in Seiten (*buckets, pages*) zusammengefaßt sind, auf die nur am Stück zugegriffen werden kann. Für die Analyse der Zugriffskosten bei externen Strukturen ist unser Modell der REAL RAM aus Abschnitt 1.2.4 nicht geeignet (die RAM hatte unendlichen, permanenten Hauptspeicher und brauchte deshalb keinen Externspeicher!). Bei externen Datenstrukturen wird der Zeitbedarf durch die Anzahl der benötigten Seitenzugriffe abgeschätzt und der Speicherplatzbedarf durch die Anzahl der benötigten Datenseiten im Externspeicher.

interne und externe Datenstrukturen

Kosten

Wir werden in diesem Kapitel nur interne Datenstrukturen zur Speicherung von Punkten betrachten.

statische und dynamische Datenstrukturen

Wenn die Menge der zu speichernden Datenobjekte im wesentlichen unverändert bleibt, kann man sich mit *statischen* Datenstrukturen begnügen, die – einmal erzeugt – nur noch Anfragen zu unterstützen brauchen. Wenn sich dagegen der Datenbestand häufig ändert, werden *dynamische* Datenstrukturen benötigt, bei denen auch das Einfügen und Entfernen von Datenobjekten effizient möglich ist.

Standardbeispiel: sortiertes Array

Natürlich ist der Entwurf statischer Datenstrukturen einfacher. So läßt sich etwa ein eindimensionales Verzeichnis, in dem nur gesucht werden soll, bequem durch ein sortiertes Array implementieren, während für ein dynamisches Verzeichnis ein höherer Aufwand nötig ist. Verwendet man zum Beispiel AVL-Bäume, muß man Rotationen und Doppelrotationen implementieren. Benutzt man dagegen interne B-Bäume der Ordnung 1, so hat man mit den Rebalancierungsoperationen *split*, *balance* und *merge* zu tun; siehe z. B. Güting [69] oder andere Bücher über Datenstrukturen.

Dynamisierung als Paradigma

Erstaunlicherweise gibt es *allgemeine Techniken zur Dynamisierung* statischer Datenstrukturen, die uns die Beschäftigung mit solchen Details von Datenstrukturen ersparen können! Mit ihrer Hilfe lassen sich ohne großen Aufwand sortierte Arrays oder gewöhnliche Suchbäume, aber ebenso auch höherdimensionale Strukturen in effiziente dynamische Datenstrukturen verwandeln. Solche Dynamisierungstechniken sind auch für Anwendungen außerhalb der Algorithmischen Geometrie wichtig. Wir werden uns im nächsten Abschnitt ausführlich mit diesem Ansatz beschäftigen und dabei Methoden kennenlernen, die nicht ganz so

Leichte Implementierbarkeit

effizient arbeiten wie optimale Lösungen, aber dafür leicht zu implementieren sind; die Schwierigkeiten liegen statt dessen mehr in der Analyse.

3.2 Dynamisierung

Im Prinzip möchten wir uns die *Dynamisierung* in einem Modul implementiert denken. Das Modul importiert einen statischen abstrakten Datentyp *TStat*, in dem nur folgende Operationen erklärt sind:

build(V, D):	Erzeugt eine Struktur V vom Typ *TStat*, in der alle Datenobjekte der Menge D gespeichert sind.
query(V, q):	Beantwortet die Anfrage für die in V gespeicherten Datenobjekte und das Anfrageobjekt q.

extract(V, D): Faßt die in der Struktur V gespeicherten Objekte in der Menge D zusammen und gibt einen Zeiger auf D zurück.

Daneben gebe es eine triviale Löschoperation *erase*(V), die in $O(1)$ Zeit die Struktur samt ihrem Inhalt vernichtet.[1]

Exportiert wird ein dynamischer abstrakter Datentyp *TDyn* mit entsprechenden Operationen *Build*(W, D), *Query*(W, q) und *Extract*(W, D) sowie den folgenden Aktualisierungsoperationen:

Insert(W, d): Fügt Datenobjekt d in die Struktur W ein.

Delete(W, d): Entfernt Datenobjekt d aus der Struktur W.

Beide Datentypen beschreiben Strukturen V bzw. W zur Speicherung von Objekten d desselben Typs; siehe Abbildung 3.1. Zur Unterscheidung schreiben wir die Operationen auf dem dynamischen Datentyp mit großen Anfangsbuchstaben.

Natürlich hat das Modul *Dynamize* keinerlei Information über die Implementierung des Datentyps *TStat*. Wenn zum Beispiel die Datenobjekte und Anfrageobjekte Zahlen sind und *Query*(V, q) die gewöhnliche Suchanfrage bedeutet, könnte man zur Implementierung von *TStat* sortierte Arrays oder Suchbäume verwenden; hiervon ist *Dynamize* nichts bekannt.

Man könnte meinen, daß eine solche Dynamisierung „von außen“ nicht möglich ist. Doch hier ist eine triviale Lösung: Die Operationen *Build*, *Query*, und *Extract* werden unverändert von *TStat* übernommen. Einfügen und Entfernen werden kurzerhand wie folgt implementiert:

$$\begin{aligned} \mathit{Insert}(W, d): &\quad \mathit{Extract}(W, D);\ \mathit{Build}(W, D \cup \{d\}) \\ \mathit{Delete}(W, d): &\quad \mathit{Extract}(W, D);\ \mathit{Build}(W, D \setminus \{d\}). \end{aligned}$$

Wegwerf-Dynamisierung

Jedesmal die Struktur komplett neu aufzubauen ist natürlich sehr ineffizient! Wenn wir diesen Ansatz zum Beispiel auf eine Implementierung mit sortierten Arrays anwenden, kostet jede Einfüge- und Entferne-Operation $\Theta(n \log n)$ Zeit, denn *Build* würde ja die Datenobjekte bei jedem Aufruf neu sortieren, bevor das Array gefüllt wird. Trotzdem steckt in diesem Ansatz der Kern einer guten Idee, wie wir in den nächsten Abschnitten sehen werden.

In Abbildung 3.1 finden sich Bezeichnungen für die Laufzeiten der einzelnen Operationen und den Speicherplatzbedarf. Zum Beispiel ist $Q_V(n)$ der Zeitbedarf einer Anfrage an die Struktur V, die zur Zeit n Datenobjekte enthält. Dieser Zeitbedarf

[1] Man kann die alte Struktur unverändert einer *garbage collection* einverleiben, aus der man sich später beim Neubau bedient.

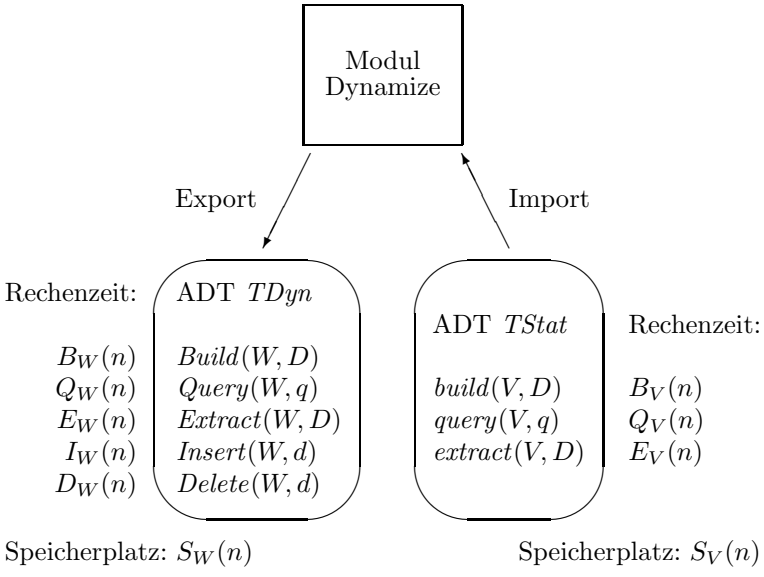


Abb. 3.1 Dynamisierung von Datentypen ohne Kenntnis ihrer Implementierung: die Idealvorstellung.

hängt davon ab, durch welche Datenstruktur (Array, Liste, Baum) der Typ *TStat* implementiert ist. Auf der linken Seite bezeichnet $Q_W(n)$ entsprechend den Aufwand einer Anfrage an die dynamische Struktur W mit n Elementen; dieser Aufwand hängt nicht nur von der Implementierung von *TStat* ab, sondern auch davon, wie geschickt wir den Typ *TDyn* im Modul *Dynamize* implementieren!

Damit beschäftigen wir uns in den folgenden Abschnitten. Wir werden hier die Operationen *Insert* und *Delete* so implementieren, daß sie *im Mittel* effizient arbeiten. Für praktische Anwendungen ist das oft schon ausreichend. Mit ähnlichen Methoden, aber erheblich höherem Aufwand läßt sich auch eine *worst-case*-effiziente Implementierung realisieren; siehe Overmars [117].

Annahmen über die Schranken

Zu den Laufzeiten und dem Speicherplatzbedarf der Implementierung des Datentyps *TStat* machen wir folgende Annahmen, die aus praktischer Sicht plausibel sind und uns die Analyse sehr erleichtern bzw. erst ermöglichen:

monoton

(1) Die Anfragezeit $Q_V(n)$ und die Zeit $E_V(n)$ für das Extrahieren der Datenobjekte wachsen monoton[2] in n; Beispiele für solche Funktionen sind: 1, $\log n$, $\sqrt{n}$, n, $n \log n$, n^2, 2^n.

[2] „Monoton“ bedeutet immer „schwach monoton“; $E_V(n) = 1$ wäre also erlaubt.

(2) Die Zeit $B_V(n)$ für den Aufbau und der Speicherplatzbedarf $S_V(n)$ wachsen sogar nach Division durch n noch monoton in n; Beispiele: n, $n \log n$, n^2, 2^n. — superlinear

(3) Für jede Funktion in $\{Q_V, B_V, E_V, S_V\}$ gibt es ein $C \geq 1$, so daß bei Verdoppelung des Arguments der Funktionswert höchstens um den Faktor C wächst; Beispiele: 1, $\sqrt{n}$, n, n^2, auch $\log n$ mit $n > 1$, sowie alle Produkte solcher Funktionen, aber nicht 2^n. — beschränktes Wachstum

(4) Die gespeicherten Datenobjekte aus einer Struktur zu extrahieren dauert höchstens so lange wie der Aufbau der Struktur. — Extrahieren schneller als Aufbauen

Eigenschaft (2) impliziert die Subadditivität der Funktionen $B_V(n)$ und $S_V(n)$, denn — Subadditivität

$$\frac{B_V(m)}{m} \leq \frac{B_V(m+n)}{m+n} \quad \text{und} \quad \frac{B_V(n)}{n} \leq \frac{B_V(m+n)}{m+n}$$

ergibt durch Addition

$$\begin{aligned} B_V(m) + B_V(n) &\leq m\frac{B_V(m+n)}{m+n} + n\frac{B_V(m+n)}{m+n} \\ &= B_V(m+n). \end{aligned}$$

Dies läßt sich durch Induktion auf mehr als zwei Argumente verallgemeinern. Ein paar Bemerkungen zu Eigenschaft (3) sind im folgenden zusammengestellt:

Übungsaufgabe 3.2 Sei $f : \mathbb{N} \longrightarrow \mathbb{R}_+$ monoton wachsend. Man zeige:

(i) Gibt es ein $k \in \mathbb{N}, k \geq 2$, für das ein $C > 1$ existiert mit $f(kn) \leq Cf(n)$ für alle n, so ist f polynomial nach oben beschränkt, d. h. es gibt ein $r \geq 1$ mit $f(n) \in O(n^r)$.

(ii) Es sei die Voraussetzung von (i) erfüllt. Dann gibt es für jedes α mit $0 < \alpha < 1$ ein $D > 1$, so daß für alle n gilt: $f(n) \leq D \cdot f(\lceil \alpha n \rceil)$.

Im übrigen gilt die Umkehrung von (i) nicht: Es gibt Funktionen $f(n) \in O(n^r)$, für die keine solchen Zahlen k und C existieren.

3.2.1 Amortisiertes Einfügen: die Binärstruktur

Wir betrachten in diesem Abschnitt eine einfache Möglichkeit, die Operation *Insert*(W, d) effizient zu implementieren. Die Idee besteht darin, die zu speichernden n Datenobjekte nicht in einer

einzigen statischen Struktur V unterzubringen, sondern sie auf mehrere Strukturen V_i zu verteilen. Wenn jetzt ein neues Objekt hinzukommt, müssen nicht alle V_i neu aufgebaut werden, sondern nur ein paar wenige. Sei

$$n = a_l 2^l + a_{l-1} 2^{l-1} + \ldots + a_1 2 + a_0 \text{ mit } a_i \in \{0, 1\}$$

die Darstellung von n im Binärsystem. Für jeden Koeffizienten $a_i = 1$ sehen wir eine Struktur V_i vor, die genau 2^i viele Objekte enthält; siehe Abbildung 3.2. Hierbei spielt es keine Rolle, welches Datenobjekt in welchem V_i gespeichert ist. Die Gesamtheit dieser Strukturen V_i ergibt die *Binärstruktur* W_n zur Speicherung von n Objekten.[3] Beim Aufbau von W_n ist also folgendes zu tun:

Binärstruktur

Build(W, D): Berechne die Binärdarstellung von $n = |D|$.
Zerlege D in Mengen D_i mit $|D_i| = 2^i$
entsprechend der Binärdarstellung.
Für jede Menge D_i führe *build*(V_i, D_i) aus.

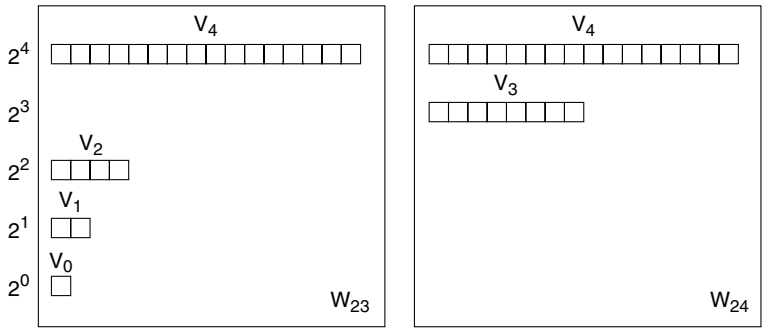


Abb. 3.2 Die Binärstruktur W_n enthält eine Struktur V_i für jedes i, für das 2^i in der Binärdarstellung von n mit dem Koeffizienten 1 auftritt, links für $n = 23$ rechts für $n = 24$.

Das folgende Lemma zeigt, daß man die Struktur W im wesentlichen genauso schnell aufbauen kann wie eine gleich große Struktur V.

Lemma 3.1 *Für den Zeitaufwand beim Aufbau der Binärstruktur W_n aus n Objekten gilt*

$$B_W(n) \in O(B_V(n)).$$

[3] Die V_i sind im Bild als Arrays dargestellt; es kann sich aber um ganz beliebige Strukturen handeln.

Beweis. Die Berechnung der Binärdarstellung von n und die Zerlegung der Datenmenge D können in Zeit $O(n)$ erfolgen.

Für die Ausführung einer Operation $build(V_i, D_i)$ wird $B_V(2^i)$ Zeit benötigt. Weil $i \leq l = \lfloor \log n \rfloor$ gilt,[4] ergibt das insgesamt höchstens

$$\begin{aligned}
\sum_{i=0}^{\lfloor \log n \rfloor} B_V(2^i) &= \sum_{i=0}^{\lfloor \log n \rfloor} 2^i \frac{B_V(2^i)}{2^i} \\
&\leq \sum_{i=0}^{\lfloor \log n \rfloor} 2^i \frac{B_V(n)}{n} \\
&\leq 2 \cdot 2^{\log n} \frac{B_V(n)}{n} \\
&\in O(B_V(n)).
\end{aligned}$$

In der zweiten Zeile haben wir Annahme (2) benutzt, nach der die Funktion $B_V(n)/n$ monoton wächst.

Insgesamt ergibt sich folgende Kostenschranke für den Bau von W_n:

$$B_W(n) \in O(n + B_V(n)) = O(B_V(n)), \qquad B_W(n)$$

weil $B_V(n)$ mindestens linear wächst. □

Um die Datenobjekte aus ganz W zu extrahieren, muß man sie in jeder Teilstruktur V_i einsammeln. Das kann insgesamt in Zeit

$$E_W(n) \leq \log n \cdot E_V(n) \qquad E_W(n)$$

erfolgen.

Anfragen

Kommen wir zu den Anfragen! Wir wollen die Operation $Query(W, q)$ dadurch implementieren, daß wir für jede Struktur V_i die Anfrage $query(V_i, q)$ beantworten und dann die Antworten zusammensetzen.

Für Bereichsanfragen, wie wir sie in Abschnitt 3.1 kennengelernt haben, ist solch ein Vorgehen ohne weiteres möglich, denn für eine Schnittanfrage gilt trivialerweise

$$\{d \in D\,;\; d \cap q \neq \emptyset\} = \bigcup_i \{d \in D_i\,;\; d \cap q \neq \emptyset\},$$

zerlegbare Anfrage

und mit Inklusionsanfragen verhält es sich genauso. Allgemein nennen wir $Query(W, q)$ eine *zerlegbare Anfrage*, wenn sich die Antwort in linearer Zeit aus den Antworten auf Anfragen $query(V_i, q)$ zusammensetzen läßt; hierbei nehmen wir an, daß die Objekte aus W auf Strukturen V_i verteilt sind.

[4] Mit $\log n$ meinen wir immer den Logarithmus zur Basis 2.

Außer Bereichsanfragen gibt es noch viele andere Anfragearten. Wenn zum Beispiel die Datenobjekte in D und das Anfrageobjekt q Punkte sind, könnte $Query(W, q)$ nach dem zu q nächstgelegenen Punkt fragen, der in W gespeichert ist. Diese Anfrage wäre ebenfalls zerlegbar. Fragen wir dagegen nach dem dichtesten Punktepaar in W, haben wir mit einer nicht zerlegbaren Anfrage zu tun, denn die beiden beteiligten Punkte brauchen nicht in derselben Struktur V_i zu stehen.

Ab jetzt nehmen wir an, daß $Query(W, q)$ zerlegbar ist. Wir erhalten für die oben angegebene Implementierung von $Query(W, q)$ die Laufzeit

$Q_W(n)$

$$\begin{aligned} Q_W(n) &\leq \sum_{i=0}^{\lfloor \log n \rfloor} Q_V(2^i) + C \cdot \log n \\ &\leq D \cdot (\log n \cdot Q_V(n) + C \cdot \log n) \\ &\in O(\log n \cdot Q_V(n)) \end{aligned}$$

wegen der Monotonie von $Q_V(n)$; dabei bezeichnet $C \cdot \log n$ den Aufwand für das Zusammensetzen der $\log n$ vielen Teilantworten.

Für den Speicherplatzbedarf gilt

$S_W(n)$

$$S_W(n) \leq \sum_{i=0}^{\lfloor \log n \rfloor} S_V(2^i) \in O(S_V(n))$$

mit derselben Begründung wie im Beweis von Lemma 3.1.

Einfügen

Schließlich kommen wir zum Einfügen, unserem eigentlichen Ziel. Wenn sich die Anzahl der Datenobjekte von n auf $n + 1$ erhöht, muß die Menge der statischen Strukturen V_i so aktualisiert werden, daß sie danach der Binärdarstellung von $n+1$ entspricht.

Im Beispiel von Abbildung 3.2 wird durch Einfügen eines neuen Objekts d aus $n = 23 = 10111_2$ die Zahl $n + 1 = 24 = 11000_2$. Wir müssen also hier die Operationen

$$\begin{array}{l} extract(V_0, D_0);\ extract(V_1, D_1);\ extract(V_2, D_2); \\ D := D_0 \cup D_1 \cup D_2 \cup \{d\}; \\ build(V_3, D); \end{array}$$

ausführen.

Allgemein gilt: Wenn an den letzten j Stellen in der Binärdarstellung von n lauter Einsen stehen und davor eine Null, müssen $V_0, \ldots, V_{j-1}$ gelöscht werden, und V_j wird neu aufgebaut. Wenn wir annehmen, daß sich die Menge D in Zeit $O(j)$ aus $D_0, \ldots, D_{j-1}$ und d zusammensetzen läßt, erhalten wir für

jede Zahl $n = k2^{j+1} + 2^j - 1$, die mit genau j Einsen endet, die Abschätzung

$$\begin{aligned} I_W(n) &\leq \left(\sum_{i=0}^{j-1} E_V(2^i)\right) + Cj + B_V(2^j) \\ &\leq \left(\sum_{i=0}^{j-1} B_V(2^i)\right) + Cj + B_V(2^j) \\ &\in O\left(B_V(2^j)\right) \end{aligned}$$

wie im Beweis von Lemma 3.1.

Wenn die Zahl k den Wert 0 hat, ist $B_V(2^j)$ so groß wie $B_V(n)$; auf den ersten Blick sieht es deshalb so aus, als hätten wir gegenüber der naiven Wegwerf-Implementierung, bei der jedesmal die gesamte Struktur neu aufgebaut wird, nichts gewonnen.

In Wahrheit haben wir aber einen großen Vorteil erzielt: Zwar kann es für spezielle Werte von n vorkommen, daß unsere Struktur komplett oder zu einem großen Teil neu aufgebaut werden muß, aber dieser Fall tritt nicht so häufig ein!

Betrachten wir noch einmal Abbildung 3.2. Beim Einfügen des 24. Datenobjekts muß zwar eine Teilstruktur für acht Objekte komplett neu gebaut werden, aber das schafft Platz auf den unteren drei Etagen von W, so daß sich die nächsten Objekte mit sehr geringem Aufwand einfügen lassen. Anders ausgedrückt: Die Investition an Umbauarbeit *amortisiert sich* im Laufe der Zeit!

Amortisierung

Die interessante Frage lautet also: Wie hoch sind die Kosten *im Mittel*, wenn man eine längere Folge von Einfüge-Operationen betrachtet? Die Betrachtung der mittleren Kosten pro Operation ist für die Praxis durchaus sinnvoll. Wer nacheinander fünfzig neue Objekte eingeben muß, interessiert sich hauptsächlich für den im ungünstigsten Fall entstehenden Gesamtaufwand und nicht für den Zeitaufwand jeder einzelnen Operation.

Allgemein stellen wir uns vor, daß wir mit einer leeren Struktur W starten. Nun werden in bunter Folge s Operationen unterschiedlichen Typs ausgeführt. Eine von ihnen heiße *Work*. Angenommen, es kommen k Aufrufe von *Work* in der Operationenfolge vor. Gilt dann stets

$$\frac{\text{Gesamtkosten der } k \text{ } \textit{Work}\text{-Operationen}}{k} \leq \overline{W}(s)$$

für eine monotone Kostenfunktion $\overline{W}(s)$, so sagen wir, die Operation *Work* sei in *amortisierter Zeit* $\overline{W}(s)$ ausführbar.

amortisierte Kosten

Zwei Tatsachen sind hier zu beachten: Erstens handelt es sich bei dieser Mittelbildung nicht um Wahrscheinlichkeiten; wir berechnen keine Erwartungswerte, sondern den *worst case* für eine Folge von Operationen!

Zweitens betrachten wir bei dieser Definition die amortisierten Kosten $\overline{W}$ nicht als Funktion der aktuellen Strukturgröße n, sondern als Funktion von s, der Länge der Operationenfolge.[5] Zwischen n und s besteht aber ein Zusammenhang: Wenn neue Objekte nur einzeln in die Struktur eingefügt werden können, muß stets $n \leq s$ sein, denn für jedes der n Objekte kommt in der Folge ja eine Einfügung vor.

Übungsaufgabe 3.3 Angenommen, die Kosten einer einzelnen Operation $Work(W_n)$ sind stets kleiner gleich $A(n)$, wobei A eine monoton wachsende Funktion ist. Man zeige, daß dann $A(s)$ auch eine obere Schranke für die amortisierten Kosten von *Work* darstellt.

Bei unserer Binärstruktur gibt es außer den *Insert*-Operationen zur Zeit nur *Query*-Operationen, die an der Größe der Struktur nichts verändern. Wir können uns deshalb zur Ermittlung der amortisierten Einfügekosten auf Operationenfolgen beschränken, die aus lauter *Insert*-Operationen bestehen; es ist dann $s = n$.

bis jetzt: nur *Insert*

Um den Gesamtaufwand für die ersten n Einfügungen abzuschätzen, zählen wir, wie oft beim Hochzählen von 0 auf n der Koeffizient a_j von 2^j in der Binärdarstellung von null auf eins springt; jedesmal entstehen Kosten in Höhe von $B_V(2^j)$. Offenbar springt a_0 bei jedem zweiten Schritt auf eins, a_1 bei jedem vierten, und so fort. Insgesamt ergibt sich:

$$\begin{aligned}
\sum_{i=1}^{n} I_W(i) &\leq D \cdot \sum_{j=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{j+1}} \right\rceil B_V(2^j) \\
&\leq Cn \sum_{j=0}^{\lfloor \log n \rfloor} \frac{B_V(2^j)}{2^j} \\
&\leq Cn \sum_{j=0}^{\lfloor \log n \rfloor} \frac{B_V(n)}{n} \\
&\in O(\log n \cdot B_V(n)).
\end{aligned}$$

Die mittleren Kosten der ersten n Einfügeoperationen liegen daher in

$$O\left(\frac{\log n}{n} B_V(n)\right).$$

[5] Für kurze Folgen, in denen zum Beispiel keine *Query*-Operationen vorkommen, wird die Abschätzung besonders aussagekräftig.

Fassen wir zusammen!

Theorem 3.2 *Zu einem statischen Datentyp wie in Abbildung 3.1 erlaubt es die beschriebene Binärstruktur, die Operation Insert(W, d) in amortisierter Zeit*

$$\overline{I}_W(s) \in O\left(\frac{\log s}{s} B_V(s)\right)$$

auszuführen. Gegenüber der statischen Implementierung steigt dabei der Zeitbedarf für Anfragen und für das Extrahieren aller Datenobjekte um den Faktor $\log n$*, wobei* n *die aktuelle Strukturgröße bezeichnet; die Kosten für den Aufbau der Struktur und der Speicherplatzbedarf bleiben der Größe nach unverändert.*

Standardbeispiel

Betrachten wir unser Standardbeispiel: sortierte Arrays zur Unterstützung von Suchanfragen auf n Zahlen. Hier ist $B_V(n) \in O(n \log n)$. Also können wir in amortisierter Zeit $O((\log n)^2)$ eine neue Zahl einfügen. Das Suchen in der halbdynamischen Struktur erfolgt in Zeit $O((\log n)^2)$.

3.2.2 Amortisiertes Entfernen durch gelegentlichen Neubau

Objekte zu entfernen ist schwieriger als neue einzufügen, weil wir uns den Ort in der Struktur, an dem ein Objekt entfernt werden soll, nicht aussuchen können.

Bei manchen Datenstrukturen ist es einfacher, ein Objekt *als entfernt zu markieren,* anstatt es physisch zu entfernen. So läßt sich zum Beispiel in einem sortierten Array jedes Element in Zeit $O(\log n)$ finden und mit einer Markierung versehen, während bei einer echten Entfernung $\Theta(n)$ Zeit für das Schließen der Lücke benötigt würde.

Als entfernt markierte Objekte zählen nicht mehr zur aktuellen Datenmenge D; bei Anfragen werden sie übergangen. Sie beanspruchen aber weiterhin Speicherplatz und kosten Rechenzeit: Wenn im Array der Größe n alle Objekte bis auf r viele als entfernt markiert sind, benötigt eine Suchanfrage immer noch $\Theta(\log n)$ Zeit, statt $\Theta(\log r)$.

Der Wirkungsgrad wird um so schlechter, je größer n, die Größe der Struktur, im Vergleich zu r, der Größe der Datenmenge, wird.

Wir sagen, die Operation *delete* sei *schwach implementiert*, wenn ihre Ausführung die Größe der Struktur unverändert läßt. Die Kosten sämtlicher Operationen hängen von der Größe der Struktur ab; sie sind also so hoch, als hätte seit dem Bau der

schwaches Entfernen

Struktur kein *delete* stattgefunden. Eine schwache Implementierung von *delete* nennt man manchmal auch *schwaches Entfernen*.

$WD_V(n)$

Wir setzen ab jetzt voraus, daß der gegebene statische Datentyp zusätzlich eine schwach implementierte *delete*-Operation mit den Kosten $WD_V(n)$ anbietet. Auch für diese Kostenfunktion wird vorausgesetzt, daß bei Vervielfachung des Arguments der Wert nur um einen festen Faktor wächst, siehe Eigenschaft (3) auf Seite 113.

In diesem Abschnitt wollen wir zeigen, wie man aus einer schwachen Implementierung von *delete* eine „starke“ macht, die ein gutes amortisiertes Laufzeitverhalten aufweist.

Die Idee ist ganz naheliegend: Wann immer eine Struktur V frisch aufgebaut ist, werden *Delete*-Operationen zunächst durch schwaches Entfernen realisiert. Sobald die Menge der aktuell vorhandenen Objekte nur noch halb so groß ist wie die Struktur V, wird V gelöscht und eine neue Struktur aus den vorhandenen Objekten aufgebaut. Die Kosten dafür werden rechnerisch auf die vorangegangenen Entferne-Operationen umgelegt. Dieser Ansatz, der im übrigen auch von Hash-Verfahren bekannt ist, führt zu folgendem Ergebnis:

Theorem 3.3 *Gegeben sei eine Implementierung eines Datentyps mit den Kosten*

$$B_V(n), E_V(n), Q_V(n), I_V(n), S_V(n), WD_V(n)$$

für build, extract, query, insert, Speicherplatzbedarf und schwaches delete; dabei bezeichnet n die Größe der Struktur V einschließlich der schwach entfernten Objekte. Dann läßt sich durch gelegentlichen Neubau eine Struktur W implementieren mit den Kosten

$$\begin{aligned}
B_W(r) &= B_V(r) \\
E_W(r) &\in O\left(E_V(r)\right) \\
Q_W(r) &\in O\left(Q_V(r)\right) \\
I_W(r) &\in O\left(I_V(r)\right) \\
S_W(r) &\in O\left(S_V(r)\right) \\
\overline{D}_W(s) &\in O\left(WD_V(s) + \frac{B_V(s)}{s}\right),
\end{aligned}$$

wobei r die Größe der aktuellen Datenmenge bezeichnet und s die Länge einer Operationenfolge bei anfangs leerer Struktur.

Beweis. Die Struktur W wird durch eine Struktur V realisiert, die zu jedem Zeitpunkt höchstens die Größe $2r$ hat. Daraus folgt

$$Q_W(r) \leq Q_V(2r) \leq C \cdot Q_V(r)$$

wegen Eigenschaft (3) der Kostenfunktionen. Dasselbe Argument gilt für $E_W(r)$, $I_W(r)$ und $S_W(r)$.

Sei nun eine Folge von s Operationen gegeben, in der k Entferneoperationen

$$Delete(W_{n_1}, d_1), \ldots, Delete(W_{n_k}, d_k)$$

vorkommen. Wenn bei Ausführung von $Delete(W_{n_i}, d_i)$ kein Neubau erforderlich ist, kostet das schwache *delete* nur $WD_V(n_i)$ viel Zeit; wegen $n_i \leq s$ ergibt das insgesamt höchstens

$$\sum_{i=1}^{k} WD_V(n_i) \leq k \cdot WD_V(s).$$

Für diejenigen $n_{i_j}, 1 \leq j \leq t$, mit $n_{i_j} = 2r_{i_j}$ entstehen zusätzlich Baukosten in Höhe von $B_V(r_{i_j})$. Dann sind seit dem letzten Neubau aber r_{i_j} schwache *delete*-Operationen ausgeführt worden, weil die aktuelle Datenmenge ja jetzt r_{i_j} Objekte weniger enthält als die Struktur. Also ist

$$r_{i_1} + \ldots + r_{i_t} \leq k,$$

und es folgt

$$\begin{aligned}
\frac{\sum_{j=1}^{t} B_V(r_{i_j})}{k} &\leq \frac{\sum_{j=1}^{t} B_V(r_{i_j})}{r_{i_1} + \ldots + r_{i_t}} \\
&= \sum_{j=1}^{t} \frac{r_{i_j}}{r_{i_1} + \ldots + r_{i_t}} \cdot \frac{B_V(r_{i_j})}{r_{i_j}} \\
&\leq \sum_{j=1}^{t} \frac{r_{i_j}}{r_{i_1} + \ldots + r_{i_t}} \cdot \frac{B_V(s)}{s} \\
&= \frac{B_V(s)}{s}.
\end{aligned}$$

Insgesamt liegen daher die mittleren Kosten der k vielen *Delete*-Operationen in $O\Big(WD_V(s) + \frac{B_V(s)}{s}\Big)$. □

3.2.3 Amortisiertes Einfügen und Entfernen

Wir wollen jetzt zeigen, wie sich die Binärstruktur und die Methode des gelegentlichen Neubaus miteinander kombinieren lassen, um Datenobjekte sowohl einfügen als auch entfernen zu können.

Gegeben sei also ein statischer Datentyp mit schwach implementiertem *delete*. Wie in Abschnitt 3.2.1 wird zunächst die

Operation *Insert*(W, d) mit der Binärstruktur implementiert. Das schwache Entfernen steht nun zwar auf jeder Teilstruktur V_i von W zur Verfügung, aber noch nicht auf W selbst: Wenn ein Objekt d als entfernt markiert werden soll, wissen wir zunächst nicht, in welchem V_i es enthalten ist.

Binärstruktur mit balanciertem Suchbaum

Deshalb wird die Binärstruktur durch einen balancierten Suchbaum T ergänzt, der für jedes in W gespeicherte Objekt d einen Verweis auf diejenige Teilstruktur V_i enthält, in der d sich befindet. Der Baum T ermöglicht es auch, das Einfügen bereits vorhandener Objekte zu verhindern.

Hierbei nehmen wir an, daß die Objekte über eine eindeutige Kennzeichnung (einen sog. *Schlüssel* oder *key*) identifiziert werden können, die einer vollständig geordneten Menge entstammt, und daß ein Größenvergleich in Zeit $O(1)$ möglich ist.

Eine „frisch aufgebaute“ Struktur W_{15} sähe dann nach Entfernen der Objekte 11 und 1 aus wie in Abbildung 3.3 dargestellt.

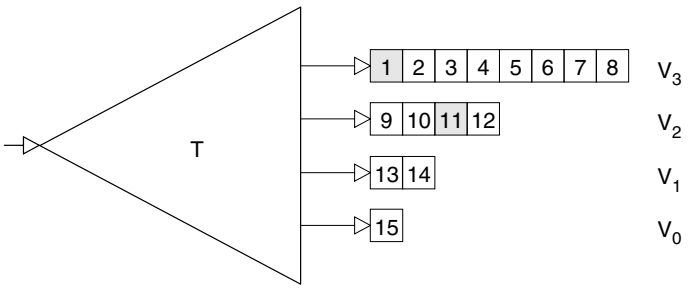


Abb. 3.3 Die Struktur W_{15} mit einem Suchbaum T für die Teilstrukturen V_i der gespeicherten Objekte.

Kosten des Suchbaums

Welche zusätzlichen Kosten entstehen durch die Verwaltung des Suchbaums T? Von *Query*-Operationen ist T nicht betroffen. Beim schwachen *Delete*(W, d) kommt ein additiver Betrag in $O(\log n)$ für die Suche in T und das anschließende Entfernen des Eintrags für d aus T hinzu.

Wird ein Objekt d neu eingefügt, muß in T ein Eintrag für d angelegt werden. Aber damit nicht genug: Beim Einfügen von d verschwinden Teilstrukturen $V_0, \ldots, V_{j-1}$, und ein V_j wird neu erzeugt. Für alle darin enthaltenen Datenobjekte müssen die Verweise jetzt auf V_j zeigen.

Diese Änderung läßt sich einfach durchführen, wenn wir alle Verweise in T, die auf dieselbe Teilstruktur V_i zeigen, in einer verketteten Liste zusammenfassen. Wir brauchen dann nur die Listen von V_0 bis V_{j-1} zu durchlaufen, zusammenzufügen und alle Einträge in „V_j“ umzuändern. Das kann in Zeit proportional

zur Größe 2^j von V_j geschehen. Diese zusätzlichen Kosten fallen nicht ins Gewicht, weil für den Neubau von V_j ohnehin $O(B_V(2^j))$ viel Zeit benötigt wird.

Auf dieser Basis implementieren wir jetzt die *Delete*-Operation wie in Abschnitt 3.2.2 beschrieben: Wenn so viele Elemente schwach entfernt worden sind, daß nur noch die Hälfte aller gespeicherten Objekte zur aktuellen Datenmenge gehört, wird die ganze Struktur komplett neu aufgebaut. Für den Neubau des Binärbaums T fallen dabei nur lineare Kosten an, da die Elemente ja schon sortiert sind. Analog zu Theorem 3.3 ergibt sich die Schranke

gelegentlicher Neubau

$$\overline{D}_W(s) \in O\left(\log s + WD_V(s) + \frac{B_V(s)}{s}\right)$$

für die amortisierten Kosten des Entfernens; dabei schätzt $\log s \geq \log n$ die Zeit für die Suche in T bei einer Strukturgröße n ab.

Wie sieht es nun mit den Kosten des Einfügens aus? In Abschnitt 3.2.1 hatten wir ja vor Theorem 3.2 nur Folgen von lauter *Insert*-Operationen betrachtet. Unsere Abschätzung beruhte darauf, daß beim schrittweisen Hochzählen das zu 2^j gehörende Bit a_j nur bei jedem 2^{j+1}-ten Schritt auf eins springt und dabei Kosten in Höhe von $B_V(2^j)$ entstehen.

Amortisiert sich der Einfügeaufwand?

Hieran ändert sich nichts, wenn einzelne Objekte als entfernt markiert sind, weil sie bei der Aktualisierung der Binärstruktur wie aktuelle Objekte behandelt werden. Wenn aber zwischendurch *viele Delete*-Operationen stattfinden und die Struktur in halber Größe neu aufgebaut wird, entspricht dem ein *shift* der Binärdarstellung um eine Stelle nach rechts. Das vorher zu 2^j gehörende Bit wird dadurch zum Koeffizienten von 2^{j-1}; also springt es bei den nachfolgenden Einfügeoperationen doppelt so oft auf eins wie bisher. Andererseits halbieren sich aber die Kosten, die dabei jedesmal entstehen! Insgesamt werden die Einfügekosten also durch einen gelegentlichen Neubau der Struktur nicht größer. Es folgt:

Neubau verursacht *shift*

Theorem 3.4 *Gegeben sei ein statischer Datentyp mit schwach implementiertem delete. Die Kombination von Binärstruktur und gelegentlichem Neubau liefert eine Dynamisierung mit folgenden Eigenschaften: Einfügen kann man in amortisierter Zeit*

$$\overline{I}_W(s) \in O\left(\log s \frac{B_V(s)}{s}\right),$$

Entfernen in amortisierter Zeit

$$\overline{D}_W(s) \in O\left(\log s + WD_V(s) + \frac{B_V(s)}{s}\right).$$

Der Zeitaufwand für Anfragen und für das Extrahieren aller Datenobjekte steigt um den Faktor $\log r$*, wobei* r *die Größe der aktuellen Datenmenge bezeichnet. Die übrigen Kosten ändern sich der Größe nach nicht.*

Damit haben wir unser Ziel erreicht und einen statischen Datentyp so dynamisiert, daß die Kosten von *Insert* und *Delete* gering sind, wenn man über eine Folge von Operationen mittelt.

alternative Amortisierungsbegriffe

Es gibt übrigens verschiedene Vorschläge dafür, wie diese Mittelung vorzunehmen ist. Man kann großzügiger sein, als wir es bei der Definition der amortisierten Kosten in Abschnitt 3.2.1 waren, und zwischen den verschiedenen Typen von Operationen nicht unterscheiden; dabei dürften dann die Kosten für das Entfernen eines Objekts dem Einfügen mit in Rechnung gestellt werden. Man kann aber auch strenger sein und zum Beispiel verlangen, daß der Mittelwert aus den Kosten einer Operation $Insert(W_n, d)$ und den Kosten der vorausgegangenen m *Insert*-Operationen durch eine Funktion der aktuellen Strukturgröße n beschränkt ist; hierbei darf $m \geq 0$ frei gewählt werden.

Übungsaufgabe 3.4 Man zeige, daß bei Verwendung dieser strengen Definition die Aussage von Theorem 3.4 für die *Insert*-Operation nicht mehr gilt. Hinweis: Man betrachte eine Folge von $2^{a+b} - 2^a$ Einfügungen in die anfangs leere Struktur, an die sich $2^{a+b} - 2^a - 2^b + 1$ Entferneoperationen und eine weitere Einfügung anschließen.

schwaches Einfügen

Abschließend sei erwähnt, daß sich die Operation *Insert* besonders einfach implementieren läßt, wenn der zugrundeliegende statische Datentyp ein *schwaches Einfügen* $WI(V_n, d)$ anbietet. Dies ist oft der Fall, wenn zur Implementierung Bäume verwendet werden; man kann dann den Ort suchen, an dem das Objekt d seiner Größe nach stehen müßte, es dort einfügen und auf eine Rebalancierung verzichten. Hierdurch kann zwar der Baum im Laufe der Zeit degenerieren und hohe Zugriffskosten verursachen, aber dagegen hilft die Technik aus Abschnitt 3.2.2: ein Neubau, sobald die Kosten der Operationen doppelt so hoch werden, wie sie nach Anzahl der aktuellen Datenobjekte eigentlich sein müßten.

In den vorangegangenen Abschnitten haben wir die Kosten von Umbauarbeiten *rechnerisch* über die Zeit verteilt und so eine niedrige obere Schranke für den mittleren Aufwand erhalten.

Verteilen des Umbaus über die Zeit

Wenn man daran interessiert ist, die Kosten einer jeden Aktualisierungsoperation auch im *worst case* niedrig zu halten, muß man einen Schritt weitergehen und die Umbauarbeiten selbst über die Zeit verteilen.

Dabei entsteht ein Problem: Eine Struktur, die sich noch im Umbau befindet, steht für Anfragen nicht zur Verfügung. Jedes Datenobjekt, das zu einem gerade im Umbau befindlichen Teil der Struktur gehört, muß deshalb noch ein zweites Mal gespeichert sein, in einem für *Query*-Operationen zugänglichen Teil. Man kann zeigen, daß dieser Ansatz tatsächlich funktioniert. Es ergibt sich damit folgende Verschärfung von Theorem 3.4:

Theorem 3.5 *Sei TStat ein abstrakter Datentyp mit Operationen, wie sie am Anfang dieses Abschnitts beschrieben wurden. Dann läßt sich damit ein dynamischer Datentyp TDyn folgendermaßen implementieren:*

$$
\begin{array}{ll}
\mathit{Build}(W,D) & \text{in Zeit } O(B_V(n)) \\
\mathit{Query}(W,q) & \text{in Zeit } O(\log n \cdot Q_V(n)) \\
\mathit{Insert}(W,d) & \text{in Zeit } O\left(\frac{\log n}{n} B_V(n)\right) \\
\mathit{Delete}(W,d) & \text{in Zeit } O\left(\log n + WD_V(n) + \frac{B_V(n)}{n}\right) \\
\text{Speicherplatzbedarf} & O(S_V(n)).
\end{array}
$$

Dabei bezeichnet n die Anzahl der zum jeweiligen Zeitpunkt gespeicherten Datenobjekte, und alle Schranken gelten im worst case.

Wer sich für den — recht komplizierten — Beweis von Theorem 3.5 interessiert, sei auf das Buch [117] von Overmars verwiesen. Dort und in Mehlhorn [103] finden sich auch die Resultate über Dynamisierung mit günstigen amortisierten Kosten, die wir in diesem Abschnitt vorgestellt haben. Eine allgemeine Einführung in die Analyse amortisierter Kosten geben auch Cormen et al. [36].

Nach diesem Ausflug in das Gebiet der Dynamisierung kehren wir nun zur Geometrie zurück und betrachten geometrische Datenstrukturen für Punkte, mit denen sich die in Abschnitt 3.1 beschriebenen Anfrageprobleme effizient lösen lassen.

zurück zur Geometrie!

3.3 Interne Datenstrukturen für Punkte

Man kann Punkte im $\mathbb{R}^k$ in einem gewöhnlichen „eindimensionalen“ Suchbaum speichern, wenn man als Ordnung zum Beispiel die *lexikographische Ordnung* verwendet:

lexikographische Ordnung

$$
(p_1,\ldots,p_k) < (q_1,\ldots,q_k) \iff \begin{array}{l} \text{es gibt ein } i \text{ mit } 0 \le i \le k-1, \\ \text{so daß } p_j = q_j \text{ für } 1 \le j \le i \\ \text{und } p_{i+1} < q_{i+1} \text{ ist.} \end{array}
$$

Hat der Suchbaum dann logarithmische Höhe, benötigt die Suche nach einem einzelnen Punkt $\Theta(k \log n)$ viel Zeit, weil für jeden Größenvergleich von zwei Punkten schlimmstenfalls $\Omega(k)$ einzelne Vergleiche erforderlich sind.

Anfragen mit mehrdimensionalen Anfrageobjekten werden aber von eindimensionalen Suchbäumen nicht effizient unterstützt.

Übungsaufgabe 3.5 Seien p, q zwei Punkte in der Ebene, so daß p lexikographisch kleiner als q ist. Man beschreibe das Intervall $[p, q]$ geometrisch.

Wir werden uns deshalb im folgenden mit Datenstrukturen beschäftigen, die *alle* Dimensionen berücksichtigen.

3.3.1 Der *k*–d–Baum

k–d–Baum

Unser erstes Beispiel ist der k-dimensionale Suchbaum, kurz *k–d–Baum* genannt. Wie sein Name[6] schon andeutet, stellt er eine natürliche Verallgemeinerung des eindimensionalen Suchbaums dar. Sei also D eine Menge von n Punkten im $\mathbb{R}^k$, die es zu speichern gilt. Zur Vereinfachung nehmen wir zunächst an, daß sich die Punkte aus D in allen Koordinaten voneinander unterscheiden. Außerdem wird zunächst nur der Fall $k = 2$ betrachtet.

vereinfachende Annahme

Splitkoordinate

Splitwert

Wir wählen nun eine der Koordinaten als *Splitkoordinate*, zum Beispiel X, und bestimmen einen *Splitwert* s, der selbst nicht als X-Koordinate eines Punktes in D vorkommt. Dann wird die Punktmenge D durch die *Splitgerade* $X = s$ in die beiden Teilmengen

$$\begin{aligned} D_{<s} = \{(x, y) \in D; x < s\} &= D \cap \{X < s\} \\ D_{>s} = \{(x, y) \in D; x > s\} &= D \cap \{X > s\} \end{aligned}$$

zerlegt. Für $D_{<s}$ und $D_{>s}$ verfährt man jeweils genauso wie mit D, nur wird diesmal Y als Splitkoordinate verwendet. Danach ist wieder X an der Reihe, und so fort, bis lauter einelementige Punktmengen übrigbleiben; diese werden nicht weiter zerteilt.

Auf diese Weise entsteht ein binärer Baum, der *2–d–Baum für die Punktmenge D* genannt wird. Jedem inneren Knoten des Baums entspricht eine Splitgerade.

Ein Beispiel ist in Abbildung 3.4 gezeigt. Die erste Splitgerade ist die Senkrechte $X = 5$. Die Punktmenge $D_{<5}$ links davon wird durch die waagerechte Gerade $Y = 4$ weiter zerlegt, in $D_{>5}$ wird dagegen $Y = 5$ als neue Splitgerade verwendet, usw.

[6] Wir bezeichnen aus traditionellen Gründen die Dimension mit k.

Wir können jedem Knoten v eines 2–d–Baums ein achsenparalleles Rechteck $R(v)$ in der Ebene zuordnen: der Wurzel die Ebene selbst, ihren Söhnen die beiden Halbebenen zu beiden Seiten der ersten Splitgeraden, und so fort. Allgemein entstehen die beiden Rechtecke der Söhne eines Knotens v durch Zerlegung von $R(v)$ mit der zu v gehörenden Splitgeraden.

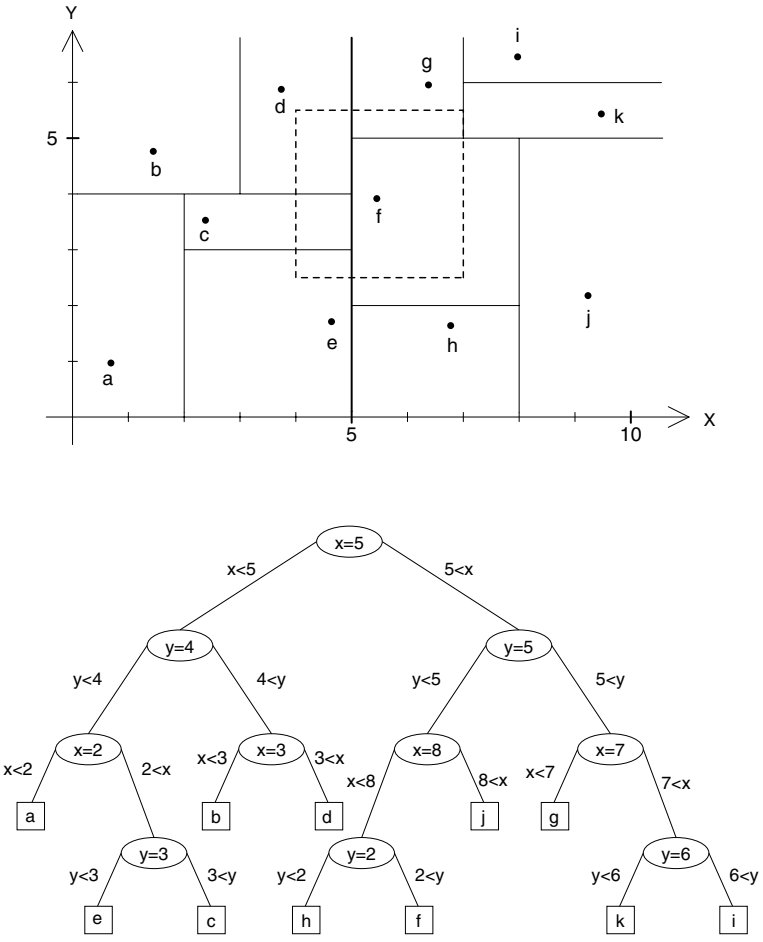


Abb. 3.4 Ein k–d–Baum für $k = 2$ und elf Punkte. In den Knoten sind die Splitgeraden angegeben; oben ist ein Beispiel für ein Anfragerechteck gestrichelt eingezeichnet.

In Abbildung 3.4 wurden die aus einem Knoten w herausführenden Kanten jeweils mit den Bezeichnungen der beiden Halbebenen beschriftet, die durch die Splitgerade von w definiert

werden. Das Rechteck $R(v)$ eines Knotens v ergibt sich als Durchschnitt der Halbebenen auf dem Pfad von der Wurzel zu v.

Die den Blättern eines 2–d–Baums zugeordneten Rechtecke bilden eine disjunkte Zerlegung der Ebene; jedes von ihnen enthält genau einen Punkt der Datenmenge D.

Suche nach einem Punkt

Die *Suche nach einem einzelnen Punkt* ist in einem 2–d–Baum sehr einfach: Man folgt immer der Kante, die mit der richtigen Halbebene beschriftet ist. Der Zeitaufwand ist proportional zur Höhe des Baumes.

Bereichsanfrage

Auch die Beantwortung einer *Bereichsanfrage* mit achsenparallelem Rechteck q ist einfach: Es genügt, alle Knoten v mit

$$R(v) \cap q \neq \emptyset$$

zu besuchen. Für jeden Blattknoten, der diese Bedingung erfüllt, muß getestet werden, ob sein Datenpunkt im Anfragerechteck q liegt. Falls ja, wird der Datenpunkt ausgegeben.

Wenn die Bedingung $R(v) \cap q \neq \emptyset$ für einen Knoten v erfüllt ist, so gilt sie erst recht für jeden Vorgänger u auf dem Pfad von der Wurzel zu v, denn das Rechteck $R(u)$ enthält $R(v)$.

Übungsaufgabe 3.6 Welche Knoten des in Abbildung 3.4 gezeigten 2–d–Baums müssen besucht werden, wenn das Anfragerechteck $q = [4, 7] \times [2.5, 5.5]$ vorliegt?

Die Antwort auf Übungsaufgabe 3.6 wirkt zunächst enttäuschend: Es sieht so aus, als müßten wir im schlimmsten Fall fast den ganzen Baum absuchen, auch wenn das Anfragerechteck nur wenige Datenpunkte enthält. Schauen wir genauer hin!

Lemma 3.6 *Sei T ein 2–d–Baum der Höhe h, in dem n Punkte gespeichert sind. Dann läßt sich jede Bereichsanfrage mit achsenparallelem Rechteck q in Zeit $O(2^{h/2} + a)$ beantworten, wobei a die Anzahl der in q enthaltenen Datenpunkte bezeichnet.*

Beweis. Es gibt zwei Typen von Knoten v in T, die die Bedingung $R(v) \cap q \neq \emptyset$ erfüllen und deshalb besucht werden müssen:

(1) solche Knoten v mit $R(v) \subseteq q$ und

(2) solche Knoten v mit $R(v) \not\subseteq q$ und $R(v) \cap q \neq \emptyset$.

Sobald wir auf unserem Weg in den Baum T hinein auf einen Knoten v vom *Typ (1)* treffen, können wir den an v hängenden Teilbaum $T(v)$ durchlaufen und alle Datenpunkte in den $a(v)$ vielen Blättern ausgeben; das geht in Zeit $O(a(v))$.

Ist v ein Knoten vom *Typ (2)*, gibt es zwei Fälle: Entweder liegt q ganz in $R(v)$; diese Knoten v liegen auf dem Anfangsstück

eines Pfades in T, ihre Anzahl beträgt daher höchstens h. Oder mindestens eine der vier Kanten – sagen wir l – des Anfragerechtecks q schneidet das Rechteck $R(v)$. Sei[7]

$t_k =$ Anzahl der Knoten v vom Typ (2) der Tiefe k in T, deren Rechteck $R(v)$ von der Kante l geschnitten wird.

Für die Wurzel von T und ihre beiden Söhne gilt offenbar $t_1 \leq 1$ und $t_2 \leq 2$. Allgemein gilt $t_{k+2} \leq 2t_k$, wie Abbildung 3.5 verdeutlicht. Hieraus folgt per Induktion

$$t_k \leq 2^{\frac{k}{2}} \quad \text{für } k = 1, 2, \ldots, h.$$

Für den Besuch aller Knoten vom Typ (2) ergibt sich daher eine obere Laufzeitschranke in

$$O\left(h + \sum_{k=1}^{h} 2^{\frac{k}{2}}\right) = O\left(2^{\frac{h}{2}}\right).$$

Weil jeder Knoten vom Typ (1) minimaler Tiefe einen Vater vom Typ (2) hat, ist hierdurch auch der Anmarsch zu den Typ (1)–Knoten abgedeckt, den wir bisher noch nicht betrachtet hatten. Damit ist Lemma 3.6 bewiesen. □

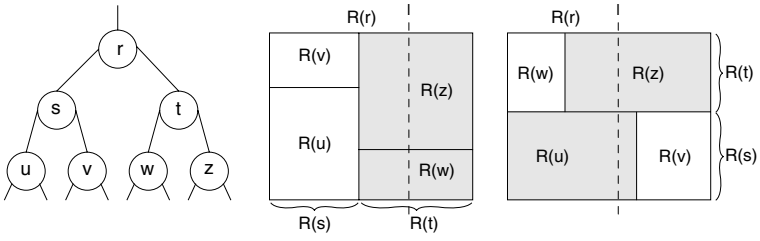


Abb. 3.5 Ein achsenparalleles Liniensegment l kann höchstens zwei der vier Enkel eines Rechtecks schneiden. Ganz links ist ein Ausschnitt von T dargestellt, daneben die beiden möglichen Unterteilungen des von l geschnittenen Rechtecks $R(r)$, je nachdem, ob die Splitgerade von r senkrecht oder waagerecht ist.

Spätestens jetzt stellt sich die Frage nach der *Höhe* eines 2–d–Baums zur Speicherung von n Punkten. Höhe

[7] Hierbei bezeichnet die *Tiefe eines Knotens* die Anzahl der Knoten auf dem Pfad von der Wurzel zu diesem Knoten plus eins, also die Gesamtzahl der zu verfolgenden Zeiger, wenn man einen Zeiger auf die Wurzel mitrechnet.

ausgeglichener Binärbaum

Wir nennen einen Binärbaum *ausgeglichen*, wenn für jeden inneren Knoten gilt: Die Blattzahlen seiner beiden Teilbäume unterscheiden sich höchstens um eins. Die Höhe eines ausgeglichenen Binärbaums mit n Blättern ist durch $\lceil \log n \rceil$ nach oben beschränkt. Lemma 3.6 zufolge liegt die Zeit für die Beantwortung einer Bereichsanfrage für einen ausgeglichenen 2–d–Baum also in $O(\sqrt{n} + a)$!

Aufbau ausgeglichener 2–d–Bäume

Zu einer gegebenen Menge D von n Punkten in der Ebene läßt sich ein ausgeglichener 2–d–Baum leicht konstruieren: Wir sortieren zunächst die Punkte nach ihren X- und Y-Koordinaten. Die beiden sortierten Listen werden dann dazu benutzt, die Punktmenge D rekursiv in gleich große Teilmengen zu zerlegen, wobei abwechselnd X und Y als Splitkoordinaten verwendet werden. Der Zeitaufwand für einen einzelnen Teilungsschritt ist linear in der Anzahl der verbleibenden Punkte; insgesamt ergibt sich ein Zeitaufwand in $O(n \log n)$.

Fassen wir zusammen:

statischer 2–d–Baum

Theorem 3.7 *Ein ausgeglichener 2–d–Baum für n Punkte in der Ebene läßt sich in Zeit $O(n \log n)$ konstruieren. Er belegt $O(n)$ viel Speicherplatz. Eine Bereichsanfrage mit achsenparallelem Rechteck läßt sich in Zeit $O(\sqrt{n} + a)$ beantworten; hierbei bezeichnet a die Größe der Antwort.*

Offenbar läßt sich ein Datenpunkt in Zeit $O(\log n)$ schwach entfernen, indem das entsprechende Blatt des 2–d–Baums als nicht belegt markiert wird. Aus Theorem 3.4 bzw. Theorem 3.5 über Dynamisierung folgt daher:

dynamisierter 2–d–Baum

Theorem 3.8 *Man kann bei dynamisierten 2–d–Bäumen in Zeit $O((\log n)^2)$ einen neuen Punkt einfügen und in Zeit $O(\log n)$ einen alten Punkt entfernen, wenn man bei achsenparallelen Bereichsanfragen eine gegenüber der statischen Struktur gestiegene Laufzeit von $O(\sqrt{n} \log n + a)$ in Kauf nimmt.*

Übungsaufgabe 3.7 Müßte in Theorem 3.8 die obere Schranke für die Laufzeit der Bereichsanfrage nicht korrekt $O((\sqrt{n} + a) \log n)$ lauten?

allgemeine Punktmengen

Bisher waren wir von der vereinfachenden Annahme ausgegangen, daß die Datenpunkte sich in ihren X- *und* Y-Koordinaten voneinander unterscheiden. Diese Einschränkung soll nun beseitigt werden. Das Problem, mit dem wir zu tun bekommen, wird in Abbildung 3.6 verdeutlicht: Es ist nicht möglich, *beliebige* Punktmengen mit einer achsenparallelen Schnittgeraden in zwei *gleich große* Teile zu zerlegen! Das war aber erforderlich, um einen ausgeglichenen Baum zu erhalten.

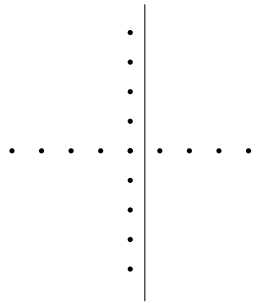


Abb. 3.6 Keine achsenparallele Splitgerade kann hier ein Größenverhältnis $\geq \frac{1}{3}$ zwischen der kleineren und der größeren Punktmenge erzielen.

Ob sich das Größenverhältnis der beiden Teilmengen beschränken läßt und welche Auswirkungen das auf die Effizienz des 2–d–Baums hat, sind interessante Fragen, denen wir in Übungsaufgabe 3.8 nachgehen. Wir lösen jetzt das Problem auf andere Weise: Bisher haben wir mit Schnittgeraden gearbeitet, die keinen der Datenpunkte enthalten. Diese Bedingung geben wir auf. Zur Speicherung der Punkte *auf* der Schnittgeraden wird ein ausgeglichener 1–d–Baum verwendet – ein ganz normaler eindimensionaler Suchbaum! Ein Beispiel für die auf diese Weise *erweiterten 2–d–Bäume* gibt Abbildung 3.7.

erweiterter 2–d–Baum

Wie man sieht, ist unser 2–d–Baum *ternär* geworden: Unterhalb eines jeden inneren Knotens ist der eindimensionale Suchbaum für die Punkte auf der Splitgeraden hinzugekommen.

Gelten unsere Ergebnisse auch für erweiterte 2–d–Bäume? Einem inneren Knoten v in einem eindimensionalen Teilbaum T_1 ist kein Rechteck $R(v)$, sondern ein Intervall $I(v)$ zugeordnet, das auf der Splitgeraden des Wurzelknotens w von T_1 liegt. Ist $R(w) \cap q \neq \emptyset$, müssen wir alle Knoten v in T_1 mit $I(v) \cap q \neq \emptyset$ besuchen. Für diejenigen Knoten v mit $I(v) \subseteq q$ müssen wir den gesamten an v hängenden Teilbaum besuchen und alle Punkte in dessen Blättern ausgeben; der Gesamtaufwand hierfür wird durch die Größe der Antwort abgeschätzt. Gilt dagegen

$$I(v) \not\subseteq q \text{ und } I(v) \cap q \neq \emptyset,$$

so muß das Intervall $I(v)$ eine Kante l des Anfragerechtecks q schneiden, ohne in l enthalten zu sein. Hierfür bestehen zwei Möglichkeiten: l schneidet $I(v)$ orthogonal, oder l und $I(v)$ liegen auf derselben Geraden, und $I(v)$ enthält einen bestimmten Endpunkt von l. In beiden Fällen kann höchstens ein Sohn von v diese Eigenschaft erben. Insgesamt gibt es daher im eindimensio-

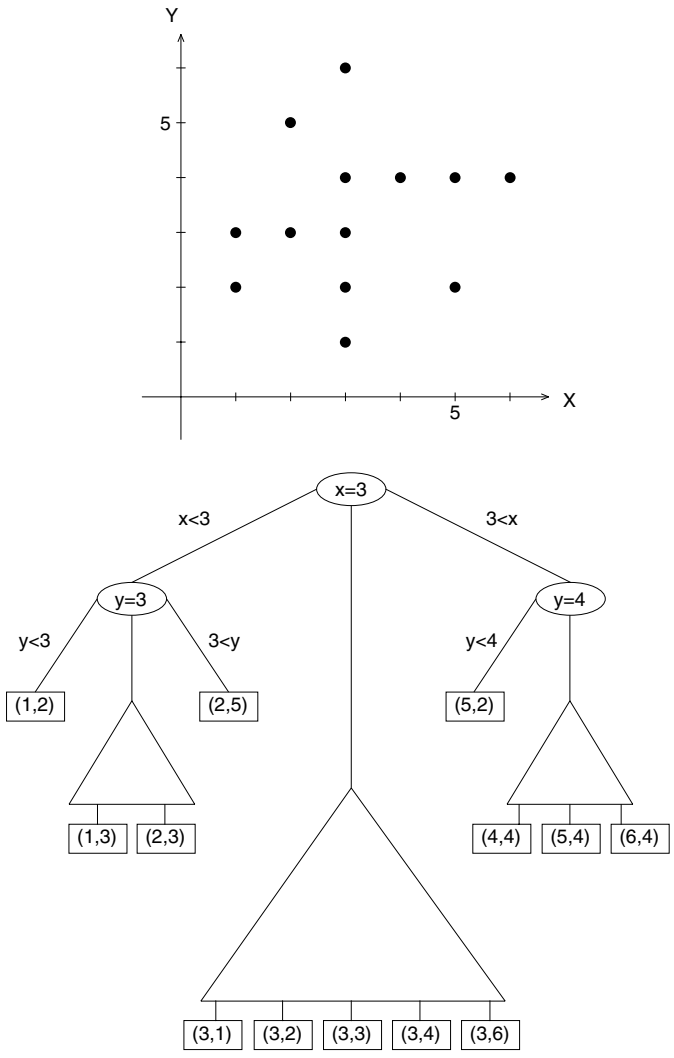


Abb. 3.7 Ein erweiterter 2–d–Baum für eine Menge von Punkten, deren Koordinaten nicht paarweise verschieden sind.

nalen Teilbaum T_1 nur $O(h_1)$ viele solche Knoten, und sie liegen auf ihren Pfaden ohne Lücken.

ausgeglichen

Ein erweiterter 2–d–Baum heißt *ausgeglichen*, wenn für jeden inneren Knoten v die Teilbäume seiner beiden äußeren Söhne jeweils höchstens halb so viele Punkte enthalten wie der gesamte Teilbaum von v. Diese Bedingung läßt sich leicht einhalten, da ja auch der mittlere Sohn Datenpunkte enthalten darf.

Betrachten wir einen ausgeglichenen 2–d–Baum und darin einen eindimensionalen Teilbaum T_1, dessen Wurzel w die Tiefe k besitzt. Insgesamt kann w maximal $\frac{n}{2^{k-1}}$ viele Blätter unter sich haben; insbesondere ist deshalb die Höhe von T_1 durch $\log \frac{n}{2^{k-1}} = \log n - k + 1$ beschränkt. Die Gesamthöhe h eines erweiterten 2–d–Baums einschließlich aller eindimensionalen Teilbäume ist also wiederum logarithmisch beschränkt.

Weil bei einer Bereichsanfrage nach wie vor höchstens $2^{\frac{k}{2}}$ viele derartige Knoten w der Tiefe k zu inspizieren sind, liegt der zusätzliche Aufwand für das Besuchen aller eindimensionalen Teilbäume daher in[8]

$$O\left(\sum_{k=1}^{h} 2^{\frac{k}{2}} \cdot \log(2^{h-k+1})\right) = O\left(2^{\frac{h}{2}} \sum_{k=1}^{h} \frac{h-k+1}{2^{\frac{h-k}{2}}}\right) = O\left(2^{\frac{h}{2}}\right).$$

Folglich gelten Theorem 3.7 und Theorem 3.8 auch für erweiterte 2–d–Bäume.

Anwendung: kleinster Abstand zwischen Punkten im $\mathbb{R}^3$

Als Anwendung betrachten wir eine Lösung des in Abschnitt 2.4.1 von Kapitel 2 vorgestellten Problems, in einer Menge von n Punkten im $\mathbb{R}^3$ den kleinsten Abstand zu bestimmen.

Wir waren das Problem mit einer *sweep plane* angegangen, die sich längs einer Koordinatenachse durch den Raum bewegt. In der Sweep-Status-Struktur müssen alle Punkte gespeichert werden, die in einem Streifen bestimmter Breite hinter der *sweep plane* liegen; jeder der n Punkte wird im Laufe des *sweeps* genau einmal in die *SSS* eingefügt und höchstens einmal daraus entfernt. Außerdem sind maximal n rechteckige Bereichsanfragen zu beantworten, bei denen die Größe der Antwort durch eine Konstante beschränkt ist.

Wenn wir hierfür einen 2–d–Baum verwenden, können wir nach Theorem 3.8 in Zeit $e(n) \in O((\log n)^2)$ Punkte einfügen und entfernen und benötigen $b(n) \in O(\sqrt{n} \log n)$ Zeit für eine Bereichsanfrage. Daraus ergibt sich folgende Verbesserung der Laufzeit $\Theta(n^2)$ des naiven Verfahrens:

Korollar 3.9 *Der kleinste Abstand zwischen n Punkten im $\mathbb{R}^3$ läßt sich in Zeit $O(n^{\frac{3}{2}} \log n)$ und linearem Speicherplatz bestimmen.*

Amortisierung genügt

Um Korollar 3.9 zu erhalten, genügt übrigens Theorem 3.4. Denn wir betrachten ja hier eine *Folge* von n Einfüge- und Entferne-Operationen bei anfangs leerer Struktur und interessieren uns für die Gesamtkosten, die durch das n-fache der mittleren Kosten beschränkt sind.

[8] Für $|\alpha| < 1$ hat die Reihe $\sum_{i=0}^{\infty} (i+1)\alpha^i$ einen endlichen Wert; Beweis durch Ableitung beider Seiten von $\sum_{i=0}^{\infty} X^i = \frac{1}{1-X}$.

höhere Dimensionen

Das Konzept der 2–d–Bäume läßt sich direkt auf höhere Dimensionen übertragen. In einem k–d–Baum werden die k Koordinaten zyklisch abwechselnd als Splitkoordinaten verwendet.

Splithyperebenen

An die Stelle der Splitgeraden treten *Splithyperebenen* $X_i = a$ des $\mathbb{R}^k$. Alle Datenpunkte, die in einer Splithyperebene liegen, werden in einem $(k-1)$–d–Baum gespeichert.

Die Analyse von k–d–Bäumen enthält keine wesentlichen neuen Ideen, ist aber technisch recht kompliziert. Wer sich dafür interessiert, sei auf Lee und Wong [92] verwiesen.

Wir zitieren ohne Beweis die Verallgemeinerung von Theorem 3.7 auf Dimensionen $k > 2$.

Theorem 3.10 *Ein ausgeglichener k–d–Baum zur Speicherung von n Punkten des $\mathbb{R}^k$ hat eine durch $O(\log n)$ beschränkte Höhe. Er läßt sich in Zeit $O(n \log n)$ konstruieren und benötigt $O(n)$ viel Speicherplatz. Eine orthogonale Bereichsanfrage läßt sich in Zeit $O\left(n^{1-\frac{1}{k}} + a\right)$ beantworten; dabei bezeichnet a die Größe der Anwort.*

Bei diesen Aussagen hängen die Konstanten in der O-Notation von der Dimension k ab.

Eine Stärke des k–d–Baums liegt im geringen Platzbedarf; man kann zeigen, daß für Bereichsanfragen $\Omega(n^{1-\frac{1}{k}})$ eine untere Schranke darstellt, wenn man mit linearem Speicherplatz auskommen muß; vgl. Mehlhorn [103]. Insofern ist der k–d–Baum optimal.

Stellt man jedoch für die Speicherung von n Punkten im $\mathbb{R}^k$ mehr als $O(n)$ Speicherplatz zur Verfügung, läßt sich der Zeitbedarf für Bereichsanfragen weiter reduzieren. Hierauf gehen wir in Abschnitt 3.3.2 ein.

Zum Abschluß dieses Abschnitts gehen wir der Frage nach, wie sich überhaupt Punktmengen im $\mathbb{R}^k$ durch orthogonale Hyperebenen in möglichst gleich große Teile zerlegen lassen.

Zerlegung von Punktmengen

Übungsaufgabe 3.8

(i) Sei D eine Menge von n Punkten im $\mathbb{R}^2$. Man zeige, daß es eine achsenparallele Gerade gibt, die selbst keinen Punkt aus D enthält und die Menge D so zerlegt, daß der kleinere Teil mindestens $\frac{n-1}{4}$ viele Punkte enthält. Ist diese Schranke scharf?

(ii) Könnte man auf die Erweiterung von 2–d–Bäumen durch eindimensionale Suchbäume für die Punkte auf Splitgeraden verzichten und statt dessen die Splitgeraden gemäß (i) wählen, ohne eine Laufzeitverlängerung hinzunehmen?

(iii) Man verallgemeinere (i) auf Punktmengen im $\mathbb{R}^k$.

3.3.2 Der Bereichsbaum

In diesem Abschnitt betrachten wir eine Datenstruktur zur Speicherung einer Menge D von n Datenpunkten des $\mathbb{R}^k$, die orthogonale Bereichsanfragen besser unterstützt. So wird es (bei der statischen Version) in der Ebene möglich, eine Anfrage mit achsenparallelem Rechteck in Zeit $O\left((\log n)^2 + a\right)$ zu beantworten; allerdings wird dazu $O(n \log n)$ viel Speicherplatz benötigt. Diese Datenstruktur heißt *Bereichsbaum* (englisch: *range tree*).

Bereichsbaum

Ein *eindimensionaler* Bereichsbaum T^1 für n Zahlen auf der X-Achse ist ein gewöhnlicher binärer Suchbaum. Wir verlangen, daß er *ausgeglichen* ist, sich also die Blätterzahlen der beiden Teilbäume eines jeden Knotens um höchstens *eins* unterscheiden. Abbildung 3.8 zeigt ein Beispiel. Die Zahlen sind in den Blättern gespeichert.

Dimension 1

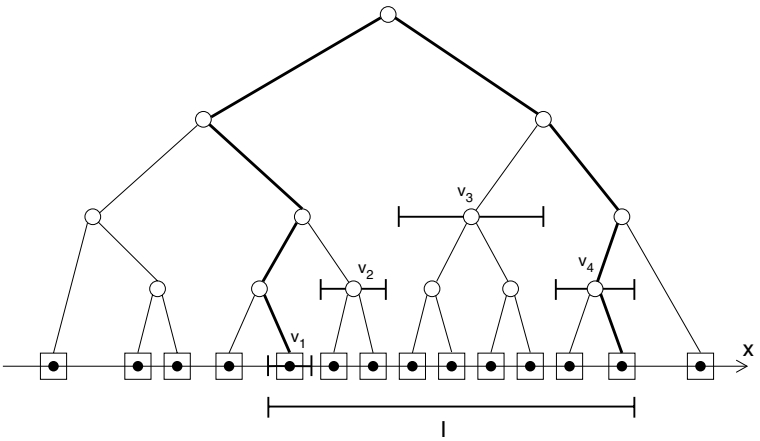


Abb. 3.8 Ein eindimensionaler Bereichsbaum T^1 und ein Anfrageintervall I.

Jedem Knoten v von T^1 ist ein Intervall $I(v)$ zugeordnet, das sich von der kleinsten bis zur größten Zahl erstreckt, die im Teilbaum von v gespeichert ist. Das Intervall eines Blattknotens enthält nur die dort gespeicherte Zahl. Um die Beantwortung von Bereichsabfragen zu erleichtern, werden bei jedem Knoten v die Intervalle $I(w_1)$ und $I(w_2)$ seiner beiden Söhne w_1 und w_2 gespeichert.

Intervall $I(v)$

Zur Beantwortung einer Bereichsanfrage mit Intervall I muß man die anhängenden Teilbäume der am weitesten oben gelegenen Knoten v in T^1 besuchen, für die $I(v) \subseteq I$ gilt. Im Beispiel von Abbildung 3.8 sind das gerade die Knoten v_1, v_2, v_3 und v_4. Ihre

Intervalle schöpfen das Anfrageintervall I aus: Zusammen enthalten sie alle gespeicherten Zahlen, die auch in I enthalten sind.

Übungsaufgabe 3.9 Seien $v_1, \ldots, v_l$ die Knoten minimaler Tiefe in einem eindimensionalen Bereichsbaum T^1, deren Intervall $I(v_j)$ vollständig im Anfrageintervall I enthalten ist. Man zeige:

(i) Die Vereinigung der Intervalle $I(v_j)$ enthält genau dieselben Daten wie I.

(ii) Die Intervalle $I(v_j)$ sind paarweise disjunkt.

(iii) Man kann in Zeit $O(\log n)$ die Knoten $v_1, \ldots, v_l$ von T^1 besuchen und die Bereichsanfrage für I in Zeit $O(\log n + a)$ beantworten.

Man sieht, daß es nicht unbedingt notwendig ist, die Blätter eines Suchbaums zu einer Liste zu verketten, um eindimensionale Bereichsanfragen effizient beantworten zu können.

k-dimensionaler Bereichsbaum

Wir wenden uns nun dem höherdimensionalen Fall zu. Ein *k-dimensionaler Bereichsbaum* T^k für D wird rekursiv folgendermaßen definiert: Man baut einen eindimensionalen Bereichsbaum T^1 für die Menge

$$D^1 = \{x_1; \exists x_2, \ldots, x_k : (x_1, x_2, \ldots, x_k) \in D\}$$

der X_1-Koordinaten aller Datenpunkte.[9] Dann konstruiert man für jeden Knoten v von T^1 einen $(k-1)$-dimensionalen Bereichsbaum T_v^{k-1} für die Menge

$$D_v^{k-1} = \{(x_2, \ldots, x_k); \exists x_1 \in I(v) : (x_1, x_2, \ldots, x_k) \in D\},$$

die sich ergibt, wenn wir alle Datenpunkte im Intervall $I(v)$ des Knotens v hernehmen und ihre X_1-Koordinaten abschneiden. Am Knoten v von T^1 wird ein Zeiger angebracht, der auf den Baum T_v^{k-1} verweist.

Für den Fall, daß $k = 2$ war und unsere rekursive Definition mit der Konstruktion der Bäume $T_v^{k-1} = T_v^1$ zum Ende kommt, fügen wir jedem Blatt eines Baums T_v^1 eine Liste derjenigen Punkte aus D bei, für die die Suche in eben diesem Blatt endet. Abbildung 3.9 zeigt ein Beispiel für den Fall $k = 2$.

[9] Daß verschiedene Punkte in D dieselbe X_1-Koordinate haben können, stört dabei nicht.

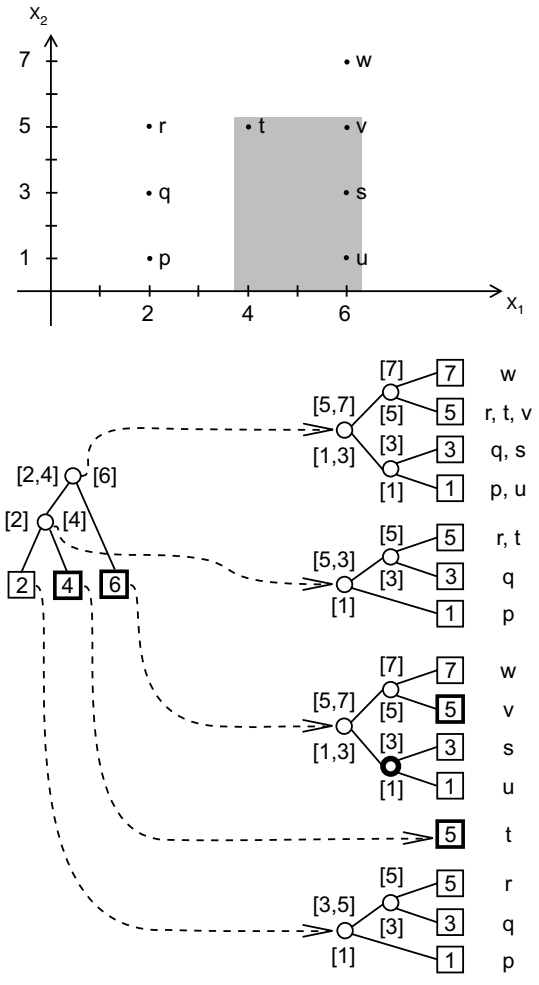


Abb. 3.9 Ein zweidimensionaler Bereichsbaum. Die Knotenintervalle der Söhne sind mit eckigen Klammern zu beiden Seiten der Väter notiert. Die Anfrage nach allen Punkten im eingezeichneten Bereich führt zu einem Besuch der hervorgehobenen Knoten minimaler Tiefe und den daran hängenden Blättern.

Bereichsanfrage

Eine Bereichsanfrage mit orthogonalem Hyperrechteck (oder: Hyperquader) $q \subset \mathbb{R}^k$ für die Daten in T^k wird nun folgendermaßen beantwortet:

(0) Falls $k = 1$ ist, beantworte die Anfrage; sonst

(1) stelle q in der Form $q = I \times q'$ dar, wobei I ein Intervall und q' ein $(k-1)$-dimensionales Hyperrechteck ist;

(2) bestimme die Knoten $v_1, \ldots, v_l$ in T^1, deren Intervalle $I(v_i)$ zusammen das Intervall I ausschöpfen;

(3) für $1 \leq i \leq l$: beantworte die Bereichsanfrage q' für die Datenstruktur $T^{k-1}_{v_i}$.

Die Korrektheit dieser rekursiven Prozedur liegt auf der Hand: Alle in einem Baum $T^{k-1}_{v_1}$ gespeicherten Punkte haben eine X_1-Koordinate in $I(v_1) \subseteq I$; man muß also nur noch die anderen Koordinaten inspizieren um festzustellen, ob solch ein Punkt zu q gehört. Dies geschieht rekursiv, bis nur noch eine eindimensionale Anfrage übrigbleibt.

Nicht auf den ersten Blick klar ist, wieviel Speicherplatz ein Bereichsbaum benötigt und wie lange ein *range query* dauert. Diesen Fragen wollen wir nun nachgehen.

statischer Bereichsbaum

Theorem 3.11 *Sei $k \geq 2$. Ein k-dimensionaler Bereichsbaum zur Speicherung von n Punkten im $\mathbb{R}^k$ benötigt $O(n(\log n)^{k-1})$ viel Speicherplatz und kann in Zeit $O(n(\log n)^{k-1})$ aufgebaut werden. Eine orthogonale Bereichsanfrage läßt sich damit in Zeit $O((\log n)^k + a)$ beantworten.*

Beweis. Wir dürfen annehmen, daß die Punkte aus D in *allen* ihren Koordinaten paarweise verschiedene Werte haben (andernfalls wird das Verhalten des Bereichsbaums höchstens besser). Dann brauchen wir zwischen den Punkten und Teil-Tupeln ihrer Koordinaten nicht zu unterscheiden.

Die Struktur T^k besteht insgesamt aus lauter eindimensionalen Bereichsbäumen T^1. Wir sagen, ein Datenpunkt *kommt in T^1 vor*, wenn eine seiner Koordinaten in einem Blatt von T^1 gespeichert ist. Im ersten Bereichsbaum T^1 von T^k kommt jeder Datenpunkt p vor. Seine X_1-Koordinate ist in $\log n$ vielen Intervallen $I(v)$ enthalten. Jeder Baum T^{k-1}_v beginnt wiederum mit einem eindimensionalen Baum, in dem p vorkommt. Hier ist die X_2-Koordinate in $\log n$ vielen Intervallen enthalten, und so fort.

Insgesamt kommt jeder der n Datenpunkte in

$$\sum_{i=0}^{k-1} (\log n)^i = \frac{(\log n)^k - 1}{\log n - 1} \in \Theta((\log n)^{k-1})$$

vielen eindimensionalen Bäumen vor. Folglich wird $O(n(\log n)^{k-1})$ viel Speicherplatz benötigt. Speicherplatzbedarf

Zur Konstruktion eines k-dimensionalen Bereichsbaums werden zunächst die folgenden Listen L_j, $1 \leq j \leq k$, generiert: L_j enthält die Koordinatenvektoren $(x_1, \ldots, x_j)$ der Punkte in D und ist nach der letzten Koordinate X_j sortiert. Dies kann in Zeit $O(n \log n)$ geschehen, wobei die Konstante in der O-Notation von der Dimension k abhängt. Mit Hilfe der Listen L_1 läßt sich der erste Baum T^1 von T^k in Zeit $O(n)$ aufbauen. Er hat $2n-1$ viele Knoten v_i, für die jeweils ein $(k-1)$-dimensionaler Bereichsbaum $T_i = T^{k-1}_{v_i}$ zu konstruieren ist. Um ein erneutes Sortieren der Punkte zu vermeiden, bedienen wir uns der Listen L_2 und $L_3, \ldots, L_k$ in folgender Weise: An der Wurzel von T^1 wird von jeder Liste eine Kopie hinterlegt. Dann wird jeweils eine weitere Kopie sequentiell durchlaufen und entsprechend der X_1-Koordinaten auf die Söhne der Wurzel verteilt. Dieser Vorgang wiederholt sich, bis schließlich jeder Knoten v_i in T^1 über Listen $L_{2,i}, \ldots, L_{k,i}$ verfügt, die genau diejenigen Koordinatenvektoren enthalten, deren X_1-Werte in seinem Intervall $I(v_i)$ liegen. Dann werden bei allen Listen außer bei L_k die X_1-Koordinaten entfernt, weil sie für den Bau der Bäume T_i nicht mehr benötigt werden. Am Ende der rekursiven Konstruktion wird die Liste L_k dazu verwendet, an den Blättern der eindimensionalen Bäume der letzten Koordinate die Listen mit den zugehörigen Punkten anzubringen, siehe Abbildung 3.9. Der Zeit– und Speicherplatzbedarf für diesen Aufteilungsschritt wird von den übrigen Kosten dominiert. Aufbau

Im Baum T_i seien n_i viele Punkte gespeichert; dann dürfen wir per Induktion annehmen, daß sich T_i in Zeit $\leq C \cdot n_i(\log n_i)^{k-2}$ aufbauen läßt.

Weil jeder der n Datenpunkte in nur $\log n$ vielen Bäumen T_i vorkommt, gilt

$$\sum_{i=1}^{2n-1} n_i \leq n \log n.$$

Also dauert der Aufbau aller T_i zusammen nicht länger als

$$\begin{aligned} C \sum_{i=1}^{2n-1} n_i(\log n_i)^{k-2} &\leq C(\log n)^{k-2} \sum_{i=1}^{2n-1} n_i \\ &\leq Cn(\log n)^{k-1}. \end{aligned}$$

Zur Beantwortung einer *Bereichsanfrage* werden in Schritt (2) die $2 \log n$ vielen Knoten v_j in T^1 bestimmt, deren Intervalle das Anfrageintervall I ausschöpfen. Für jeden Baum $T^{k-1}_{v_j}$ genügt nach Induktionsvoraussetzung $C((\log n_j)^{k-1} + a_j)$ Zeit, um die Bereichsanfrage

a_j vielen Datenpunkte aus I zu berichten, die dort gespeichert sind. Insgesamt ergibt sich die Laufzeitschranke

$$C \log n + C \sum_{j=1}^{2\lceil \log n \rceil} \left((\log n_j)^{k-1} + a_j \right) \in O((\log n)^k + a). \qquad \square$$

Um alle $O((\log n)^{k-1})$ Vorkommen eines Datenpunktes p in einem k-dimensionalen Bereichsbaum zu finden und p *schwach zu entfernen*, wird $O((\log n)^k)$ viel Zeit benötigt. Also folgt aus Theorem 3.5:

dynamisierter Bereichsbaum

Theorem 3.12 *Man kann k-dimensionale Bereichsbäume so dynamisieren, daß sich in Zeit $O((\log n)^k)$ Punkte einfügen und entfernen lassen, wenn man eine Verschlechterung der Laufzeit für achsenparallele Bereichsanfragen auf $O((\log n)^{k+1} + a)$ in Kauf nimmt.*

kleinster Abstand im $\mathbb{R}^3$

Für unser altes Problem, den kleinsten Abstand zwischen n Punkten im $\mathbb{R}^3$ zu bestimmen, erhalten wir bei Verwendung dynamisierter zweidimensionaler Bereichsbäume für die Sweep-Status-Struktur folgende Lösung:

Korollar 3.13 *Der kleinste Abstand zwischen n Punkten im $\mathbb{R}^3$ läßt sich in Zeit $O(n(\log n)^3)$ und Speicherplatz $O(n \log n)$ bestimmen.*

noch einmal: *sweep*

Dieses Ergebnis – ebenso wie Korollar 3.9 – bleibt ein wenig hinter unseren Erwartungen zurück, denn die untere Schranke für die Laufzeit beträgt nach Korollar 1.10 ja „nur “ $\Omega(n \log n)$! Hier zeigen sich die Grenzen des Sweep-Verfahrens: In der Ebene funktioniert es vorzüglich, weil sich die Sweep-Status-Struktur mit einem eindimensionalen balancierten Baum implementieren läßt, der Zugriffe in Zeit $O(\log n)$ ermöglicht. Beim *sweep* im $\mathbb{R}^3$ benötigen wir eine zweidimensionale Struktur und haben Zugriffskosten in $O((\log n)^2)$.[10] Die Argumentation bei Mehlhorn [103] zeigt, daß dies prinzipielle Ursachen hat.

Tatsächlich läßt sich das Problem des minimalen Abstands bzw. des dichtesten Paars in höheren Dimensionen effizienter lösen als mit *sweep*; vgl. Bentley und Shamos [17] oder Schwarz et al. [129].

[10] Der dritte Faktor $\log n$ in Korollar 3.13 geht auf die erhöhten Kosten der Bereichsanfragen infolge von Dynamisierung zurück; er ließe sich durch direkte Balancierung der Bereichsbäume vermeiden.

3.3.3 Der Prioritätssuchbaum

Zum Abschluß betrachten wir eine Datenstruktur, die ausschließlich zur Speicherung von Punktmengen in der Ebene dient. Sie stellt eine sinnvolle Kombination aus eindimensionalem Bereichsbaum und *heap* dar und wird als *Prioritätssuchbaum* (englisch: *priority search tree*) bezeichnet.

Prioritätssuchbaum

Gegeben sei eine Menge D von n Punkten $p_i = (x_i, y_i)$ in der Ebene. Wir nehmen an, daß ihre X-Koordinaten paarweise verschieden sind.

vereinfachende Annahme

Zunächst wird ein eindimensionaler Bereichsbaum[11] für die Menge der in den Punkten in D vorkommenden X-Koordinaten aufgebaut. In jedem Knoten des Baums wird Platz für einen Datenpunkt reserviert.

Aufbau

In dieses anfangs leere Skelett werden die Punkte $p \in D$ so eingetragen, daß folgende Bedingungen erfüllt sind:

(1) Jeder Punkt $p = (x, y)$ steht auf dem Suchpfad zu seiner X-Koordinate x;

(2) längs eines Pfades von der Wurzel zu einem Blatt nehmen die Y-Koordinaten der Punkte monoton zu (Heap-Eigenschaft);

(3) die Punkte stehen so dicht bei der Wurzel wie möglich.

Ein Beispiel für einen Prioritätssuchbaum ist in Abbildung 3.10 gezeigt. Wir machen uns zunächst klar, warum diese Forderungen in jedem Fall erfüllbar sind:

Übungsaufgabe 3.10 Man zeige, daß im leeren Skelett eines Prioritätssuchbaums genügend Platz ist, um die Punkte so unterzubringen, wie die Bedingungen (1) bis (3) es verlangen. Finden alle Punkte in den inneren Knoten Platz?

Um nun das leere Skelett zu füllen, können wir folgendermaßen vorgehen: Zunächst sortieren wir die Datenpunkte nach aufsteigenden Y-Koordinaten. Dann fügen wir sie in dieser Reihenfolge ein. Beim Einfügen von $p = (x, y)$ laufen wir den Suchpfad zum Blatt x hinab, bis wir auf einen freien Knoten treffen, und legen p dort ab. Übungsaufgabe 3.10 garantiert, daß stets ein freier Knoten angetroffen wird. Offenbar läßt sich der gesamte Aufbau in Zeit $O(n \log n)$ erledigen.

[11] Also ein ausgeglichener binärer Suchbaum, bei dem die X-Koordinaten in den Blättern gespeichert sind.

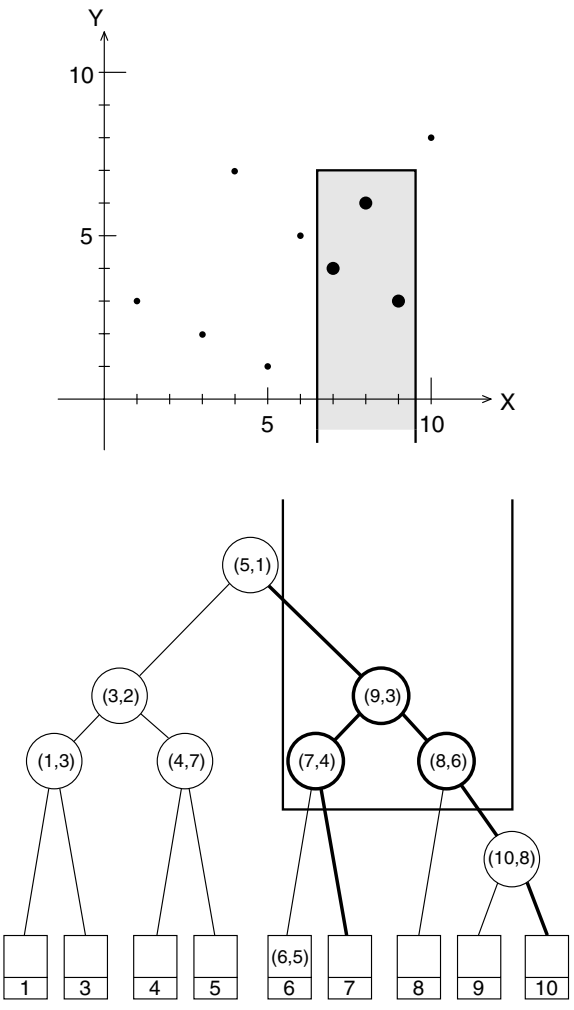


Abb. 3.10 Ein Prioritätssuchbaum für 9 Punkte in der Ebene und eine Anfrage mit dem Halbstreifen $[6.5, 9.5] \times (-\infty, 7]$.

Anfrage

Was läßt sich mit einem Prioritätssuchbaum anfangen? Zunächst kann man eine eindimensionale *Anfrage* nach allen Punkten mit X-Koordinate in einem Intervall I damit beantworten, denn ein Prioritätssuchbaum ist ja insbesondere auch ein eindimensionaler Bereichsbaum für die X-Koordinaten.

Wenn wir die in Abbildung 3.8 gezeigten Suchpfade nach den Intervallgrenzen von I betrachten, sind die Punkte mit X-Koordinaten in I jedenfalls in solchen Knoten des Prioritätssuchbaums gespeichert, die auf oder zwischen den Suchpfaden liegen.

Um die gesuchten Punkte zu finden, müssen wir also von der Wurzel aus in diesen Teil des Baums hinabsteigen. Dabei werden die Y-Koordinaten der Punkte größer, je tiefer wir kommen.

Angenommen, wir wären nur an denjenigen Punkten (x, y) aus D interessiert, für die $x \in I$ und $y \leq y_0$ gilt, also an der Menge

$$D \cap (I \times (-\infty, y_0]).$$

Dann können wir beim Hinabsteigen in den Baum einfach anhalten und umkehren, sobald die Y-Werte größer als y_0 werden, und auf diese Weise den Besuch von Knoten vermeiden, deren Punkte nicht zum Anfragebereich gehören.

Man sieht: Der Prioritätssuchbaum unterstützt orthogonale Bereichsanfragen für Rechtecke, bei denen eine Kante (hier: die untere) im Unendlichen liegt. Wir nennen eine solche Anfrage eine *Halbstreifen-Anfrage*. Ein Beispiel ist in Abbildung 3.10 dargestellt. Die Suchpfade nach den Grenzen des X-Intervalls sind fett hervorgehoben. Genau die Punkte in den fett umrandeten Knoten sind zu berichten. Natürlich müssen auch deren Söhne besucht werden, um festzustellen, daß die darin enthaltenen Punkte zu große Y-Koordinaten haben, aber ein Besuch bei den Enkeln ist überflüssig.

Halbstreifen-Anfrage

Theorem 3.14 *Ein Prioritätssuchbaum für n Punkte der Ebene kann in Zeit $O(n \log n)$ aufgebaut werden. Er benötigt $O(n)$ viel Speicherplatz und gestattet es, eine Halbstreifen-Anfrage in Zeit $O(\log n + a)$ zu beantworten.*

statischer Prioritätssuchbaum

Für rechteckige Bereichsanfragen liefert ein einzelner Prioritätssuchbaum keine guten Resultate; wir werden aber nachher sehen, was sich mit der Kombination mehrerer Prioritätssuchbäume erreichen läßt.

Übungsaufgabe 3.11 Warum ist der Summand $\log n$ in der Schranke für die Anfragezeit in Theorem 3.14 erforderlich?

Weil ein Punkt offenbar in Zeit $O(\log n)$ schwach entfernt werden kann, folgt aus Theorem 3.5 über Dynamisierung:

Theorem 3.15 *Man kann bei einem dynamisierten Prioritätssuchbaum für n Punkte in Zeit $O((\log n)^2)$ einen Punkt einfügen und in Zeit $O(\log n)$ entfernen, wenn man eine Laufzeit von $O((\log n)^2 + a)$ für Halbstreifen-Bereichsanfragen in Kauf nimmt.*

dynamisierter Prioritätssuchbaum

dynamischer Prioritätssuchbaum

Wenn man „maßgeschneiderte“ Techniken für das Einfügen und Entfernen verwendet, lassen sich alle Operationen in Zeit $O(\log n)$ bzw. $O(\log n + a)$ ausführen; siehe Ottmann und Widmayer [116]. Im folgenden legen wir diese „schnelle“ Dynamisierung zugrunde.

allgemeine Punktmenge

Von unserer anfänglichen Annahme, daß die Datenpunkte paarweise verschiedene X-Koordinaten haben müssen, machen wir uns jetzt frei. Statt (x, y) betrachten wir $((x, y), y)$ als zweidimensionales Datenobjekt und verwenden die lexikographische Ordnung für die erste Komponente.

Anwendung: Intervallüberlappung

Prioritätssuchbäume haben interessante Anwendungen. Betrachten wir zum Beispiel das Problem, eine Menge von Intervallen auf der X-Achse so zu speichern, daß schnell ermittelt werden kann,

(1) welche Intervalle von einem X-Anfragewert aufgespießt werden oder

(2) welche Intervalle ein Anfrageintervall überlappen;

offenbar ist (1) ein Spezialfall von (2).

Effiziente Lösungen für dieses Problem ergeben sich durch die Verwendung der speziellen Datenstrukturen *Segment-Baum* oder *Intervall-Baum*, die z. B. bei Güting [69] erläutert werden.

Wir wollen nun zeigen, daß sich auch der Prioritätssuchbaum hierfür verwenden läßt. Hilfreich ist folgende Beobachtung, die durch Abbildung 3.11 illustriert wird:

Lemma 3.16 *Seien $I_0 = [x_0, y_0]$ und $I = [x, y]$ zwei Intervalle auf der X-Achse. Dann gilt:*

$$\begin{aligned} I_0 \text{ überlappt } I &\iff x \le y_0 \text{ und } x_0 \le y \\ &\iff (y_0, x_0) \text{ liegt rechts unterhalb von } (x, y). \end{aligned}$$

Während der Beweis dieses Sachverhalts trivial ist, sind seine Konsequenzen recht nützlich: Anstatt alle gespeicherten Intervalle $[x_0, y_0]$ zu bestimmen, die das Anfrageintervall $[x, y]$ überlappen, brauchen wir nur die Punkte (y_0, x_0) zu finden, die in der Ebene rechts unterhalb von (x, y) liegen. Und hierfür läßt sich der Prioritätssuchbaum bestens verwenden, denn die Viertelebene $[x, \infty) \times (-\infty, y]$ ist ein nach rechts unbeschränkter Halbstreifen!

geometrische Transformation

Als Folge dieser geschickt angewandten *geometrischen Transformation* erhalten wir:

Theorem 3.17 *Man kann n Intervalle in linearem Platz so speichern, daß sich Überlappungsanfragen in Zeit $O(\log n + a)$ beantworten lassen. Es ist möglich, ein Intervall in Zeit $O(\log n)$ einzufügen oder zu entfernen.*

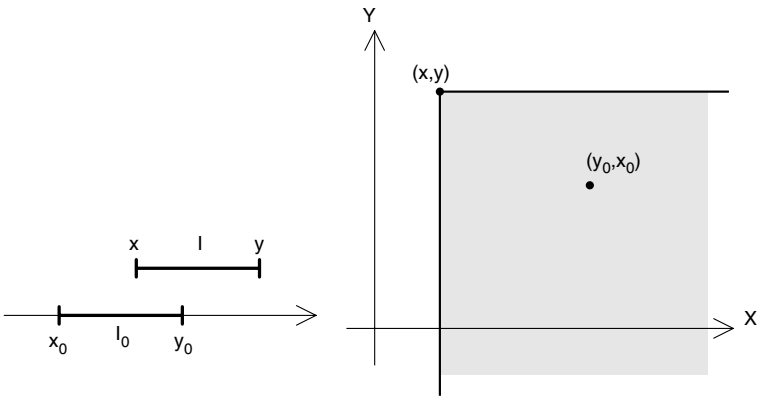


Abb. 3.11 Das Problem Intervallüberlappung wird in eine Bereichsanfrage transformiert.

Eigentlich ist eine Bereichsanfrage mit einem Halbstreifen von einer Anfrage mit einem „echten“ Rechteck nicht so sehr verschieden; man könnte auf die Idee kommen, das Rechteck aus zwei Halbstreifen zusammenzusetzen, wie Abbildung 3.12 zeigt. Natürlich führt dieser Ansatz *nicht* zu einer Output-sensitiven Lösung des Anfrageproblems für Rechtecke, denn jeder der beiden Halbstreifen könnte sehr viele Datenpunkte enthalten, ihr Durchschnitt aber nur wenige.

Trotzdem hat diese Idee einen guten Kern.

Theorem 3.18 *Sei $h > 0$ fest vorgegeben. Dann lassen sich n Punkte in der Ebene in linearem Platz so speichern, daß jede Bereichsanfrage mit einem achsenparallelen Rechteck der Höhe h in Zeit $O(\log n + a)$ beantwortet werden kann. Das Einfügen und Entfernen eines Punktes kann in Zeit $O(\log n)$ erfolgen.*

Anwendung: Bereichsanfragen mit fester Höhe

Beweis. Wir denken uns die Ebene in waagerechte Streifen der Höhe h zerlegt, die von unten nach oben durchnumeriert werden. Die Nummern der $\leq n$ vielen Streifen, in denen sich Datenpunkte befinden, werden in einem balancierten Suchbaum T_s gespeichert. Für jeden Streifen S_i werden zwei Prioritätssuchbäume eingerichtet: T_i^o für nach oben beschränkte und T_i^u für nach unten beschränkte Halbstreifen; siehe Abbildung 3.13.

Jedes orthogonale Anfragerechteck q der Höhe h schneidet in der Regel zwei aufeinanderfolgende Streifen. Ihre Nummern erhalten wir als Ergebnis einer eindimensionalen Bereichsanfrage an T_s.

Für den oberen Streifen ist eine Anfrage mit nach oben beschränktem Halbstreifen auszuführen, beim unteren Streifen ist

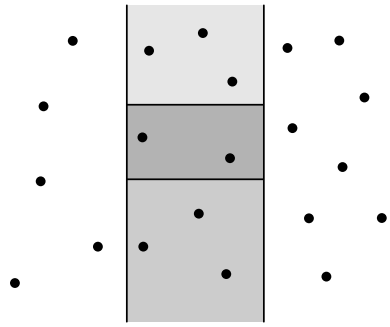


Abb. 3.12 Der Durchschnitt von zwei Halbstreifen bildet ein Rechteck.

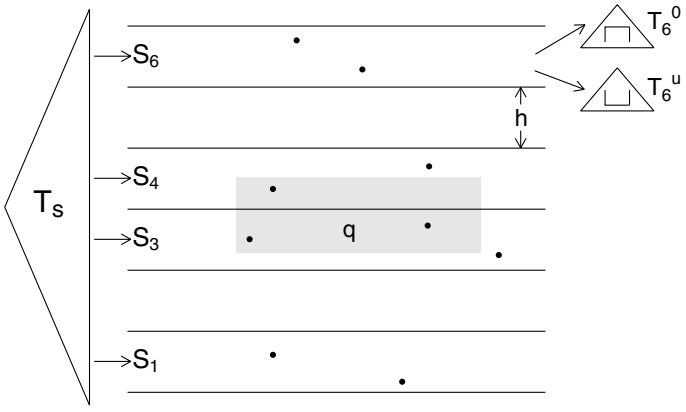


Abb. 3.13 Für jeden nicht-leeren waagerechten Streifen werden zwei Prioritätssuchbäume eingerichtet.

der Halbstreifen nach unten beschränkt. Da für jeden Typ der passende Prioritätssuchbaum T_i^o bzw. T_i^u zur Verfügung steht, kann die Anfrage schnell beantwortet werden. Die Schranken für Laufzeit und Speicherbedarf ergeben sich unmittelbar. □

Weitere Anwendungsmöglichkeiten dieses Prinzips finden sich in Klein et al. [88].

Es gibt eine Fülle von Ergebnissen über geometrische Datenstrukturen, die im Rahmen dieses Kapitels nicht einmal gestreift werden konnten, insbesondere zu externen Datenstrukturen für ausgedehnte Objekte. Wer sich näher für dieses Thema interessiert, sei auf die beiden Bände von Samet [124, 125] hingewiesen. Aktuelle Ergebnisse über interne Strukturen finden sich z. B. bei van Kreveld [139] und Schwarzkopf [130].

Ein Java-Applet, mit dem man interaktiv Prioritätssuchbäume und zweidimensionale Suchbäume bearbeiten kann, findet sich unter

`http://www.geometrylab.de/Geodast/`

Lösungen der Übungsaufgaben

Übungsaufgabe 3.1 Ein Kreis mit Mittelpunkt (x, y) und Radius r enthält den Punkt $p = (p_1, p_2)$ genau dann, wenn

$$\sqrt{(x - p_1)^2 + (y - p_2)^2} \leq r$$

ist. Die Punkte $(x, y, r) \in \mathbb{R}^3$, die diese Ungleichung erfüllen, haben die Eigenschaft, daß der Abstand von Punkt p zu (x, y) im $\mathbb{R}^2$ kleiner gleich der Höhe des Punktes (x, y, r) über der XY-Ebene im $\mathbb{R}^3$ ist; also bilden sie den massiven Kegel mit Spitze $(p_1, p_2, 0)$ und Öffnungswinkel 90°, dessen Mittelsenkrechte auf der XY-Ebene senkrecht steht.

Übungsaufgabe 3.2

(i) Nach den Voraussetzungen gilt

$$\begin{aligned} f(n) = f(k^{\log_k n}) \leq f(k^{\lceil \log_k n \rceil}) &\leq C^{\lceil \log_k n \rceil} \cdot f(1) \\ &\leq C^{1+\log_k n} \cdot f(1) \\ &= C \cdot n^{\log_k C} \cdot f(1) \in O(n^r) \end{aligned}$$

mit $r = \log_k C$; dabei haben wir $C^{\log_k n} = n^{\log_k C}$ benutzt.

(ii) Wegen der Monotonie von f gilt nach Voraussetzung

$$f(n) \leq f\left(k \left\lceil \frac{n}{k} \right\rceil\right) \leq C \cdot f\left(\left\lceil \frac{1}{k} n \right\rceil\right).$$

Ist $\frac{1}{k} \leq \alpha$, folgt die Behauptung direkt aus der Monotonie von f. Sei also $0 < \alpha < \frac{1}{k}$, und sei $r \in \mathbb{N}$ so groß, daß $1/k^r \leq \alpha$ ist. Dann folgt mit Induktion über r

$$f(n) \leq C \cdot f\left(\left\lceil \frac{1}{k} n \right\rceil\right) \leq C^r \cdot f\left(\left\lceil \frac{1}{k^r} n \right\rceil\right) \leq C^r \cdot f(\lceil \alpha n \rceil).$$

Übungsaufgabe 3.3 Angenommen, in einer Operationenfolge der Länge s, die auf eine anfangs leere Struktur ausgeführt wird, kommen k Operationen des Typs *Work* vor. Die aktuellen Strukturgrößen mögen dabei $n_1, \ldots, n_k$ betragen. Nach Voraussetzung und wegen der Monotonie von A sind die Kosten von $\mathit{Work}(n_i)$ durch $A(n_i) \leq A(s)$ beschränkt. Also gilt

$$\frac{1}{k} \sum_{i=1}^{k} A(n_i) \leq \frac{1}{k} k A(s) = A(s).$$

Demnach ist $A(s)$ eine obere Schranke für die amortisierten Kosten von *Work*.

Übungsaufgabe 3.4 Nach den *Delete*-Operationen sind noch $n = 2^b - 1$ Objekte vorhanden. Die letzte Einfügung kostet also $B_V(2^b)$ viel Zeit. Für jede Konstante C kann man b so groß wählen, daß

$$B_V(2^b) > Cb\frac{B_V(2^b)}{2^b} \geq C \log n \frac{B_V(n)}{n}$$

ist; die Kosten der letzten Einfügung liegen also, für sich genommen, oberhalb der in Theorem 3.4 angegebenen Schranke.

Versucht man, durch Mittelung mit den Kosten der $m \geq 1$ vorausgehenden *Insert*-Operationen unterhalb dieser Schranke zu bleiben, so geht in diesen Mittelwert immer die vorletzte Einfügung ein; sie allein hat bereits $B_V(2^a)$ viel Zeit gekostet. Mit $m + 1 \leq 2^{a+b}$ ergibt sich

$$\begin{aligned}
\frac{\sum \text{Kosten der letzten } m+1 \text{ Einfügungen}}{m+1} &\geq \frac{B_V(2^a)}{m+1} \\
&\geq \frac{B_V(2^a)}{2^{a+b}} = \frac{B_V(2^a)}{2^a} \cdot \frac{1}{2^b} \\
&> Cb\frac{B_V(2^b)}{2^b} \geq C \log n \frac{B_V(n)}{n};
\end{aligned}$$

wir brauchen dafür nur zu festem b die Zahl a so groß zu wählen, daß

$$\frac{B_V(2^a)}{2^a} > CbB_V(2^b)$$

gilt. Das ist immer möglich, wenn die Funktion $B_V(n)/n$ streng monoton wächst, zum Beispiel für $B_V(n) = n \log n$.

Übungsaufgabe 3.5 Sei $p = (p_1, p_2), q = (q_1, q_2)$. Falls $p_1 < q_1$ gilt, hat die Punktmenge $[p, q]$ die in Abbildung 3.14 gezeigte Gestalt. Im Fall $p_1 = q_1$ muß $p_2 < q_2$ gelten, und es bleibt nur das Intervall $[p_2, q_2]$ auf der Senkrechten durch p_1 übrig.

Übungsaufgabe 3.6 Außer den Blättern mit den Punkten a, b, h, i, j, k werden alle Knoten besucht. Aber nur der Punkt f liegt im Anfragerechteck.

Übungsaufgabe 3.7 Bei einer wörtlichen Anwendung von Theorem 3.5: ja. Aber dieses Resultat beruht auf der in Abschnitt 3.2.1 eingeführten Idee, die n Punkte in maximal $\log n$ vielen statischen 2–d–Bäumen unterschiedlicher Größe zu speichern. Eine Bereichsanfrage muß für jeden einzelnen Baum T_i beantwortet werden; enthält er m_i Punkte, wird hierfür nach Theorem 3.7 $O(\sqrt{m_i} + a_i)$

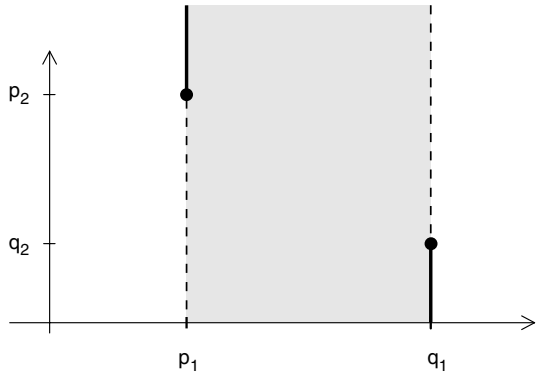


Abb. 3.14 Nur die dick eingezeichneten Randstücke gehören zum Intervall $[p, q]$ bezüglich der lexikographischen Ordnung.

Zeit benötigt, wobei a_i die Anzahl der Punkte in T_i bezeichnet, die zum Anfragebereich gehören. Die Behauptung von Theorem 3.8 ist also richtig, denn es gilt

$$\sum_{i \leq \log n} C(\sqrt{m_i} + a_i) \leq C(\log n \sqrt{n} + a).$$

Übungsaufgabe 3.8

(i) und (iii): Wir wollen die n-elementige Punktmenge $D \subset \mathbb{R}^k$ mit einer orthogonalen Hyperebene $H_s = \{X_1 = s\}$ so in zwei Teile $D_{<s}, D_{>s}$ zerlegen, daß $D \cap H_s = \emptyset$ ist und

$$\min \{|D_{<s}|, |D_{>s}|\} \geq \frac{n-1}{2k}$$

gilt. Zu diesem Zweck lassen wir s von $-\infty$ an wachsen (ein *sweep*!) und verfolgen, wie $|D_{<s}|$ zunimmt; dabei werden nur solche Werte von s betrachtet, für die H_s keine Punkte aus D enthält. Sie bilden disjunkte offene Intervalle auf der X_1-Achse.

Irgendwann springt $|D_{<s}| < \frac{n-1}{2k}$ auf $|D_{<t}| \geq \frac{n-1}{2k}$. Ist $|D_{<t}|$ noch kleiner gleich $|D_{>t}|$, haben wir in H_t die gesuchte Hyperebene gefunden. Ist $|D_{<t}| > |D_{>t}|$ und $|D_{>t}| \geq \frac{n-1}{2k}$, sind wir ebenfalls fertig. Andernfalls ist $|D_{>t}| < \frac{n-1}{2k}$, und es folgt

$$\begin{aligned} |D_{<t}| - |D_{<s}| &= n - |D_{>t}| - |D_{<s}| \\ &> n - \frac{n-1}{2k} - \frac{n-1}{2k} \\ &= \frac{(k-1)n+1}{k}, \end{aligned}$$

siehe Abbildung 3.15. Wenn die Anzahl der Punkte „links“ von der wandernden Hyperebene zwischen s und t um diesen Betrag nach oben springt, muß es eine Position s' dazwischen geben, in der die Hyperebene $H_{s'}$ mindestens

$$\left\lfloor \frac{(k-1)n+1}{k} \right\rfloor + 1$$

viele Punkte aus D enthält.

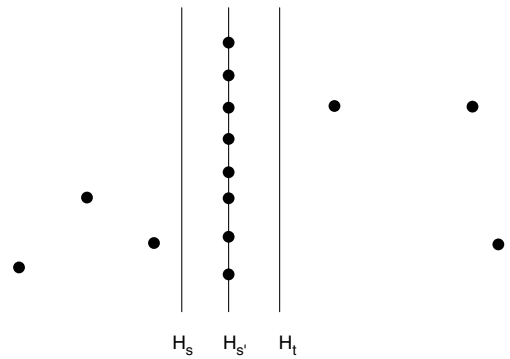


Abb. 3.15 Wenn 14 Punkte sich durch eine Senkrechte nicht so trennen lassen, daß die kleinere Teilmenge mindestens 4 Punkte enthält, gibt es eine Senkrechte, auf der mindestens 8 Punkte liegen.

Diese Überlegung stellen wir für jede der k Koordinaten an. Angenommen, es gibt jedesmal eine Hyperebene H_i, die mindestens $\lfloor \frac{(k-1)n+1}{k} \rfloor + 1$ viele Punkte aus D enthält. Dann ist

$$\sum_{i=1}^{k} |D \cap H_i| \geq k \left(\left\lfloor \frac{(k-1)n+1}{k} \right\rfloor + 1 \right) > (k-1)n + 1.$$

Andererseits ist der Durchschnitt aller H_i ein Punkt im $\mathbb{R}^k$; folglich kann höchstens ein Punkt aus D in allen k Hyperebenen vorkommen, jeder andere Punkt in höchstens $k-1$ vielen. Es entsteht der Widerspruch

$$\sum_{i=1}^{k} |D \cap H_i| \leq k + (n-1)(k-1) = (k-1)n + 1.$$

Daß es keine größere untere Schranke als $\frac{n-1}{2k}$ für die Größe des kleineren Teils geben kann, zeigt die Verallgemeinerung der in Abbildung 3.6 gezeigten Punktmenge auf höhere Dimensionen.

(ii) Nein. Wenn bei jedem Teilungsschritt ein Größenverhältnis von $1:3$ zwischen den beiden Teilmengen erlaubt wäre, so könnte der resultierende 2–d–Baum eine Höhe von $h = \left\lceil \log_{\frac{4}{3}} n \right\rceil = \left\lceil \frac{\log n}{2-\log 3} \right\rceil \approx 2.409 \cdot \log n$ erreichen. Nach Lemma 3.6 hätten wir dann für Bereichsanfragen die Laufzeitschranke $O(2^{\frac{h}{2}}) = O(n^{1.204\ldots})$, wobei die Größe a der Antwort noch hinzukäme.

Übungsaufgabe 3.9

(i) Wenn wir bei denjenigen Blättern von T^1, deren Zahlen in I liegen, beginnen und so lange zu den Vaterknoten hochsteigen, wie deren Intervalle ganz in I enthalten sind, enden wir genau bei den Knoten $v_1, \ldots, v_l$.

(ii) Kein Knoten v_i kann Vorfahre eines anderen Knotens v_j sein. Folglich sind $I(v_i)$ und $I(v_j)$ disjunkt.

(iii) Wenn einer der Knoten v_i selbst nicht die Wurzel von T^1 ist, gilt für seinen Vaterknoten w

$$I(w) \cap I \neq \emptyset \text{ und } I(w) \not\subset I,$$

denn v_i ist ja ein Knoten minimaler Tiefe, dessen Intervall ganz in I enthalten ist. Alle Knoten w mit dieser Eigenschaft liegen in T^1 auf den beiden Suchpfaden zu den Intervallgrenzen von I;[12] denn für jeden Knoten x, der nicht auf einem der beiden Suchpfade liegt, gilt entweder $I(x) \subseteq I$ oder $I(x) \cap I = \emptyset$. Also kann man die Knoten v_i in Zeit $O(\log n)$ finden. Um den Inhalt ihrer Intervalle $I(v_i)$ auszugeben, muß man alle Zahlen berichten, die in den Blättern der Teilbäume gespeichert sind, die an den Knoten v_i hängen. Die für diesen Abstieg benötigte Zeit ist proportional zur Anzahl der Blätter, also zur Größe a der Antwort. Insgesamt läßt sich eine Bereichsanfrage deshalb in Zeit $O(\log n + a)$ beantworten.

Übungsaufgabe 3.10 Ein Binärbaum mit n Blättern hat $n-1$ innere Knoten. In einem kompletten Prioritätssuchbaum für n Punkte muß also mindestens ein Blatt einen Punkt enthalten (es können aber auch mehr sein).

Es bleibt zu zeigen, daß der vorhandene Platz ausreicht, um gleichzeitig die Bedingungen (1) und (2) für Prioritätssuchbäume zu erfüllen. Wir verwenden Induktion über n. Für $n = 1$ ist die Behauptung klar. Ist $n \geq 2$, betrachten wir die beiden Teilbäume,

[12] Die Suchpfade sind in Abbildung 3.8 fett hervorgehoben.

die am Wurzelknoten hängen. Nach Induktionsvoraussetzung lassen sich diejenigen Punkte, deren X-Koordinaten in den jeweiligen Blättern stehen, entsprechend (1) und (2) in den Teilbäumen unterbringen.

Wenn wir nun den leeren Wurzelknoten oben aufsetzen, können wir rekursiv denjenigen Punkt aus den beiden Söhnen aufsteigen lassen, der die kleinere Y-Koordinate hat. Am Ende sind alle drei Bedingungen erfüllt.

Übungsaufgabe 3.11 Wenn man im Beispiel von Abbildung 3.10 eine Anfrage mit dem Halbstreifen $[0, 2] \times (-\infty, 2]$ beantwortet, läuft man zu den Blättern 1 und 3, ohne einen Datenpunkt zu finden. Im allgemeinen läßt sich nicht vermeiden, daß man den Baum in ganzer Höhe durchlaufen muß.

4

Durchschnitte und Sichtbarkeit

In diesem Kapitel werden einige geometrische Probleme behandelt, die zu den Klassikern der Algorithmischen Geometrie zählen. Fast alle haben mit der Bildung von *Durchschnitten* zu tun oder mit *Sichtbarkeit in einfachen Polygonen.* Die Probleme lassen sich leicht formulieren, aber trotzdem sind die Lösungen oft überraschend und keineswegs selbstverständlich.

4.1 Die konvexe Hülle ebener Punktmengen

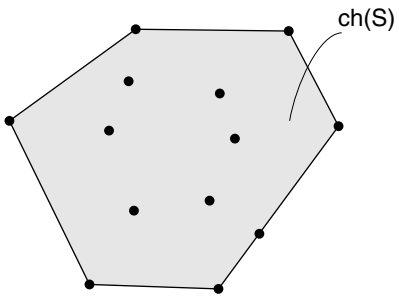


Abb. 4.1 Eine Menge S von 13 Punkten in der Ebene und ihre konvexe Hülle.

In Kapitel 1 auf Seite 22 hatten wir die *konvexe Hülle* $ch(A)$ einer Teilmenge A des $\mathbb{R}^d$ durch konvexe Hülle

$$ch(A) = \bigcap_{\substack{K \supseteq A \\ K \text{ konvex}}} K$$

definiert; sie ist die kleinste konvexe Menge im $\mathbb{R}^d$, die A enthält.

Ein praktisches Verfahren zur Berechnung der konvexen Hülle ergibt sich durch diese Definition aber noch nicht.

In diesem Abschnitt sollen deshalb effiziente Verfahren vorgestellt werden, um die konvexe Hülle einer Menge S von n Punkten in der Ebene zu konstruieren. Abbildung 4.1 zeigt ein Beispiel. Es fällt auf, daß die konvexe Hülle der gegebenen Punktmenge ein konvexes Polygon ist, dessen Ecken zur Punktmenge S gehören.

Noch ein anderer Sachverhalt wird durch Abbildung 4.1 suggeriert: Stellt man sich die Punkte als Nägel vor, die aus der Ebene herausragen, wirkt der Rand der konvexen Hülle wie ein Gummiband, das sich um die Nägel herumspannt. Wenn diese Vorstellung richtig ist, müßte der Rand von $ch(S)$ der kürzeste geschlossene Weg sein, der die Punktmenge S umschließt. Wir werden uns noch vergewissern, daß diese Aussage, die ja nicht unmittelbar aus der oben angegebenen Definition der konvexen Hülle folgt, tatsächlich zutrifft.

4.1.1 Präzisierung des Problems und untere Schranke

Zuerst machen wir uns klar, daß der Rand der konvexen Hülle einer ebenen Punktmenge tatsächlich so ausieht, wie Abbildung 4.1 nahelegt.

Lemma 4.1 *Sei S eine Menge von n Punkten in der Ebene. Dann ist $ch(S)$ ein konvexes Polygon, dessen Ecken Punkte aus S sind.*

Beweis. Durch Induktion über n. Für $n \leq 2$ ist die Behauptung klar.[1] Sei also $n \geq 3$. Wir wählen einen Punkt p aus und wenden die Induktionsvoraussetzung auf die Restmenge $S' = S \setminus \{p\}$ an; ihre konvexe Hülle ist also ein konvexes Polygon mit Ecken in S'. Liegt auch p in $ch(S')$, ist nichts weiter zu zeigen. Liegt p außerhalb, haben wir die in Abbildung 4.2 gezeigte Situation. Wir nehmen eine von p ausgehende Halbgerade H, die $ch(S')$ nicht schneidet, und drehen sie so lange *gegen* den Uhrzeiger um p, bis sie $ch(S')$ berührt; sei a der von H berührte Punkt von $ch(S')$. Analog drehen wir die Halbgerade *mit* dem Uhrzeiger, bis sie den Rand von $ch(S')$ in einem Punkt b berührt. Behauptung:

$$ch(S) = ch(S') \cup tria(a, b, p).$$

Offenbar enthält jede konvexe Obermenge von S erst recht die konvexen Hüllen $ch(S')$ und $tria(a, b, p)$ der Teilmengen S' und

[1] Wenn man dazu bereit ist, einen einzelnen Punkt oder ein Liniensegment als konvexe Polygone anzuerkennen.

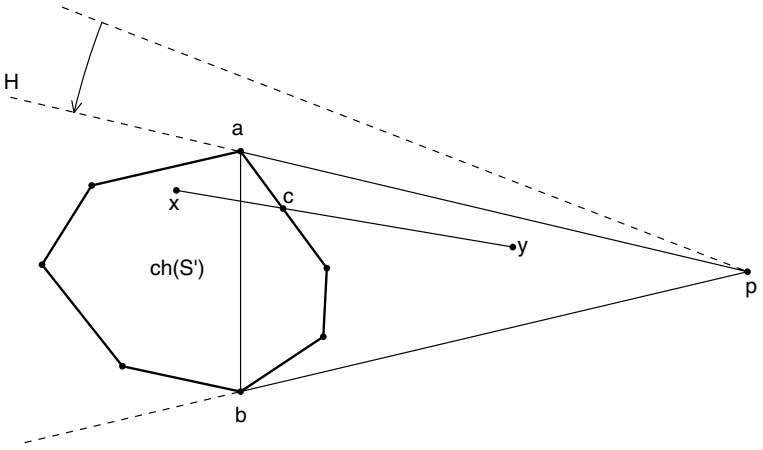


Abb. 4.2 Für jeden Punkt $z \in ab$ ist das Liniensegment zp in der konvexen Hülle von $S' \cup \{p\}$ enthalten.

$\{a, b, p\}$ von S. Es bleibt zu zeigen, daß die Vereinigung von $ch(S')$ und dem Dreieck $tria(a, b, p)$ eine konvexe Menge ist. Weil jede der beiden Mengen für sich konvex ist, müssen wir nur für je zwei Punkte $x \in ch(S')$ und $y \in tria(a, b, p) \setminus ch(S')$ nachweisen, daß das Liniensegment xy in der Vereinigung der Mengen enthalten ist.

Sei c der Schnittpunkt von xy mit dem Rand von $ch(S')$. Weil ap und bp Teile von Tangenten von p an $ch(S')$ sind, ist c in $tria(a, b, p)$ enthalten. Wegen $xc \subset ch(S')$ und $cy \subset tria(a, b, p)$ liegt das ganze Segment xy in der Vereinigung.

Also stimmt die Behauptung, und der Rand von $ch(S)$ ist ein konvexes Polygon mit Ecken in S. □

präzisierte Problemstellung

Jetzt können wir unser Problem korrekt formulieren: Gegeben ist eine Menge von n Punkten in der Ebene; zu bestimmen ist das konvexe Polygon $ch(S)$ durch Angabe seiner Ecken in der Reihenfolge gegen den Uhrzeigersinn.

Es stellt sich heraus, daß unser Problem mindestens so schwierig ist wie das Sortierproblem.

untere Schranke

Lemma 4.2 *Die Konstruktion der konvexen Hülle von n Punkten in der Ebene erfordert $\Omega(n \log n)$ viel Rechenzeit.*

Beweis. Sei A ein beliebiges Verfahren zur Konstruktion der konvexen Hülle einer ebenen Punktmenge. Gegeben seien n unsortierte reelle Zahlen $x_1, \ldots, x_n$. Wir wenden A an, um die konvexe

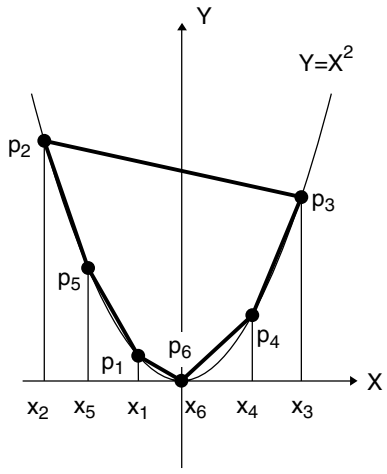


Abb. 4.3 Die konvexe Hülle von 6 Punkten auf einer Parabel.

Hülle der Punkte $p_i = (x_i, x_i^2)$ zu berechnen. Sie liegen auf der Parabel $Y = X^2$, und jedes p_i ist ein Eckpunkt der konvexen Hülle; siehe Abbildung 4.3.

Nachdem wir das konvexe Polygon $ch(\{p_1, \ldots, p_n\})$ berechnet haben, können wir leicht in linearer Zeit seine Eckpunkte entgegen dem Uhrzeigersinn ausgeben. Dabei werden ab dem am weitesten links gelegenen Punkt die Punkte p_i in aufsteigender Reihenfolge ihrer X-Koordinaten berichtet. Damit sind die Zahlen x_i sortiert, was nach Korollar 1.8 auf Seite 41 die Zeitkomplexität $\Omega(n \log n)$ besitzt. □

Bevor wir uns der Konstruktion der konvexen Hülle zuwenden, wollen wir uns klarmachen, daß ihr Rand tatsächlich die kürzeste geschlossene Kurve ist, die die Punktmenge S umschließt.

Lemma 4.3 *Sei W ein einfacher, geschlossener Weg, der die konvexe Hülle der Punktmenge S umschließt.*[2] *Dann ist der Rand von $ch(S)$ höchstens so lang wie W.*

Beweis. Von jeder Ecke v_i von $ch(S)$ ziehen wir die winkelhalbierende Halbgerade h_i nach außen. Dabei trifft jedes h_i die Kurve W, eventuell sogar mehrfach. Wir orientieren W gegen den Uhrzeiger, starten von einem Punkt t und bezeichnen den jeweils ersten Schnittpunkt von W mit der Halbgeraden h_i als w_i; siehe Abbildung 4.4.

Die Halbgerade h_i bildet mit den beiden Kanten von v_i einen Winkel $\geq \pi/2$; für je zwei konsekutive Halbgeraden h_i und h_{i+1}

[2] Wir erinnern an den Jordanschen Kurvensatz; siehe Seite 12.

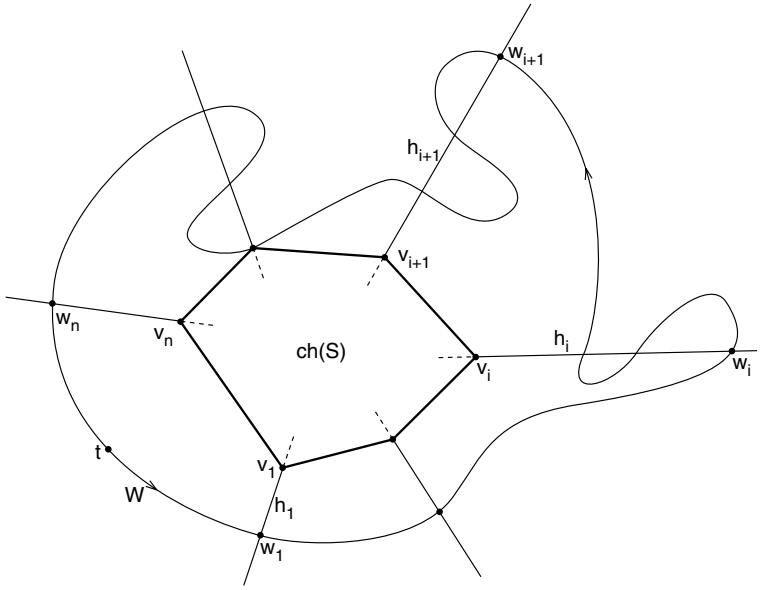


Abb. 4.4 Für jedes i ist $|v_i v_{i+1}| \leq |w_i w_{i+1}|$. Folglich ist W mindestens so lang wie der Rand des konvexen Polygons.

ist deshalb (v_i, v_{i+1}) das Punktepaar mit dem kleinsten Abstand in $h_i \times h_{i+1}$. Darum gilt

$$|v_i v_{i+1}| \leq |w_i w_{i+1}| \leq |W_{i,i+1}|,$$

wobei $W_{i,i+1}$ das Stück von W zwischen w_i und w_{i+1} bezeichnet. Insgesamt ist daher die Gesamtlänge von $\partial ch(S)$ nicht größer als die Länge von W. □

Man kann sich leicht überlegen, daß Lemma 4.3 auch für solche Wege W gilt, die nicht einfach sind und $ch(S)$ in einer ihrer Schlingen enthalten, d. h. in einer beschränkten Zusammenhangskomponente von $\mathbb{R}^2 \setminus W$.

Außerdem kann man Lemma 4.3 auf beliebige konvexe Mengen (statt $ch(S)$) verallgemeinern und folgende nützliche Folgerung ziehen:

Korollar 4.4 *Ist W ein einfacher „konvexer" Weg von a nach b, d. h. W zusammen mit dem Liniensegment ab umschließt eine konvexe Menge, und ist W' irgendein anderer Weg von a nach b, der auf der Außenseite von W verläuft, so ist W' mindestens so lang wie W.*

Diese Situation ist in Abbildung 4.5 dargestellt.

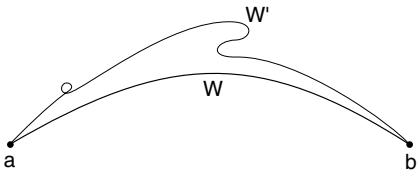


Abb. 4.5 Der Weg W' ist mindestens so lang wie der konvexe Weg W.

4.1.2 Inkrementelle Verfahren

die Idee

Es fällt auf, daß der Beweis von Lemma 4.1 schon die Idee für einen Algorithmus enthält, mit dem man die konvexe Hülle einer Punktmenge S konstruieren kann.

Man startet mit der konvexen Hülle von drei Punkten p_1, p_2, p_3 und nimmt nach und nach weitere Punkte hinzu. Wenn ein neuer Punkt p hinzukommt, testet man, ob er in der Hülle $ch(S')$ der schon verarbeiteten Punktmenge S' liegt. Falls nicht, muß das durch die Tangenten von p an $ch(S')$ erzeugte Dreieck angebaut werden; vgl. Abbildung 4.2.

Solch ein Verfahren zur Konstruktion einer Struktur für eine Menge von Objekten, bei dem die Objekte eines nach dem anderen hinzugefügt werden und die Struktur jedesmal vergrößert wird, heißt *inkrementell*.

inkrementelles Verfahren

Wie können wir unsere Idee zur inkrementellen Konstruktion der konvexen Hülle in die Tat umsetzen? Offenbar sind für $C = ch(S')$ zwei Teilprobleme zu lösen:

(1) Wie testet man effizient, ob ein Punkt p innerhalb oder außerhalb des konvexen Polygons C liegt?

(2) Wie bestimmt man die beiden Tangenten von einem Punkt p außerhalb des konvexen Polygons C an den Rand von C?

Der Rest – die Aktualisierung der schon konstruierten konvexen Hülle C durch Vereinigung mit dem Dreieck, das durch p und die beiden Tangentenpunkte gebildet wird – kann dann jeweils in konstanter Zeit geschehen, liefert also insgesamt nur einen Beitrag in Höhe von $O(n)$.

Teilproblem (1) in $O(n)$ Zeit

Auf Seite 89 hatten wir bereits ein Verfahren zur Lösung von Teilproblem (1) formuliert, das in Zeit $O(n)$ funktioniert: Man wählt eine von p ausgehende Halbgerade h aus und testet alle

Kanten von C auf Schnitt mit h; der Punkt p liegt genau dann im Innern von C, wenn die Anzahl der Schnitte ungerade ist.[3]

Die in Teilproblem (2) verlangten Tangenten lassen sich ebenfalls in Zeit $O(n)$ bestimmen, indem man einmal um den Rand von C herumwandert und an jedem Eckpunkt v prüft, ob der Strahl von p durch v das Innere von C schneidet. Wenn nicht, liegt v auf einer der gesuchten Tangenten. Es kann vorkommen, daß eine Tangente mehrere kolineare Eckpunkte von C enthält. In einem solchen Fall suchen wir den Eckpunkt a auf, der p am nächsten liegt, denn wir wollen ja alle Punkte von S berichten, die auf dem Rand der konvexen Hülle liegen; vgl. Abbildung 4.6.

Teilproblem (2) in $O(n)$ Zeit

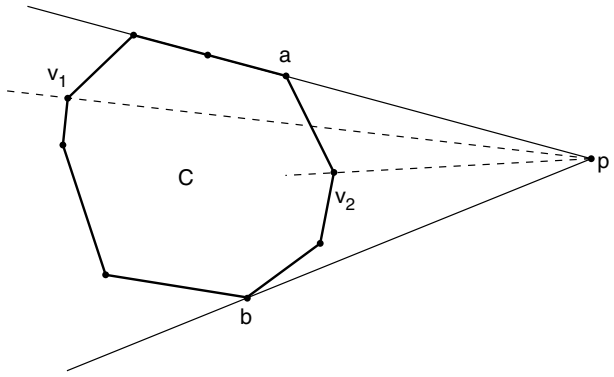


Abb. 4.6 Die Punkte a und b sind die gesuchten Tangentenpunkte. Am Schnitt von ∂C mit dem Strahl von p durch v_i läßt sich erkennen, daß v_1 und v_2 keine Tangentenpunkte sind.

Weil die Teilprobleme (1) und (2) für jeden einzufügenden Punkt p gelöst werden müssen, ergibt sich ein Verfahren mit Laufzeit in $O(n^2)$ für die Konstruktion der konvexen Hülle von n Punkten.

damit: Laufzeit $O(n^2)$

Im Hinblick auf die untere Schranke in $\Omega(n \log n)$ ist das Ergebnis noch nicht befriedigend. Wo läßt sich Rechenzeit einsparen?

Eine Idee zur effizienteren Lösung von Teilproblem (1) wird durch Abbildung 4.7 illustriert: Angenommen, wir kennen einen Punkt z im Innern des konvexen Polygons C.[4] Die Halbgeraden h_i von z durch die Ecken v_i von C zerlegen die gesamte Ebene in Sektoren, deren Spitzen am Punkt z zusammenkommen wie die Stücke einer Torte.

Teilproblem (1) in $O(\log n)$ Zeit

[3] Bei einem konvexen Polygon kann es höchstens zwei Schnittpunkte geben; trotzdem muß man im schlimmsten Fall alle Kanten testen.

[4] Wir können zu Beginn den Punkt z im Innern des Dreiecks $tria(p_1, p_2, p_3)$ wählen und brauchen ihn danach nie wieder zu verändern, weil er dann auch im Innern jeder Hülle $ch(S')$ enthalten ist.

Wenn nun ein Punkt $p \neq z$ gegeben ist, stellen wir zunächst fest, in welchem Sektor er sich befindet. Angenommen, dieser Sektor wird von den Halbgeraden h_i und h_{i+1} berandet; dann brauchen wir nur noch zu testen, ob p – von z aus gesehen – diesseits oder jenseits der Kante v_iv_{i+1} liegt.

Die Bestimmung des p enthaltenden Sektors kann in Zeit $O(\log n)$ erfolgen, denn die Halbgeraden h_i sind ja um z herum nach ihrer Steigung geordnet. Dazu bräuchte man die Punkte v_i nach ihren Winkeln um z sortiert eigentlich nur in einem Array zu speichern, das zum Beispiel mit dem Eckpunkt größter X-Koordinate beginnt; auf dieses Array ließe sich *binäres Suchen* anwenden.

Wenn aber der neue Punkt p eingefügt ist, muß dieses Eckenverzeichnis entsprechend erweitert werden, damit es für das Einfügen des nächsten Punktes wieder zur Verfügung steht. Statt eines Arrays nimmt man also besser einen *balancierten Suchbaum*; dann kann man das Eckenverzeichnis auch in Zeit $O(\log n)$ aktualisieren.

alle Teilprobleme (2) in Zeit $O(n)$

Zur Lösung von Teilproblem (2) – dem Auffinden der Tangenten – verfahren wir folgendermaßen: Wir kennen ja nun eine Kante in dem von p aus sichtbaren Teil des Randes von C, nämlich die Kante v_iv_{i+1} im Sektor von p! Von dort aus laufen wir in beide Richtungen auf dem Rand entlang, bis die Tangentenpunkte a und b gefunden sind; siehe Abbildung 4.7.

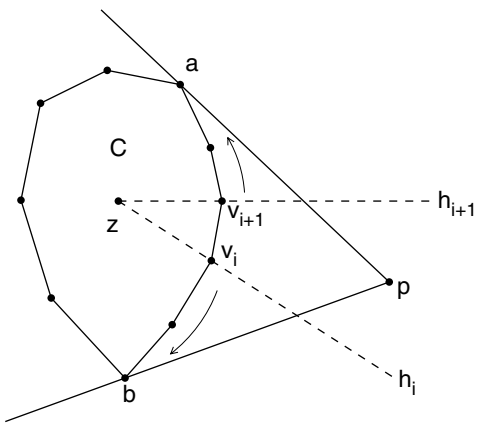


Abb. 4.7 Von der Kante v_iv_{i+1} läuft man in beide Richtungen zu den Tangentenpunkten a und b.

Zwar kann die Anzahl der unterwegs besuchten Ecken von C in $\Omega(n)$ liegen, aber jede von ihnen wird anschließend aus der konvexen Hülle entfernt und kann danach nie wieder als Eckpunkt in

Erscheinung treten. Folglich wird sie auch nie wieder besucht! Weil alle Ecken aus der Eingabemenge S stammen, kann es *insgesamt* höchstens $O(n)$ Besuche geben.

Zusammen mit der unteren Schranke aus Lemma 4.2 haben wir damit folgendes Resultat:

Laufzeit $\Theta(n \log n)$

Theorem 4.5 *Die konvexe Hülle von n Punkten in der Ebene zu konstruieren, hat die Zeitkomplexität $\Theta(n \log n)$ und benötigt linearen Speicherplatz.*

Aus theoretischer Sicht mag der gerade vorgestellte Algorithmus befriedigend erscheinen, schließlich ist seine Laufzeit größenordnungsmäßig optimal. Es geht aber noch viel einfacher, ganz ohne balancierte Bäume.

Statt dessen verwenden wir ein Array A, in dem für jeden noch nicht eingefügten Punkt $p_j \in \{p_i, p_{i+1}, \ldots, p_n\}$ ein Verweis auf diejenige Kante e_j von $ch(S_{i-1})$ steht, die vom Liniensegment zp_j geschnitten wird,[5] oder **nil**, falls p_j innerhalb von $ch(S_{i-1})$ liegt. Die Kanten der konvexen Hülle können dabei als zyklisch verkettete Liste der Eckpunkte gespeichert sein, wie für Polygone üblich.

Initialisierung

Am Anfang wird A für $i = 4$ initialisiert; dabei muß für jeden Punkt p_j mit $j \geq 4$ getestet werden, ob er außerhalb des Dreiecks $tria(p_1, p_2, p_3)$ liegt und welche Dreieckskante das Liniensegment zp_j in diesem Fall schneidet. Das kann insgesamt in linearer Zeit geschehen.

Aktualisierung

Die Frage ist, welchen Aufwand die *Aktualisierung* des Arrays A erfordert. Wenn p_i in $ch(S_{i-1})$ liegt, kann alles beim alten bleiben. Andernfalls ändert sich die konvexe Hülle, was Änderungen einiger Einträge $A[j]$ nach sich ziehen kann. Wir sagen, daß solch ein Punkt p_j *mit p_i in Konflikt steht*; genauer ausgedrückt definieren wir für $i < j$:

Konflikt

$$(p_i, p_j) \text{ in Konflikt} \iff zp_j \text{ schneidet eine Kante von } ch(S_{i-1}), \text{ die beim Einfügen von } p_i \text{ entfernt wird.}$$

Für jeden Punkt p_j, der mit p_i in Konflikt steht, ist zu testen, ob er im Dreieck $tria(a, b, p_i)$ enthalten ist – dann liegt er im Innern der neuen konvexen Hülle –, oder andernfalls, welche der beiden neuen Kanten ap_i oder bp_i von zp_j geschnitten wird; entsprechend ist der Eintrag $A[j]$ zu aktualisieren.

Damit wir nach den Punkten p_j, die mit p_i in Konflikt stehen, nicht lange suchen müssen, führen wir für jede Kante e von $ch(S_{i-1})$ eine Liste[6] mit allen p_j, für die $A[j] = e$ ist. Abbildung 4.8 zeigt ein Beispiel für die gesamte Datenstruktur.

[5] Das ist gerade die Kante, deren Sektor den Punkt p_j enthält.

[6] Physisch können diese Zeiger an einem der Endpunkte von e sitzen.

Das Einfügen von p_i vollzieht sich also insgesamt folgendermaßen:

```
if A[i] ≠ nil
then
    e := A[i];
    laufe von e auf dem Rand von ch(S_{i-1}) bis zu
    den Tangentenpunkten a und b;
    (* konvexe Hülle aktualisieren *)
    entferne das e enthaltende Randstück R
        zwischen b und a;
    füge die Kanten ap_i und bp_i ein;
    (* Hilfsstruktur aktualisieren *)
    for jede Kante e auf dem entfernten Randstück R
        for jeden Punkt p_j, auf den e verweist
            (* p_j ist in Konflikt mit p_i *)
            if zp_j ∩ ap_i ≠ ∅
            then
                A[j] := ap_i;
                füge bei Kante ap_i Verweis auf p_j ein
            elseif zp_j ∩ bp_i ≠ ∅
            then
                A[j] := bp_i;
                füge bei Kante bp_i Verweis auf p_j ein
            elseif p_j in tria(a, b, p_i)
            then
                A[j] := nil
```

Der Gesamtaufwand für das Einfügen aller Punkte p_i ist offenbar proportional zur Summe folgender Größen:

- Gesamtzahl aller jemals besuchten Kanten e,
- Gesamtzahl aller Konfliktpaare (p_i, p_j) mit $i < j$.

$\Theta(n^2)$ viele Konflikte?

Daß der erste Anteil in $O(n)$ liegt, hatten wir uns schon überlegt. Aber wie sieht es mit dem zweiten Anteil aus? Schließlich ist hier von Paaren die Rede, von denen es quadratisch viele gibt.

im *worst case*: ja!

Abbildung 4.9 zeigt ein Beispiel, bei dem insgesamt tatsächlich $\Theta(n^2)$ viele Konfliktpaare auftreten.

Man hat aber bei diesem Beispiel den Eindruck, daß die Vielzahl an Konfliktpaaren auf eine besonders ungünstige Einfügereihenfolge zurückzuführen ist. Würden etwa zuerst die Punkte p_1, p_2, $p_{\frac{n}{2}}$, $p_{\frac{n}{2}+1}$, p_n eingefügt, so träten danach keine Konflikte mehr auf, weil $ch(S)$ bereits fertig ist; insgesamt entstünden nur $O(n)$ Konflikte.

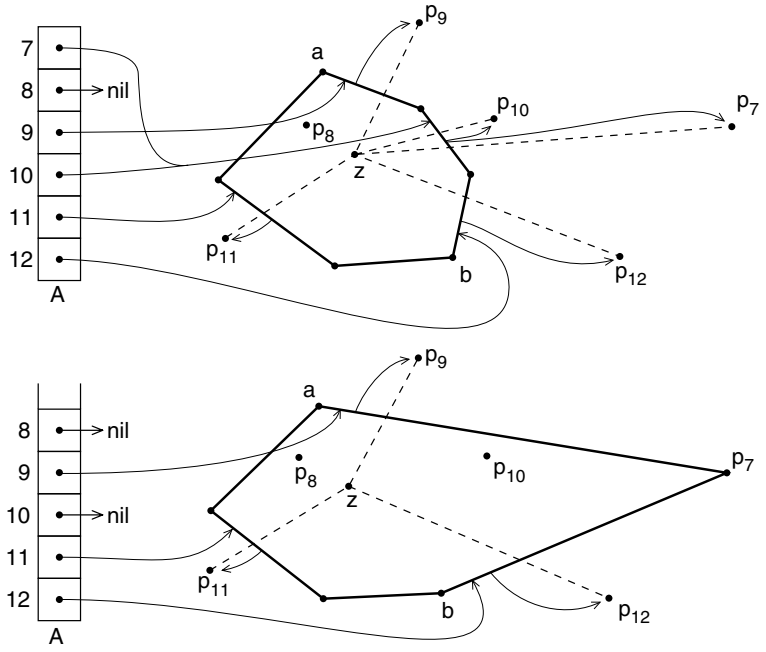


Abb. 4.8 Die konvexe Hülle nebst Hilfsstrukturen vor und nach dem Einfügen von Punkt p_7.

Diese Betrachtung legt es nahe, vor Beginn des Hüllenbaus eine *zufällige Einfügereihenfolge* auszuwürfeln und darauf zu hoffen, daß sich im Mittel über alle $n!$ möglichen Reihenfolgen ein günstigeres Laufzeitverhalten ergibt. Diese Hoffnung wird nicht enttäuscht.

zufällige Einfügereihenfolge

Theorem 4.6 *Beim Bau der konvexen Hülle von n Punkten sind bei Verwendung des oben beschriebenen inkrementellen Verfahrens $O(n \log n)$ Konfliktpaare und eine Laufzeit von $O(n \log n)$ zu erwarten, wenn jede der $n!$ möglichen Einfügereihenfolgen mit derselben Wahrscheinlichkeit gewählt wird.*

Laufzeit $O(n \log n)$ bei Randomisierung

Beweis. Nur die Aussage über die Anzahl der Konfliktpaare (p_i, p_j) ist noch zu beweisen. Sie treten in zwei Arten auf: Wenn der Punkt p_j nach Einfügen von p_i in $ch(S_i)$ enthalten ist, kann er danach keine Konflikte mehr verursachen; für jedes p_j existiert also höchstens *ein* solches Paar (p_i, p_j), insgesamt demnach nur $O(n)$ viele.

Bei den anderen Konfliktpaaren (p_i, p_j) liegt p_j immer noch außerhalb von $ch(S_i)$; das Liniensegment zp_j schneidet dann den

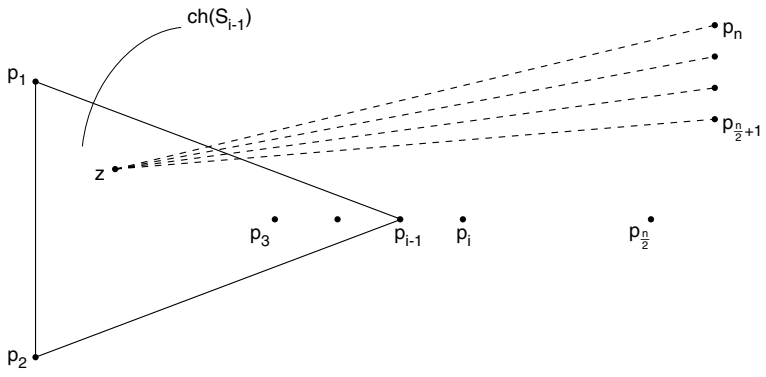


Abb. 4.9 Jeder der Punkte ganz rechts steht mit jedem $p_i, 3 \leq i \leq n/2$ in Konflikt, denn sein Liniensegment zu z schneidet eine Kante von $ch(S_{i-1})$, die beim Einfügen von p_i entfernt wird.

Rand von $ch(S_i)$ in einer Kante $e = A[j]$, die den soeben eingefügten Punkt p_i als Endpunkt hat.

Nun hat jedes Element von S_i dieselbe Chance $\frac{1}{i}$, der zuletzt eingefügte Punkt p_i zu sein.[7] Folglich besteht die Wahrscheinlichkeit $\frac{2}{i}$, daß p_i einer der beiden Endpunkte von e ist und deshalb mit p_j ein Konfliktpaar der zweiten Art bildet.

Die zu erwartende Anzahl *aller* Konfliktpaare (p_i, p_j) beträgt daher für festes j höchstens

$$1 + \sum_{i=1}^{j} \frac{2}{i} \leq 1 + 2\left(1 + \int_1^j \frac{1}{x}\, dx\right) = 1 + 2(1 + \ln j) \in O(\log n),$$

insgesamt also $O(n \log n)$. □

randomisierte inkrementelle Konstruktion: ein Paradigma

Die Wahl einer zufälligen Einfügereihenfolge mit anschließendem inkrementellem Vorgehen ist ein wichtiges und häufig verwendetes Verfahren der Algorithmischen Geometrie; in der englischsprachigen Literatur nennt man diesen Ansatz *randomized incremental construction*.[8]

[7] Wir können jede Permutation mit gleicher Wahrscheinlichkeit erzeugen, indem wir zuerst p_n in S so wählen, daß jedes Element mit Wahrscheinlichkeit $\frac{1}{n}$ an die Reihe kommt, dann p_{n-1} mit Wahrscheinlichkeit $\frac{1}{n-1}$ in $S \setminus \{p_n\}$, und so fort.

[8] Wir haben es hier mit randomisierten Algorithmen zu tun, wie sie am Ende von Abschnitt 1.2.4 eingeführt wurden: Gemittelt wird über die verschiedenen möglichen Rechengänge des Verfahrens bei einer beliebigen, festen Eingabe. Ein völlig anderer Ansatz wäre es, über die verschiedenen möglichen Eingaben zu mitteln. Außerdem sei noch erwähnt, daß die hier vorgestellten

Eine Auswahl randomisierter geometrischer Algorithmen findet sich in dem Buch von Mulmuley [107]. Das oben beschriebene randomisierte Verfahren zur Konstruktion der konvexen Hülle steht bei Seidel [134] in einem von Pach herausgegebenen Sammelband über Probleme aus der Diskreten und Algorithmischen Geometrie.

4.1.3 Ein einfaches optimales Verfahren

Es gibt eine ganze Reihe von Algorithmen zur Konstruktion der konvexen Hülle einer ebenen Punktmenge S, darunter auch solche, die worst-case-optimal *und* praktikabel sind. Wir wollen jetzt ein besonders einfaches Verfahren beschreiben, das in Drappier [49] dargestellt wird. Zunächst nehmen wir an, daß die Punkte in S paarweise verschiedene X- und Y-Koordinaten haben. Das Verfahren läuft in *zwei Phasen*: Zuerst wird ein einfaches[9] Polygon P konstruiert, dessen Ecken zu S gehören und das jeden Punkt von S enthält. Anschließend wird die konvexe Hülle von P gebildet. Damit sind wir fertig, denn es gilt allgemein folgende Aussage:

zwei Phasen

Übungsaufgabe 4.1 Für jedes einfache Polygon P, das die Punktmenge S enthält und dessen Ecken alle zu S gehören, ist $ch(P) = ch(S)$.

Das Polygon P ist von besonders einfacher Gestalt; siehe Abbildung 4.10. Anschaulich entsteht P folgendermaßen: Wir stellen uns wieder vor, daß die Punkte in S als Nägel aus der XY-Ebene herausragen. Links von S liegt ein senkrechter Wollfaden. Er wird nun von einem Gebläse nach rechts gegen die Punkte getrieben. Zuerst trifft er dabei auf den linken Extrempunkt L von S, d. h. auf den Punkt mit minimaler X-Koordinate. Dann schmiegt sich der Faden den Nägeln in der in Abbildung 4.10 gezeigten Weise an, bis er den oberen Extrempunkt O und den unteren Extrempunkt U erreicht hat; dort wird er abgeschnitten. Dasselbe Vorgehen wird von rechts wiederholt. Man nennt die beiden Fäden in ihrer endgültigen Lage die *linke* und die *rechte Kontur der Punktmenge* S. Übrigens sind die Extrempunkte L, R, U, O schon Ecken der konvexen Hülle.

Kontur einer Punktmenge

randomisierten Algorithmen vom Typ *Las Vegas* sind, weil sie in jedem Fall das korrekte Ergebnis liefern, ihre Laufzeit aber schwanken darf. Dagegen ist bei *Monte-Carlo*-Verfahren das Ergebnis nur mit einer bestimmten Wahrscheinlichkeit korrekt.

[9] Strenggenommen ist P nicht immer einfach: es kann an mehreren Stellen „flachgedrückt“ sein; siehe Abbildung 4.11. Das wird uns im folgenden aber nicht stören.

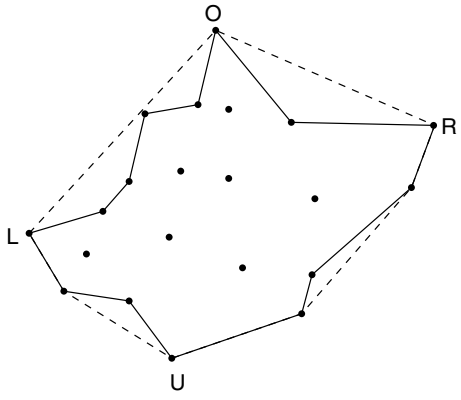


Abb. 4.10 Zur Konstruktion der konvexen Hülle der Punktmenge wird zunächst ihr Konturpolygon bestimmt.

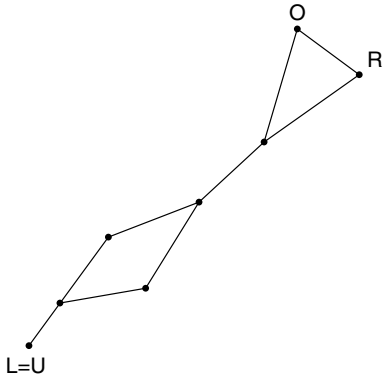


Abb. 4.11 Ein Konturpolygon mit flachen Stellen.

Abbildung 4.11 zeigt, daß sich die beiden Konturen außer an ihren Endpunkten O und U auch an anderen Stellen berühren können; das stört uns aber nicht.

Das folgende Verfahren kann auch als formale Definition der beiden Konturen aufgefaßt werden. Zuerst sortiert man die Punkte nach aufsteigenden X-Koordinaten. Dann bewegt man
sweep eine senkrechte *sweep line* von links nach rechts über die Ebene und merkt sich in der Sweep-Status-Struktur *SSS* die Punkte *MinYSoFar* und *MaxYSoFar* mit minimaler und mit maximaler Y-Koordinate, die bisher angetroffen wurden.

Am Anfang stimmen beide Punkte mit dem linken Extrempunkt L überein, am Ende ist $MaxYSoFar = O$ und $MinYSoFar = U$. Wann immer ein neuer Punkt *MaxYSoFar*

entdeckt wird, verbindet man ihn durch eine Kante mit seinem Vorgänger; dasselbe geschieht mit *MinYSoFar*.

Auf diese Weise läßt sich die linke Kontur berechnen. Mit der rechten Seite verfährt man analog, durch einen *sweep* von rechts nach links.

Nachdem die Punkte nach ihren X-Koordinaten sortiert sind, kann man die *sweeps* in Zeit $O(n)$ ausführen: In der *SSS* sind ja nur jeweils zwei Objekte zu speichern, und ein Ereignis tritt nur ein, wenn die *sweep line* einen neuen Punkt erreicht. Das sortierte Verzeichnis der Punkte reicht daher als Ereignisstruktur aus.

Sortieren + $O(n)$ genügt

Linke und rechte Kontur zusammen nennen wir das *Konturpolygon* P der Punktmenge S; offenbar ist S in P enthalten. Bis auf die Stellen zwischen O und U, an denen sich die beiden Konturen möglicherweise berühren, ist P ein einfaches Polygon, wie folgende Übungsaufgabe zeigt:

Konturpolygon

Übungsaufgabe 4.2 Die linke und die rechte Kontur können sich nicht kreuzen.

Die vorletzte Übungsaufgabe 4.1 besagt, daß wir jetzt nur noch die konvexe Hülle von P zu bilden brauchen. Dazu nehmen wir uns jedes der Fadenstücke zwischen den vier Extrempunkten vor und ziehen es stramm. Es genügt, dieses Verfahren am Beispiel des oberen Teils der linken Kontur zwischen L und O zu demonstrieren; siehe Abbildung 4.12.

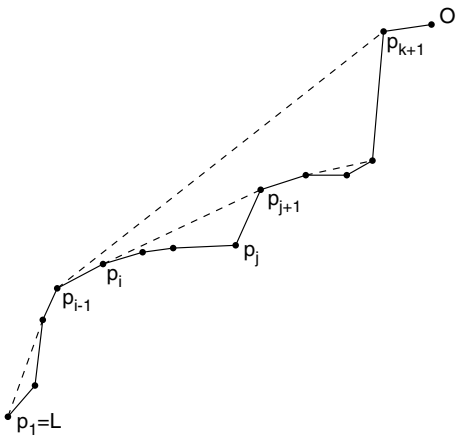


Abb. 4.12 Berechnung der konvexen Hülle am oberen Teil der linken Kontur.

Die Ecken dieses Randstücks seien von links nach rechts mit $L = p_1, p_2, \ldots$ bezeichnet. Wir durchlaufen sie in dieser Reihenfolge. Wann immer wir an eine hereinragende Ecke p_j kommen,

laufen wir so weit zurück, bis wir auf p_1 stoßen oder vorher auf einen Eckpunkt p_i, für den p_{j+1} rechts von der Geraden durch $p_{i-1}p_i$ liegt; vergleiche Abbildung 4.12. Das gesamte Randstück von p_1 bzw. p_i bis p_{j+1} wird durch das Liniensegment zwischen diesen Punkten ersetzt. Damit ist die spitze Ecke p_j verschwunden, und bis zum Punkt p_{j+1} haben wir eine konvexe polygonale Kette mit Ecken in S erzeugt, die die alte Kontur von oben umschließt.

Im Beispiel in Abbildung 4.12 wird das Liniensegment p_ip_{j+1} wieder verschwinden, wenn später $p_{i-1}p_{k+1}$ eingeführt wird.

$O(n)$ zur Berechnung von $ch(P)$

Den benötigten Zeitaufwand können wir mit einem ähnlichen Argument abschätzen, wie wir es in Abschnitt 4.1.2 für die Tangentensuche benutzt haben: Wir müssen zwar immer wieder auf dem Rand zurücklaufen, aber jeder Eckpunkt, den wir dabei passieren, wird sofort entfernt und nie wieder eingefügt. Also werden insgesamt nur $O(n)$ viele Schritte ausgeführt.

allgemeine Punktmengen

Wenn wir zulassen, daß die Punkte in S identische X- oder Y-Koordinaten haben, kann es mehrere Extrempunkte desselben Typs geben. Am Algorithmus werden hierdurch nur geringfügige Änderungen nötig. Werden zum Beispiel beim *sweep* von links nach rechts mehrere linke Extrempunkte angetroffen, verbinden wir sie alle mit senkrechten Kanten, nennen den obersten *MaxYSoFar* und den untersten *MinYSoFar*.

Damit haben wir nicht nur einen praxistauglichen Algorithmus für die Berechnung der konvexen Hülle gefunden, der auch im *worst case* optimal ist, sondern folgende Verschärfung von Theorem 4.5:

Theorem 4.7 *Die konvexe Hülle von n nach einer Koordinate sortierten Punkten in der Ebene läßt sich in Zeit $O(n)$ und linearem Speicherplatz konstruieren.*

konvexe Hülle vs. Sortieren

Im Hinblick auf Lemma 4.2 können wir festhalten: Die Berechnung der konvexen Hülle ist genauso schwierig wie das Sortieren. Hat man das eine Problem gelöst, läßt sich die Lösung des anderen in linearer Zeit daraus ableiten.

4.1.4 Der Durchschnitt von Halbebenen

Gegeben seien n Geraden $G_i, 1 \leq i \leq n$, in der Ebene; wir nehmen an, daß keine von ihnen senkrecht ist. Jedes G_i läßt sich also in der Form

$$G_i = \{(x, y) \in \mathbb{R}^2; y = a_i x + b_i\} = \{Y = a_i X + b_i\}$$

darstellen, mit reellen Zahlen a_i, b_i.

Gesucht ist der *Durchschnitt der unteren Halbebenen*

$$H_i = \{Y \leq a_i X + b_i\}.$$

Sein Rand besteht aus der unteren Kontur des Arrangements der Geraden G_i. Der Durchschnitt der H_i ist die reelle Lösungsmenge des Systems der linearen Ungleichungen

$$\begin{aligned} a_1 x + b_1 &\geq y \\ &\vdots \\ a_n x + b_n &\geq y. \end{aligned}$$

Ein wichtiges Problem namens *Lineares Programmieren* (engl. *linear programming*) besteht in der Maximierung von Zielfunktionen $f(x, y)$ mit solchen Ungleichungen als Nebenbedingungen.

Im Prinzip verfügen wir schon über die Mittel zur Berechnung solcher Durchschnitte:

Übungsaufgabe 4.3 Man gebe ein Verfahren an, mit dem sich der Durchschnitt von n Halbebenen in Zeit $O(n \log n)$ berechnen läßt. (Hinweis: *divide and conquer*)

Wir wollen hier einen interessanten Zusammenhang zu konvexen Hüllen ebener Punktmengen herausstellen, der auf einer *Dualität* zwischen Punkten und Geraden beruht.[10]

Dualität

Dazu betrachten wir die beiden Abbildungen

$$\begin{aligned} p = \quad (a, b) \quad &\longrightarrow \quad p^* = \{Y = aX + b\} \\ G^* = (-a, b) \quad &\longleftarrow \quad G = \{Y = aX + b\} \end{aligned}$$

zwischen den Punkten p der Ebene und den nicht senkrechten Geraden G. Offenbar gilt $(a, b)^{**} = (-a, b)$; für eine Dualität ist das ein kleiner Schönheitsfehler, der aber zwei nützliche Konsequenzen hat, die im folgenden aufgeführt sind. Wir benötigen sie, um den Zusammenhang zwischen Halbebenenschnitt und konvexer Hülle herzustellen.

Wir definieren für eine Gerade G und einen Punkt p den *gerichteten vertikalen Abstand* $gva(p, G)$ als die Y-Koordinate des senkrechten Vektors, der von p zu G führt; siehe Abbildung 4.13.

gerichteter vertikaler Abstand

Lemma 4.8 *Für einen Punkt p und eine Gerade G haben wir $gva(p, G) = -gva(G^*, p^*)$. Insbesondere gilt:*

$$p \text{ liegt oberhalb von } G \iff p^* \text{ verläuft oberhalb von } G^*$$

[10] Schon in Abschnitt 1.2.2 hatte uns das Dualitätsprinzip bei Graphen gute Dienste geleistet.

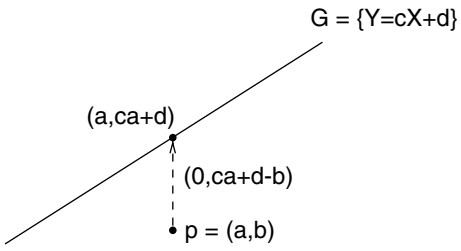


Abb. 4.13 Der gerichtete vertikale Abstand $gva(p, G)$ beträgt hier $ca + d - b$. Er ist genau dann positiv, wenn p wie eingezeichnet unterhalb von G liegt.

Beweis. Sei $p = (a, b)$ und $G = \{Y = cX + d\}$ wie in Abbildung 4.13. Dann ist $(-c, -ca + b)$ der Punkt auf p^* mit derselben X-Koordinate wie $G^* = (-c, d)$, so daß sich hier der vertikale Abstand $-ca + b - d$ ergibt. □

Aus Lemma 4.8 folgt insbesondere, daß die Inzidenz zwischen Punkten und Geraden beim Dualisieren erhalten bleibt: Es gilt

$$p \in G \Longleftrightarrow G^* \in p^* \ .$$

Die zweite nützliche Eigenschaft:

Lemma 4.9 *Seien $p = (a, b)$ und $q = (c, d)$ mit $a \neq c$. Dann gilt*

$$p^* \cap q^* = (\ell(pq))^*.$$

Hierbei bezeichnet $\ell(pq)$ die Gerade durch p und q. Man beachte, daß links der Schnittpunkt von zwei Geraden steht und rechts das Duale der Geraden $\ell(pq)$ gemeint ist.

Beweis. Sei r der Schnittpunkt der Geraden p^* und q^*. Wir nutzen den Inzidenzerhalt aus und argumentieren folgendermaßen:

$$\left.\begin{array}{l} r \in p^* \Leftrightarrow p^{**} \in r^* \\ r \in q^* \Leftrightarrow q^{**} \in r^* \end{array}\right\} \quad \begin{array}{ll} \Leftrightarrow & \ell(p^{**}q^{**}) = r^* \\ \Leftrightarrow & (\ell(p^{**}q^{**}))^* = r^{**} \\ \Leftrightarrow & (\ell(pq))^* = r \ ; \end{array}$$

dabei folgt die letzte Äquivalenz aus der Tatsache, daß die zweifache Anwendung unserer Dualität lediglich eine Spiegelung an der Y-Achse bewirkt. □

Abbildung 4.14 zeigt ein Arrangement von acht Geraden und ihre dualen Punkte. Es wurde mit *Idual* hergestellt, einem an der Universität Utrecht entwickelten System zur Visualisierung von

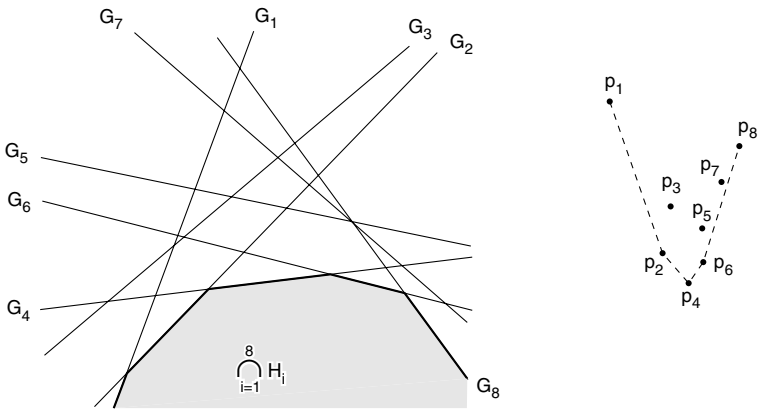


Abb. 4.14 Acht Geraden G_i und ihre dualen Punkte $p_i = G_i^*$.

Dualitäten; siehe Verkuil [140]. Der Übersichtlichkeit halber haben wir die Koordinaten weggelassen. Man sieht aber, daß die dualen Punkte um so weiter links sitzen, je steiler ihre Geraden sind.

Es fällt weiterhin auf, daß manche Geraden zum Rand des Durchschnitts der unteren Halbebene H_i keinen Beitrag leisten – und zwar genau diejenigen, deren duale Punkte nicht auf dem unteren Rand der konvexen Hülle der dualen Punktmenge liegen!

Hier besteht folgender Zusammenhang:

Theorem 4.10 *In einem Arrangement von endlich vielen, nicht senkrechten Geraden G_i ist ein Punkt $G_i \cap G_j$ genau dann ein Eckpunkt des Durchschnitts der unteren Halbebenen der G_i, wenn das Liniensegment $G_i^* G_j^*$ eine untere Kante der konvexen Hülle der dualen Punkte G_i^* ist.*

Beweis. Wie oben sei $p_i = G_i^*$. Es gilt:

$p_i p_j$ ist untere Kante der konvexen Hülle

$\Longleftrightarrow$ alle übrigen p_k liegen oberhalb der Geraden durch $p_i p_j$

$\Longleftrightarrow$ alle übrigen Geraden p_k^* verlaufen oberhalb von
$$(\ell(p_i p_j))^* = p_i^* \cap p_j^*$$

$\Longleftrightarrow$ alle übrigen G_k^{**} verlaufen oberhalb von
$$G_i^{**} \cap G_j^{**} = (G_i \cap G_j)^{**}$$

$\Longleftrightarrow$ alle übrigen G_k verlaufen oberhalb von $G_i \cap G_j$

$\Longleftrightarrow$ $G_i \cap G_j$ ist Eckpunkt des Schnitts der unteren Halbebenen.

Bei der zweiten Äquivalenz wurden Lemma 4.8 und Lemma 4.9 benutzt; bei der dritten haben wir wieder $p_i = G_i^*$ ein-

gesetzt und die triviale Tatsache verwendet, daß die Operationen Schnittpunktbildung und Spiegelung miteinander vertauscht werden können. Die vierte Äquivalenz folgt formal durch zweifache Anwendung von Lemma 4.8. □

Übungsaufgabe 4.4 Kann der Durchschnitt von endlich vielen unteren Halbebenen leer sein?

Halbebenenschnitt und konvexe Hülle

Da die Anwendung der Dualitätsabbildung auf n Geraden nur $O(n)$ Zeit erfordert und die Kanten des Halbebenenschnitts von links nach rechts in derselben Reihenfolge erscheinen wie die dualen Punkte auf der unteren konvexen Hülle, läßt sich der Durchschnitt von n Halbebenen in linearer Zeit auf die Konstruktion der konvexen Hülle von n Punkten zurückführen und umgekehrt.

Daraus ergibt sich zum einen ein weiterer optimaler Algorithmus zur Berechnung der konvexen Hülle: Übungsaufgabe 4.3 liefert ja ein Verfahren mit Laufzeit $O(n \log n)$ für den Durchschnitt von n Halbebenen. Weiterhin ergibt sich durch Dualisierung aus Lemma 4.2 auf Seite 157:

Korollar 4.11 *Die Berechnung des Durchschnitts von n Halbebenen hat die Zeitkomplexität $\Omega(n \log n)$.*

Schließlich folgt aus Theorem 4.7:

Korollar 4.12 *Der Schnitt von n unteren Halbebenen, deren Geraden nach Steigung sortiert sind, kann in Zeit $O(n)$ berechnet werden.*

Zum Schluß dieses Abschnitts über konvexe Hüllen wollen wir auf einige weiterführende Ergebnisse hinweisen.

Kirkpatrick und Seidel [85] haben einen Output-sensitiven Algorithmus angegeben, mit dem sich die konvexe Hülle von n Punkten in der Ebene sogar in Zeit $O(n \log r)$ berechnen läßt, wobei r die Anzahl der Ecken von $ch(S)$ bedeutet. Natürlich kann r viel kleiner sein als n. Werden die n Punkte zum Beispiel unabhängig nach der Gleichverteilung aus dem Einheitskreis ausgewählt, beträgt r im Mittel nur $O(n^{1/3})$, wie Raynaud [121] gezeigt hat.

höhere Dimensionen

Die konvexe Hülle von n Punkten im $\mathbb{R}^d$ läßt sich für jede feste Dimension $d > 3$ nach Chazelle [24] im *worst case* und nach Seidel [133] im Mittel über alle Einfügereihenfolgen in Zeit $O(n^{\lfloor d/2 \rfloor})$ berechnen. Das ist optimal, weil der Rand der konvexen Hülle aus so vielen Stücken bestehen kann. Bei unabhängiger, gleichverteilter Wahl von n Punkten aus der d-dimensionalen Einheitskugel hat ihre konvexe Hülle eine mittlere Komplexität in $O(n^{\frac{d-1}{d+1}})$ — sie ist also noch nicht einmal linear!

Wie sieht es im $\mathbb{R}^3$ aus? Dort ist die konvexe Hülle einer n-elementigen Punktmenge S ein *konvexes Polyeder* mit Ecken in S, an denen sich mindestens drei Kanten treffen. Nach Übungsaufgabe 1.6 auf Seite 18 hat dann der aus den Kanten und Ecken bestehende Graph die Komplexität $O(n)$.

Erstaunlicherweise läßt sich auch im $\mathbb{R}^3$ die konvexe Hülle von n Punkten in Zeit $O(n \log n)$ berechnen. *Worst-case*-effiziente Verfahren findet man zum Beispiel bei Preparata und Shamos [120]. Aber auch das in Abschnitt 4.1.2 beschriebene randomisierte inkrementelle Verfahren läßt sich leicht auf den $\mathbb{R}^3$ übertragen, wovon wir uns jetzt überzeugen wollen.

Übungsaufgabe 4.5 Zeigen Sie, daß sich die konvexe Hülle von n Punkten im $\mathbb{R}^3$ mit dem randomisierten inkrementellen Verfahren mit einer mittleren Laufzeit von $O(n \log n)$ und in linearem Speicherplatz berechnen läßt.

4.2 Triangulieren eines einfachen Polygons

In diesem Abschnitt beschäftigen wir uns mit Zerlegungen von Polygonen in Dreiecke. Sei P ein einfaches Polygon mit n Ecken. Eine *Diagonale* von P ist ein Liniensegment zwischen zwei Ecken, das in P enthalten ist, aber mit dem Rand von P nur seine Endpunkte gemein hat. In Abbildung 4.15 (i) ist also d eine Diagonale, f und g sind keine Diagonalen. Eine Diagonale zerlegt das Polygon immer in zwei echte Teilpolygone.

Diagonale

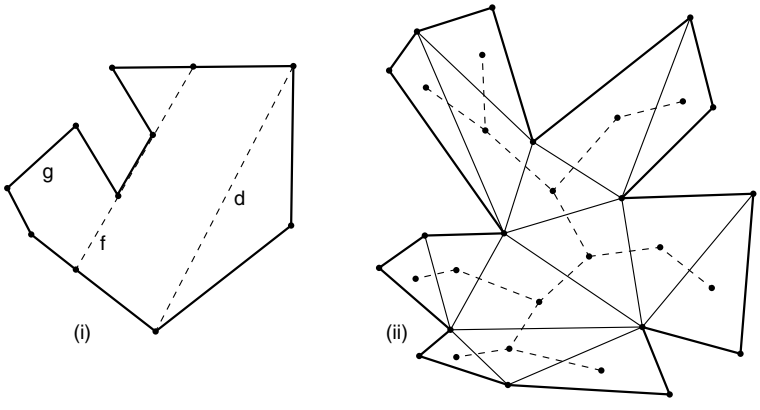


Abb. 4.15 In (i) ist nur d eine Diagonale. Figur (ii) zeigt eine Triangulation und ihr Duales.

zusätzliche Annahme

In diesem Abschnitt nehmen wir der Einfachheit halber an, daß keine drei Eckpunkte des Polygons kolinear sind. Dann kann z. B. das in Abbildung 4.15 (i) gezeigte Polygon nicht auftreten.

Existenz von Diagonalen

Zunächst ist gar nicht so klar, ob es in einem Polygon überhaupt Diagonalen gibt. Das folgende Lemma gibt darauf eine Antwort.

Übungsaufgabe 4.6 Kann man zu jedem Eckpunkt eines einfachen Polygons eine Diagonale finden?

Lemma 4.13 *Ist P konvex, so ist jedes Liniensegment zwischen zwei nicht direkt benachbarten Ecken eine Diagonale. Ist P nicht konvex und v eine beliebige spitze Ecke von P, gibt es eine Diagonale mit Eckpunkt v.*

Beweis. Nur die letzte Behauptung bedarf einer Begründung. Sei also v eine spitze Ecke von P, d. h. eine Ecke mit Innenwinkel größer π, und sei l eine Tangente an den Rand von P im Punkt v, wie in Abbildung 4.16 dargestellt; dort liegen die Nachbarecken v_1 und v_2 von v unterhalb von l. Jetzt betrachten wir den von v aus sichtbaren Teil des Randes von P oberhalb von l. Er kann nicht nur aus einer Kante bestehen. Wenn wir also von v aus zunächst nach rechts schauen und dann die Blickrichtung gegen den Uhrzeigersinn drehen, wird entweder die Sicht auf die aktuelle Kante an einer spitzen Ecke wie w' abreißen, oder wir werden den Endpunkt der aktuellen Kante erreichen, wie etwa den Punkt w. In jedem Fall gibt es oberhalb von l einen Eckpunkt x von P, der von v aus sichtbar ist und der mit v eine Diagonale bildet. □

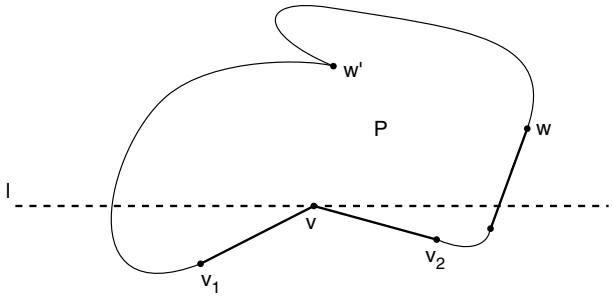


Abb. 4.16 Der Rand von P oberhalb von l kann nicht nur aus einer einzelnen Kante bestehen. Also enthält er eine von v sichtbare Ecke wie w oder w'.

Triangulation

Unter einer *Triangulation* des Polygons P verstehen wir eine maximale Menge sich nicht kreuzender Diagonalen von P, zusammen mit ∂P; Abbildung 4.15 (ii) zeigt ein Beispiel.

Lemma 4.14 *Jedes einfache Polygon P kann trianguliert werden.*

Beweis. Ist P ein Dreieck, gibt es nichts zu tun. Andernfalls stellt Lemma 4.13 sicher, daß P mindestens eine Diagonale enthält; sie zerlegt das Polygon P in zwei Teilpolygone, von denen jedes weniger als n Ecken hat. Per Induktion besitzt jedes der Teilpolygone eine Triangulation, so daß wir insgesamt eine Triangulation von P erhalten. □

Das Applet `http://www.geometrylab.de/Triangulation/` zeigt, wie ein beliebiges Polygon trianguliert werden kann.

Natürlich kann ein Polygon in der Regel auf verschiedene Weisen trianguliert werden. Die Anzahl der Dreiecke und Diagonalen ist aber stets dieselbe.

Übungsaufgabe 4.7 Man zeige, daß jede Triangulation eines einfachen Polygons mit n Ecken genau $n-2$ Dreiecke und $n-3$ Diagonalen besitzt.

Wenn wir im Innern eines jeden Dreiecks einer Triangulation T einen Punkt auswählen und diejenigen Punkte miteinander verbinden, deren Dreiecke eine gemeinsame Kante haben, erhalten wir einen Graphen T^*, der bis auf den fehlenden Knoten im Äußeren von P der *duale Graph* der Triangulation T ist; vgl. Abbildung 4.15 (ii). dualer Graph

Lemma 4.15 *Der Graph T^* ist ein Baum mit Knotengrad ≤ 3.*

Beweis. Weil jedes Dreieck höchstens drei Nachbardreiecke haben kann, ist der Grad eines jeden Knotens durch 3 beschränkt. Die Dreiecke von T füllen P lückenlos aus. Folglich ist T^* zusammenhängend.

Bleibt zu zeigen, daß T^* keinen Zyklus enthält, d. h. keinen einfachen geschlossenen Weg. Dazu überlegen wir uns, daß es in T Dreiecke gibt, von deren Kanten *nur eine* zu den Diagonalen gehört, also zwei zum Rand von P gehören. Solche Dreiecke heißen *Ohren*, und ihre Existenz wird vom Zwei-Ohren-Satz garantiert: Ohren

Theorem 4.16 *In jeder Triangulation eines einfachen Polygons mit $n \geq 4$ Ecken gibt es mindestens zwei Dreiecke, deren Rand höchstens eine Diagonale enthält.*

Beweis. Angenommen, von den Dreiecken einer Triangulation sind a Stück von einer, b Stück von zwei und c Stück von drei Diagonalen berandet. Nach Übungsaufgabe 4.7 folgt daraus

$$\begin{aligned} a+b+c &= n-2 \\ a+2b+3c &= 2(n-3)\,, \end{aligned}$$

denn jede Diagonale berandet zwei Dreiecke. Zieht man die obere Gleichung von der unteren ab, ergibt sich

$$b + 2c = n - 4 = a + b + c - 2 \,,$$

also $a = c + 2 \geq 2$. □

Nun weiter mit dem Beweis von Lemma 4.15! Sei also D ein Dreieck mit nur einer Diagonalen d als Kante. Wir entfernen es und erhalten eine Triangulation T'. Per Induktion ist T' zyklenfrei und bleibt es auch, wenn wir den Knoten im Dreieck D als neues Blatt wieder anfügen! □

Übungsaufgabe 4.8 Läßt sich jedes einfache Polygon so triangulieren, daß jedes Dreieck höchstens zwei Nachbardreiecke hat?

Daß sich Polygone triangulieren lassen, ist oft ein nützliches Beweishilfsmittel. Betrachten wir zum Beispiel die Drehungen,

Schildkröte

die eine *Schildkröte* ausführen müßte, während sie sich gegen den Uhrzeigersinn auf dem Rand eines Polygons entlangbewegt.[11] Die Kanten werden von einer beliebigen Startkante aus mit $e_0, e_1, \ldots, e_{n-1}$ durchnumeriert. Mit $\alpha_{i,i+1}$ wird die Drehung bezeichnet, die für den Übergang von e_i nach e_{i+1} erforderlich ist;

Drehsinn

siehe Abbildung 4.17. Dabei werden Linksdrehungen positiv und Rechtsdrehungen negativ gezählt.

Für $i \neq k$ bezeichnen wir mit

$$\alpha_{i,k} = \alpha_{i,i+1} + \alpha_{i+1,i+2} + \ldots + \alpha_{k-1,k}$$

die Summe der Drehwinkel von Kante e_i bis zur Kante e_k. Die Winkelsumme bei *einem vollständigen Umlauf* von e_i bis e_i wird

Gesamtdrehung

auch die *Gesamtdrehung* genannt und mit $\alpha_{i,i}$ bezeichnet.[12] Es gilt folgende schöne Aussage:

Lemma 4.17 *Sei P ein einfaches Polygon und e_i eine beliebige Kante von P. Dann ist $\alpha_{i,i} = 2\pi$.*

Beweis. Durch Induktion über die Kantenzahl $n \geq 3$. Sei $n = 3$. Da sich der äußere Drehwinkel $\alpha_{j,j+1}$ und der Innenwinkel β_j jeweils zu π aufaddieren (vgl. Abbildung 4.17), folgt

$$\alpha_{i,i} = \sum_{j=0}^{2} \alpha_{i+j,i+j+1} = \sum_{j=0}^{2} (\pi - \beta_{i+j}) = 3\pi - \sum_{j=0}^{2} \beta_{i+j} = 2\pi,$$

[11] Um diese *turtle geometry* geht es in dem gleichnamigen Buch von Abelson und diSessa [1]. In der Programmiersprache LOGO gibt es eine *turtle*, die durch Vorwärtsbewegungen und Drehungen gesteuert wird.

[12] Achtung! Die Indizes $i, j, \ldots$ werden $\bmod n$ gezählt, aber wir zählen *nicht* die Drehwinkel $\bmod 2\pi$!

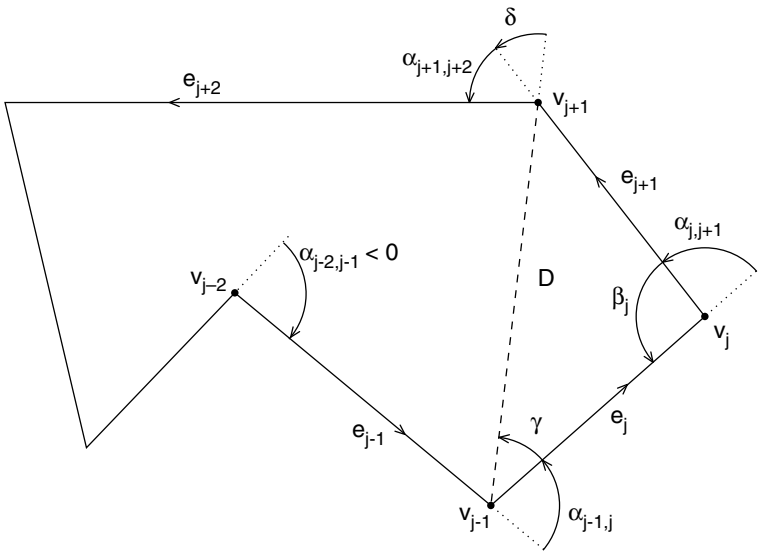


Abb. 4.17 Mit $\alpha_{j,j+1}$ wird der Winkel der Drehung bezeichnet, die man bei Verfolgung des Randes von P beim Übergang von Kante e_j zu e_{j+1} ausführen muß.

denn die Summe der Innenwinkel beträgt beim Dreieck bekanntlich π.

Hat das Polygon mehr als drei Ecken, wählen wir eine Diagonale, die ein Dreieck D vom Rest des Polygons abtrennt, und wenden die Induktionsvoraussetzung auf den Rest an. Dann fügen wir das Dreieck D wieder hinzu und betrachten die Änderung in der Winkelsumme, die sich hierdurch ergibt; mit den Bezeichnungen von Abbildung 4.17 erhöht sie sich gerade um

$$\alpha_{j,j+1} - (\gamma + \delta) = \pi - (\beta_j + \gamma + \delta) = 0. \qquad \square$$

Übungsaufgabe 4.9 Gibt es auch Polygone, die nicht einfach sind (also Selbstschnitte aufweisen) und trotzdem Lemma 4.17 erfüllen?

Ein ähnliches auf Drehwinkeln beruhendes Kriterium erlaubt uns zu entscheiden, ob ein Punkt p im inneren oder im äußeren Gebiet eines einfachen Polygons P liegt; dabei nehmen wir an, daß p nicht gerade zum Rand von P gehört. Für zwei Punkte $x, y \in \partial P$ sei $\alpha_{x,y}$ der Winkel, um den sich ein Betrachter im Punkt p insgesamt dreht, während er verfolgt, wie sich der Punkt z längs ∂P entgegen dem Uhrzeigersinn von x nach y bewegt, siehe

Abbildung 4.18[13]. Mit α_x bezeichnen wir den Gesamtdrehwinkel einer vollen Umdrehung ab x.

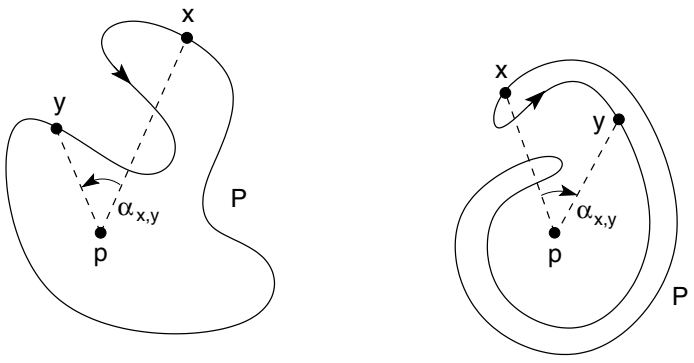


Abb. 4.18 Der Drehwinkel $\alpha_{x,y}$ beim Marsch von x nach y in Bezug auf P.

Übungsaufgabe 4.10 Seien P ein einfaches Polygon, $p \notin \partial P$ und $x \in \partial P$. Dann ist $\alpha_x = 0$, falls p außerhalb von P liegt und $\alpha_x = 2\pi$, falls p im Innern von P liegt.

triangulieren: wie schnell?

Eine interessante, aber sehr schwierige Frage lautet: Wie schnell kann ein einfaches Polygon mit n Ecken trianguliert werden? Wie wir in Abschnitt 4.3 sehen werden, läßt sich das Sichtbarkeitspolygon $vis(v)$ eines jeden Punktes $v \in P$ in Zeit $O(n)$ berechnen. Der Beweis von Lemma 4.13 zeigt, wie sich aus dem Sichtbarkeitspolygon in linearer Zeit eine Diagonale von P gewinnen läßt. Dies liefert also ein rekursives Verfahren mit Laufzeit $O(n^2)$. Der Algorithmus hätte sogar die Laufzeit $O(n \log n)$, wenn man die Diagonale stets so wählen könnte, daß die beiden Teilpolygone ungefähr gleich viele Eckpunkte haben; solche Diagonalen existieren zwar immer, sie sind aber nicht so leicht zu finden.

Jedes *konvexe* Polygon ist natürlich in Zeit $O(n)$ triangulierbar: Man startet mit einer beliebigen Ecke v und verbindet sie nacheinander mit allen übrigen, die nicht schon direkt zu v benachbart sind. Eine Verallgemeinerung der konvexen Polygone stellen die *monotonen* Polygone dar: Ein Polygon P heißt monoton, wenn es eine Gerade l gibt, so daß jede Senkrechte zu l den Rand von P in höchstens zwei Punkten schneidet.

monotones Polygon

Auch für die Klasse der monotonen Polygone läßt sich ein relativ einfacher linearer Triangulierungsalgorithmus angeben; er

[13] Das Polygon P ist nur der Übersichtlichkeit halber durch eine glatte Kurve dargestellt.

wird zum Beispiel bei O'Rourke [113] erklärt. Weil es ein Verfahren gibt, mit dem sich jedes beliebige einfache Polygon in Zeit $O(n \log n)$ in monotone Polygone zerlegen läßt, erhält man auf diese Weise ein Triangulierungsverfahren mit Laufzeit $O(n \log n)$.

Wieweit sich diese Schranke verbessern läßt, hat die Forschung mehrere Jahre lang beschäftigt. Eine abschließende Antwort konnte Chazelle [25] geben. Er zeigte, daß ein einfaches Polygon mit n Ecken in Zeit $O(n)$ trianguliert werden kann.

$O(n)$ ist optimal

Ein weitaus praktikableres Verfahren stammt von Seidel [132]; es verwendet die Technik der randomisierten inkrementellen Konstruktion, wie wir sie in Abschnitt 4.1.2 für die konvexe Hülle kennengelernt haben. Der zu erwartende Zeitbedarf ist in $O(n \log^* n)$, also fast linear.[14]

$O(n \log^* n)$ ist praktikabel

Daß das Triangulierungsproblem schwierig ist, zeigt sich auch bei einem abschließenden Blick in den $\mathbb{R}^3$. Dort geht es in Analogie zur Ebene darum, ein Polyeder mit n Ecken in Tetraeder zu zerlegen.

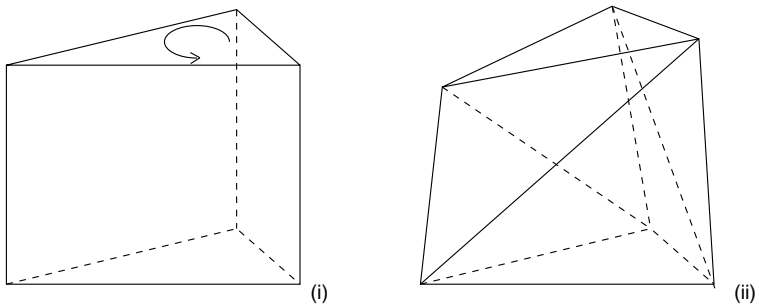


Abb. 4.19 Nach Verdrehen des Prismas ist keine Zerlegung in Tetraeder mehr möglich, weil die untere Dreiecksfläche von keiner der oberen drei Ecken ganz sichtbar ist.

Die erste Überraschung: Diese Aufgabe ist nicht immer lösbar! Ein auf Schönhardt zurückgehendes Beispiel ist in Abbildung 4.19 gezeigt. Es handelt sich um ein Prisma, also um zwei parallele kongruente Dreiecke, deren Ecken durch parallele Kanten verbunden sind. Dreht man nun das obere Dreieck unter leichtem Druck linksherum, entstehen an den Längsseiten die in (ii) gezeigten diagonalen Knicklinien, die ins Innere ragen. Man muß zusätzliche Punkte als Ecken erlauben, um eine Zerlegung in Tetraeder möglich zu machen.

Überraschungen im $\mathbb{R}^3$

Tetraeder-Zerlegung nicht immer möglich

[14] Zur Definition von $\log^* n$ siehe Seite 85.

Anzahl der Tetraeder nicht eindeutig

Die zweite Überraschung besteht darin, daß auch in den Fällen, in denen eine Tetraeder-Zerlegung möglich ist, die Anzahl der Tetraeder nicht mehr eindeutig bestimmt ist! Abbildung 4.20 zeigt ein konvexes Polyeder mit fünf Ecken und sechs Dreiecksflächen, das links in zwei und rechts in drei Tetraeder zerlegt ist. In (i) separiert das Dreieck $tria(a, b, c)$ zwei Tetraeder; in (ii) bilden die Ecken a, b, d, e ein Tetraeder, auf dessen vom Betrachter abgewandten Flächen $tria(a, e, d)$ und $tria(b, d, e)$ zwei weitere Tetraeder sitzen, die von $tria(c, d, e)$ separiert werden.

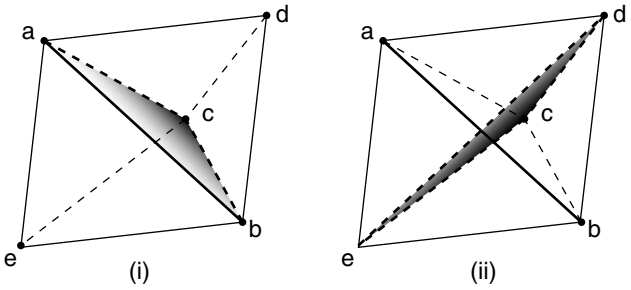


Abb. 4.20 Zwei verschiedene Tetraeder-Zerlegungen eines konvexen Polyeders.

4.3 Konstruktion des Sichtbarkeitspolygons

Sichtbarkeitspolygon

Auf Seite 20 hatten wir das *Sichtbarkeitspolygon* $vis(p)$ eines Punktes p in einem einfachen Polygon P als den Teil von P definiert, der von p aus sichtbar ist. Ein Beispiel ist in Abbildung 4.21 gezeigt. Hier und im folgenden stellen wir den Rand der Polygons als glatte Kurve dar, weil die topologischen Zusammenhänge dann klarer hervortreten.

Ziel: Berechnung von $vis(p)$

Unser Ziel besteht darin, zu gegebenem einfachem Polygon P und einem Punkt p in P das Sichtbarkeitspolygon von p effizient zu berechnen. Ein Motiv für diese Frage stammt aus dem Bereich der Bewegungsplanung für Roboter; man möchte wissen, was ein Roboter von einem Punkt p aus sehen kann, um zum Beispiel an geeigneten Stellen Orientierungsmarken anzubringen.

Orientierung: gegen den Uhrzeiger

Wir gehen davon aus, daß der Rand ∂P von P uns als zyklisch verkettete Liste von n Liniensegmenten gegeben ist. Dabei wird die Orientierung *gegen den Uhrzeigersinn* zugrunde gelegt; die von einem Punkt $p \in P$ aus sichtbaren Teile des Randes von P sind also alle *linksherum* orientiert.

Um $vis(p)$ zu konstruieren, genügt es offenbar, die Folge derjenigen Kanten auf dem Rand von $vis(p)$ zu bestimmen, die auch

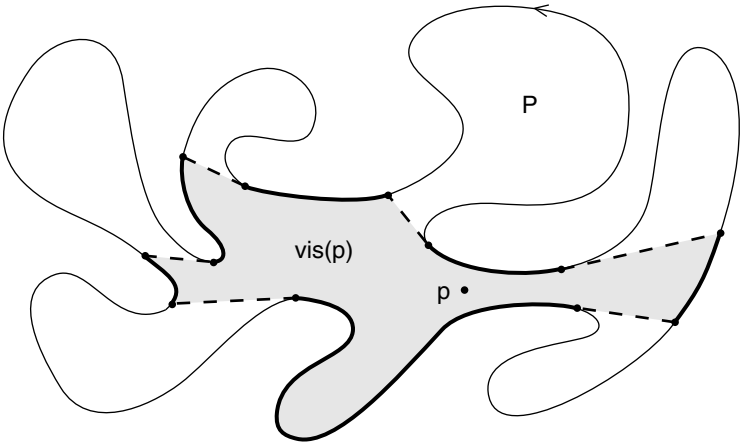


Abb. 4.21 Das Sichtbarkeitspolygon $vis(p)$ des Punktes p im Polygon P.

(Teile von) Kanten von P sind; in Abbildung 4.21 sind das genau die dick gezeichneten Stücke. Die gestrichelten Liniensegmente – *künstliche Kanten* von $vis(p)$ genannt – lassen sich dann leicht ergänzen; sie sind Teile von Tangenten von p an den Rand von P.

künstliche Kante

Jede künstliche Kante von $vis(p)$ zerlegt das Polygon P in zwei Teile: einen von p aus nicht einsehbaren und einen, der den Punkt p enthält. Die nicht einsehbaren Teile heißen auch *Höhlen*. Sie sind in Abbildung 4.21 weiß dargestellt. Der Durchschnitt derjenigen Teile, die p enthalten, ergibt gerade $vis(p)$; aber zur Konstruktion des Sichtbarkeitspolygons eignet sich diese Tatsache nicht.

Höhle

Was läßt sich im voraus über die Zeitkomplexität des Problems sagen? Zumindest muß unser Algorithmus alle Kanten von $vis(p)$ ausgeben. Da jede Kante von P hierzu einen Beitrag leisten kann, ist $\Omega(n)$ eine untere Schranke. Andererseits hatten wir in Korollar 2.16 auf Seite 86 festgestellt, daß die *untere Kontur* von n *beliebigen* Liniensegmenten in der Ebene sich in Zeit $O(\lambda_3(n) \log n)$ berechnen läßt. Diese obere Schranke gilt auch für den vorliegenden Fall; vgl. Übungsaufgabe 2.13 auf Seite 88.

Zeitkomplexität?

Wie wir sehen werden, geht es hier effizienter, weil die n Liniensegmente eben nicht „beliebig“ sind, sondern den Rand eines einfachen Polygons bilden.

4.3.1 Der Algorithmus

Zu Beginn bestimmen wir einen Punkt *R0* auf ∂P, der *mit Sicherheit* von p aus sichtbar ist. Das geht in linearer Zeit, wie folgende Übungsaufgabe zeigt:

Übungsaufgabe 4.11 Wie bestimmt man in Zeit $O(n)$ einen Punkt auf dem Rand eines Polygons mit n Kanten, der von einem vorgegebenen Punkt im Innern sichtbar ist?

Von diesem Randpunkt *R0* aus durchlaufen wir nun ∂P gegen den Uhrzeigersinn. Dabei wird ein Stapel S unterhalten, der Stücke des schon besuchten Teils von ∂P enthält, die *möglicherweise* von p aus sichtbar sind. Wenn der aktuelle Randpunkt *RAkt* wieder bei *R0* angekommen ist, enthält S *genau* die von p aus sichtbaren Teile von ∂P. Nach Einfügen der künstlichen Kanten können wir dann das Sichtbarkeitspolygon $vis(p)$ ausgeben. Unser Vorgehen sieht also im großen folgendermaßen aus:

top-down-Beschreibung

```
type
    tDreh = (links, rechts);
    tRandP = (* Punkt auf dem Rand ∂P von P *);
var
    Drehrichtung : tDreh (* in Bezug auf p *);
    R0           : tRandP (* Startpunkt *);
    RAkt         : tRandP (* aktueller Randpunkt *);
    S            : Stapel;
bestimme einen von p aus sichtbaren Randpunkt R0;
initialisiere Stapel S mit R0;
RAkt := R0;
Drehrichtung := links;
repeat
    RandDurchlauf(RAkt) (* gegen den Uhrzeigersinn *)
until RAkt = R0;
verbinde die in S gespeicherten Segmente
    mit künstlichen Kanten;
gib das Sichtbarkeitspolygon vis(p) aus
```

Abbildung 4.22 zeigt zwei typische Modi, in denen wir uns während des Durchlaufs befinden können. In (i) bewegt sich der aktuelle Punkt *RAkt* gerade links um den Punkt p herum, und das durchlaufene Stück des Randes wird als möglicherweise von p aus sichtbar in den Stapel S eingefügt.[15]

Drehrichtung links: einfügen

[15] Auch wenn wir anfangs bei *R0* starten, ist die Drehrichtung links.

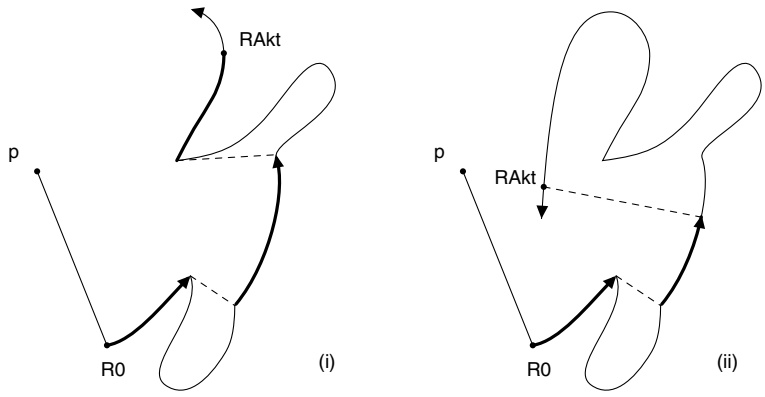


Abb. 4.22 Die dick eingezeichneten Stücke sind zur Zeit im Stapel S gespeichert. In (i) dreht sich *RAkt* links um p herum, und der durchlaufene Teil des Randes wird in S eingefügt. Bei einer Rechtsdrehung um p, wie in (ii) gezeigt, werden Randstücke aus S entfernt.

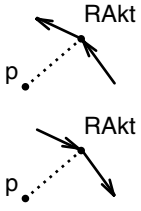
Im weiteren Verlauf *wendet* sich dann aber die Randkurve bezüglich ihrer Drehrichtung um p: In (ii) beschreibt *RAkt* eine Rechtsdrehung um p und verdeckt damit die Sicht auf die letzten Randstücke in S, die nun wieder entfernt werden müssen. Man beachte, daß zu diesem Zeitpunkt der Punkt *RAkt nicht* von p aus sichtbar ist, weil das Liniensegment zwischen ihnen den Randpunkt *RAkt* ja *von außen* erreicht. Deshalb wird der aktuelle Teil der Kurve nicht in den Stapel eingefügt.

Drehrichtung rechts: löschen

Die folgende Prozedur *RandDurchlauf* unterhält den Stapel S. Sie terminiert immer dann, wenn ein *Ereignis* eintritt wie etwa eine Änderung der Drehrichtung. Welche Ereignisse außerdem vorkommen können, diskutieren wir anschließend.

Ereignis

```
procedure RandDurchlauf(var RAkt: tRandP);
    repeat
        rücke RAkt gegen den Uhrzeiger auf ∂P vor;
        if Drehrichtung = links
        then füge durchlaufenes Randstück in S ein
        elseif Drehrichtung = rechts
        then lösche von durchlaufenem Randstück
                    verdeckte Stücke in S
    until Ereignis(RAkt) (* vgl. Abbildung 4.22 *);
```



Wende nach innen

Ein Wechsel der Drehrichtung wird dadurch ausgelöst, daß die Randkurve sich – von p aus gesehen – nach innen, also zu p hin, wendet. Dies ist das erste Ereignis, das von der Funktion *Ereignis* behandelt wird.

Wende nach außen

Es kann aber auch vorkommen, daß sich die Randkurve nach außen, also von p fort, wendet. Geschieht das aus einer Linksdrehung heraus, sprechen wir von einer *Rechtswende*. Hat sich dagegen der aktuelle Randpunkt vorher rechtsherum um p gedreht, findet eine *Linkswende* statt. Diese beiden Ereignisse sind in Abbildung 4.23 dargestellt.

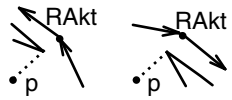
schneller Vorlauf

In jedem Fall ist die Randkurve ab dem Eckpunkt Q unsichtbar, bis sie dahinter wieder zum Vorschein kommt. Das Stück dazwischen ist deshalb für uns nicht interessant. Wir begeben uns also in den dritten möglichen Modus und führen einen *schnellen Vorlauf* durch bis zu dem Punkt, an dem der Rand frühestens wieder sichtbar werden kann. In Abbildung 4.23 (i) ist das ein Punkt auf dem Strahl, der in Q startet und vom Punkt p wegführt, in (ii) muß dieser Punkt zwischen Q und dem ersten Randpunkt R liegen, den der Strahl zur Zeit trifft.

Während des schnellen Vorlaufs wird der Stapel S nicht verändert. Sobald der Rand hinter der Ecke Q wieder zum Vorschein kommt, wird die ursprüngliche Drehrichtung fortgesetzt.

```
function Ereignis(var RAkt: tRandP): boolean;
    var Q, R: tRandP;
    Ereignis := false;
    if WendeNachInnen(RAkt)
    then
        Ereignis := true;
        Umkehr(Drehrichtung);
    if RechtsWendeNachAußen(RAkt)
    then
        Ereignis := true;
        Q := RAkt;
        SchnellerVorlauf(RAkt, Q, ∞); (* Abbildung 4.23 (i) *)
    if LinksWendeNachAußen(RAkt)
    then
        Ereignis := true;
        Q := RAkt;
        R := der vom Strahl von p durch RAkt getroffene
             Randpunkt in S;
        SchnellerVorlauf(RAkt, Q, R); (* Abbildung 4.23 (ii) *)
```

if *WirdVerdeckt*(*RAkt*)
then
Ereignis := *true*;
Q := *RAkt*;
R := der Endpunkt des in S gespeicherten Randstücks,
hinter dem ∂P bei Q verschwindet;
SchnellerVorlauf(*RAkt*, Q, R);
Umkehr(*Drehrichtung*); (* Abbildung 4.24 *)



if *RAkt* = *R0*
then *Ereignis* := *true*;

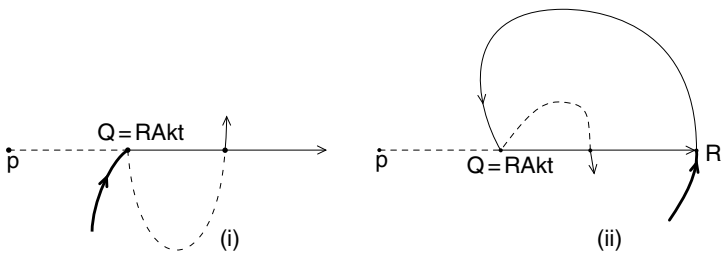


Abb. 4.23 Eine Rechtswende (i) und eine Linkswende (ii) nach außen. Der Rand des Polygons verschwindet hinter dem Eckpunkt Q.

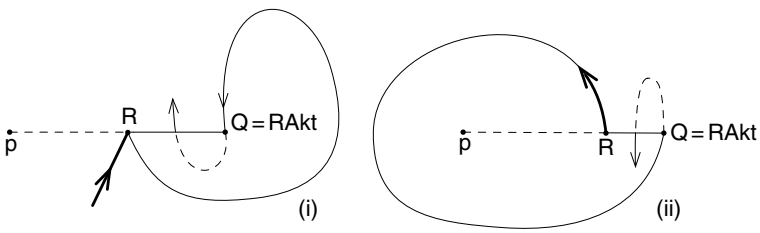


Abb. 4.24 Bei *RAkt* verschwindet der Rand hinter dem Punkt R.

Rand verschwindet

Ein weiteres mögliches Ereignis liegt vor, wenn der aktuelle Randpunkt *RAkt* hinter Teilen des Randes verschwindet, die im Stapel S gespeichert sind; siehe Abbildung 4.24. Auch hier findet ein schneller Vorlauf statt, bis der Rand bei Überschreiten des Liniensegments von R nach Q hinter dem Eckpunkt R hervorkommt. Die Drehrichtung kehrt sich bei diesem Ereignis um.

Das letzte Ereignis findet statt, wenn *RAkt* den Startpunkt *R0* wieder erreicht; hier terminiert das ganze Verfahren.

Vorsicht beim Vorlauf!

Beim Anblick der Abbildungen ist man versucht, die Prozedur *SchnellerVorlauf* so zu implementieren: Laufe bis zum nächsten

Punkt, bei dem der Polygonrand den Strahl von Q bzw. das Liniensegment von Q nach R „von der nicht sichtbaren Seite her“ überquert. Abbildung 4.25 zeigt am Beispiel einer Rechtswende nach außen, warum das nicht funktioniert.

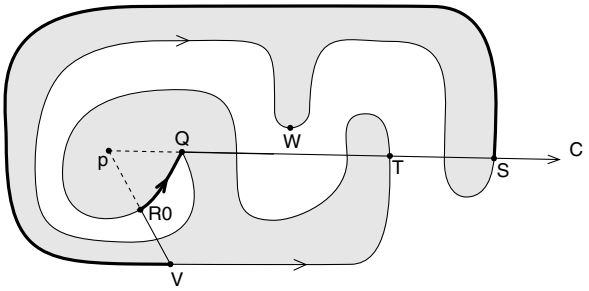


Abb. 4.25 Der Rand des Polygons kommt erst im Punkt T wieder hinter dem Eckpunkt Q hervor, aber noch nicht in S.

so nicht!

Am Punkt S kreuzt die Randkurve zum ersten Mal seit Q den Strahl C von unten nach oben. Würde unser Vorlauf dort schon enden, käme der Algorithmus anschließend völlig durcheinander: Er würde im Punkt V vergeblich darauf warten, daß die Randkurve wieder das Segment von *R0* nach V überquert.[16] Alle Randstücke, die zu diesem Zeitpunkt im Stapel S gespeichert wären, sind in Abbildung 4.25 dick eingezeichnet.

Tatsächlich muß man bis zum Punkt T vorlaufen; erst dort kommt der Rand hinter Q zum Vorschein. Der Unterschied zwischen T und S läßt sich folgendermaßen präzisieren: Das Randstück im Umlaufsinn von Q nach T kann man, ohne es über p hinwegzuheben, so deformieren, daß es zusammen mit dem Liniensegment QT ein einfaches Polygon bildet, welches p nicht enthält. Mit dem Randstück von Q nach S geht das nicht: Zusammen mit dem Liniensegment QS umschließt es den Punkt p.

Winkelzähler benutzen

Diesen Umstand kann man sich wegen Übungsaufgabe 4.10 zunutze machen, um zwischen S und T zu unterscheiden. Nach Passieren von Q verfolgen wir, wie sich der Punkt *RAkt* um p dreht. Zu diesem Zweck wird eine Variable *Winkelzähler* mitgeführt, die im Punkt Q auf null gesetzt wird und ab dort den absoluten Drehwinkel des laufenden Punktes *RAkt* in bezug auf den festen Punkt p als Wert hat. Natürlich hat *Winkelzähler* ganzzahlige Vielfache von 2π als Wert, wann immer *RAkt* den Strahl C kreuzt.

Im Punkt S hat insgesamt eine Drehung um -2π stattgefunden; dagegen ist T der erste Punkt auf dem Rand, der auf dem

[16] Die Situation entspricht der in Abbildung 4.24 (ii) gezeigten.

Strahl liegt und bei dem die Gesamtdrehung null ist. Hier wird jetzt der Vorlauf beendet.

Wir können deshalb die Prozedur *SchnellerVorlauf* folgendermaßen implementieren:[17]

```
procedure SchnellerVorlauf(var RAkt: tRandP; Q, R: tRandP);
    var Winkelzähler: real; (* zählt Drehwinkel um p *)
    Winkelzähler := 0;
    repeat
        rücke RAkt vor;
        aktualisiere Winkelzähler
    until RAkt auf QR and Winkelzähler = 0;
```

Korrektheit

Wir wollen uns jetzt klarmachen, warum dieser Algorithmus korrekt funktioniert. Man könnte meinen, daß der Stapel S zu jedem Zeitpunkt alle Teile des bisher betrachteten Randes Δ von Startpunkt *R0* zum aktuellen Punkt *RAkt* enthält, deren Sicht auf p jedenfalls nicht durch Δ selbst blockiert ist. Diese Vermutung ist aber falsch!

Übungsaufgabe 4.12 Man gebe ein Beispiel an, in dem der Stapel S vor Beendigung des Verfahrens Randstücke enthält, die durch Teile des schon durchlaufenen Randes verdeckt werden.

Stapel enthält alle sichtbaren Randpunkte

keine Überlappungen

Richtig ist nur, daß nicht zuviel aus dem Stapel entfernt wird: Nur solche Randstücke werden aus S wieder entfernt oder gar nicht erst aufgenommen, die von p aus nicht zu sehen sind. Außerdem ist durch den Algorithmus sichergestellt, daß sich zu jedem Zeitpunkt die in S gespeicherten Randstücke nicht überlappen. Aus diesen beiden Aussagen folgt die Korrektheit des Verfahrens:

Lemma 4.18 *Wenn der Algorithmus zur Berechnung von vis(p) terminiert, enthält der Stapel S genau diejenigen Stücke des Randes von P, die von p aus sichtbar sind.*

Beweis. Wir brauchen nur noch zu zeigen, daß S keine unsichtbaren Stücke enthält. Sei also T ein Randpunkt, der von p nicht sichtbar ist. Blickt man von p aus in Richtung von T, sieht man einen Randpunkt $R \neq T$. Dieser Punkt R ist im Stapel gespeichert. Weil sich die gespeicherten Randstücke nicht überlappen können, kann nicht auch T im Stapel S vorhanden sein. □

Daß der Algorithmus mit linearer Rechenzeit auskommt, liegt auf der Hand; schließlich wird der Rand des Polygons ja nur einmal umlaufen. Damit haben wir folgendes Resultat:

[17] Im Falle $R = \infty$ bezeichne QR den Strahl, der in Q beginnt und sich von p längs der Geraden durch p und Q entfernt.

Theorem 4.19 *Das Sichtbarkeitspolygon eines Punktes p in einem einfachen Polygon P mit n Kanten läßt sich in optimaler Zeit $\Theta(n)$ und in linearem Speicherplatz bestimmen.*

Der hier vorgestellte Algorithmus geht auf eine Arbeit von Lee [90] zurück, die von Joe und Simpson [83] verbessert wurde. Bei Joe [82] findet sich ein formaler Korrektheitsbeweis.

Unser korrekter Algorithmus wurde im Applet

`http://www.geometrylab.de/VisPolygon/`

implementiert.

4.3.2 Verschiedene Sichten im Inneren eines Polygons

Sicht eines beweglichen Punktes

Für Anwendungen in der Robotik möchte man gerne wissen, wie das Sichtbarkeitspolygon $vis(p)$ sich ändert, wenn der Punkt p sich im Polygon P bewegt. Wenn p sich nur wenig bewegt, wird im allgemeinen von einigen Kanten des Polygons mehr, von anderen weniger zu sehen sein, aber die Menge der *sichtbaren Ecken* ändert sich zunächst einmal nicht. Das geschieht erst, wenn p die Verlängerung einer Tangente von einer Ecke e an eine spitze Ecke (d. h. eine solche, deren Innenwinkel größer als π ist) überschreitet; siehe Abbildung 4.26. Das kritische Tangentenstück ist gestrichelt eingezeichnet; wir nennen es ein *Sichtsegment*.

Sichtsegment

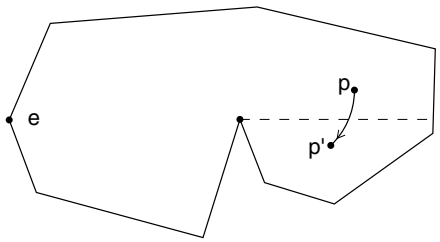


Abb. 4.26 Auf dem Weg von p zu p' wird der Eckpunkt e unsichtbar.

Man nennt zwei Punkte p, p' in P *äquivalent*, wenn von p aus genau dieselben Ecken des Polygons sichtbar sind wie von p'. Abbildung 4.27 zeigt ein Beispiel für die Aufteilung eines Polygons in Regionen äquivalenter Punkte. Wir nennen diese Äquivalenzklassen die *Sichtregionen* von P.

Sichtregion

Übungsaufgabe 4.13 Man zeige, daß zwei äquivalente Punkte einander sehen können.

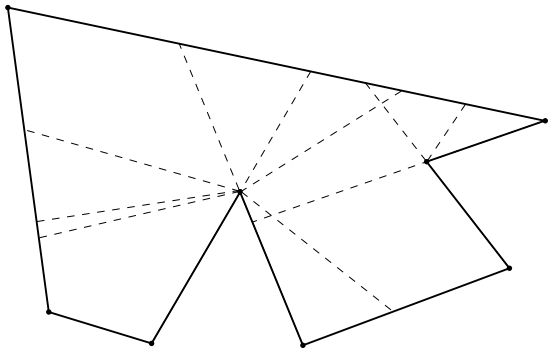


Abb. 4.27 Die Zerlegung eines Polygons in Regionen von Punkten mit ähnlicher Sicht.

Eine interessante Frage lautet, wie viele wesentlich verschiedene Sichten, d. h. wie viele Sichtregionen es in einem Polygon mit n Ecken geben kann.

Anzahl der Sichtregionen

Theorem 4.20 *Ein einfaches Polygon mit n Ecken kann in $\Theta(n^3)$ Sichtregionen zerfallen.*

Beweis. Jeder der n Eckpunkte kann mit jeder der $O(n)$ vielen spitzen Ecken höchstens ein Sichtsegment bilden. Also gibt es insgesamt nur $O(n^2)$ viele Sichtsegmente.

Halten wir nun einen Eckpunkt e fest und betrachten alle $O(n)$ zu e gehörigen Sichtsegmente, bei denen die spitze Ecke von e aus gesehen rechts sitzt, wie in Abbildung 4.28 gezeigt. Kein Punkt im Inneren eines dieser Sichtsegmente kann einen Punkt im Inneren eines anderen sehen. Folglich kann jedes der $O(n^2)$ insgesamt vorhandenen anderen Sichtsegmente *höchstens eines* der hier eingezeichneten kreuzen! Dem Eckpunkt e sind auf diese Weise $O(n^2)$ Kreuzungen von Sichtsegmenten zugeordnet; insgesamt ergibt das $O(n^3)$ viele Kreuzungen.

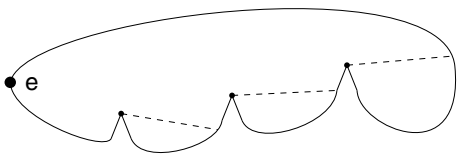


Abb. 4.28 Jede Ecke e erzeugt nur $O(n)$ „rechte“ Sichtsegmente. Sie sind zueinander unsichtbar.

Die Zerlegung von P in Sichtregionen kann als kreuzungsfreier, schlichter geometrischer Graph in der Ebene aufgefaßt werden,

dessen Knoten entweder Kreuzungen von zwei Sichtsegmenten sind und deshalb den Grad ≥ 4 haben oder Endpunkte von Sichtsegmenten auf dem Polygonrand vom Grad ≥ 3 sind.[18] Nach Übungsaufgabe 1.6 (ii) auf Seite 18 hat solch ein Graph höchstens doppelt so viele Flächen wie Knoten. Demnach ist die Anzahl der Sichtregionen durch $O(n^3)$ begrenzt.

Daß die Anzahl der Sichtregionen tatsächlich kubisch sein kann, zeigt Abbildung 4.29. □

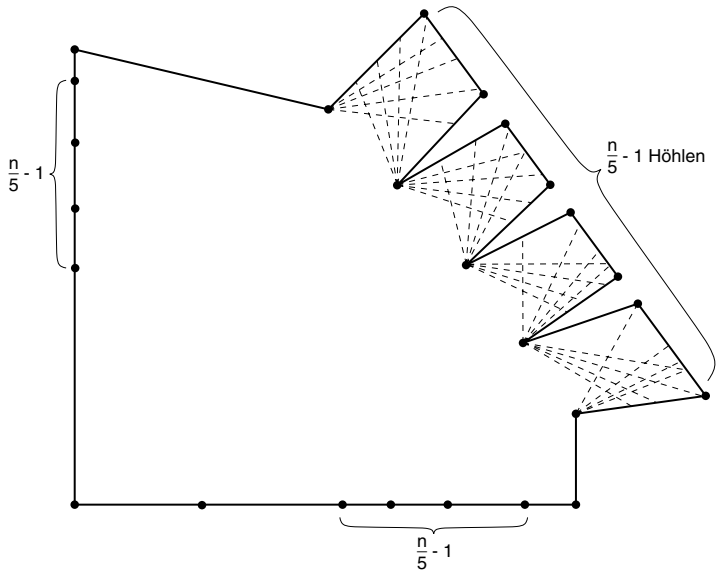


Abb. 4.29 Ein Polygon mit n Ecken kann $\Omega(n^3)$ Sichtbarkeitsregionen besitzen.

4.3.3 Das Kunstgalerie-Problem

art gallery problem

Eine ganz andere Frage ist unter dem Namen *Kunstgalerie-Problem* (engl. *art gallery problem*) bekanntgeworden. Gegeben ist ein Raum mit einem einfachen Polygon P als Grundriß. Es sollen sich Wächter so im Raum aufstellen, daß jeder Punkt des Raums bewacht ist. Die Wächter sollen nicht umhergehen, können aber von ihren Standorten in alle Richtungen schauen. Wie viele Wächter werden benötigt, und wo sollen sie sich aufstellen?

[18] Eine Folge konvexer Kanten von P fassen wir bei dieser Betrachtung zu einer Kante zusammen.

Formal ist die kleinste Zahl k gesucht, für die Punkte $p_1, \ldots, p_k$ in P existieren mit

$$P = \bigcup_{i=1}^{k} vis(p_i).$$

Interessanterweise ist die Bestimmung der minimalen Wächterzahl NP-hart; das Problem läßt sich auf 3SAT reduzieren. Ein Beweis findet sich in der Monographie über Kunstgalerie-Probleme von O'Rourke [113]. Man muß sich also mit Lösungen begnügen, die mehr Wächter verwenden als vielleicht nötig wäre.

Übungsaufgabe 4.14 Reicht es aus, an jeder spitzen Ecke einen Wächter zu postieren?

Unabhängig von der Anzahl der spitzen Ecken, die zwischen 0 und $n-3$ liegen kann, gilt die folgende Aussage, die auf Chvátal [32] zurückgeht:

Theorem 4.21 *Ein einfaches Polygon mit n Ecken kann stets von $\lfloor \frac{n}{3} \rfloor$ Wächtern überwacht werden. Es gibt beliebig große Beispiele, in denen diese Anzahl auch benötigt wird.*

Beweis. Daß man mit weniger als $\lfloor \frac{n}{3} \rfloor$ vielen Wächtern nicht auskommt, zeigt Abbildung 4.30.

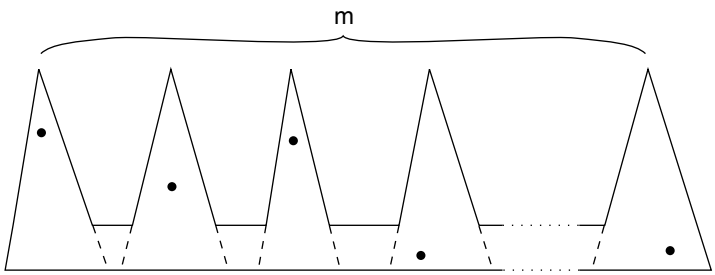


Abb. 4.30 Ein Polygon mit $3m$ Ecken, zu dessen Bewachung m Wächter erforderlich sind.

Zum Nachweis der oberen Schranke betrachten wir eine beliebige *Triangulation* des Polygons; siehe Abbildung 4.31. Wir wollen uns zunächst klarmachen, daß der resultierende kreuzungsfreie geometrische Graph *3-färbbar* ist, d. h. daß man jedem der n Eckpunkte eine von drei Farben $\{1, 2, 3\}$ so zuordnen kann, daß keine zwei Ecken, die über eine Polygonkante oder eine Diagonale der Triangulation miteinander verbunden sind, dieselbe Farbe bekommen.[19]

Triangulation

3-färbbar

[19] Der berühmte *Vier-Farben-Satz* besagt, daß man bei jedem kreuzungsfreien geometrischen Graphen mit *vier* Farben auskommt, aber Triangulationsgraphen sind etwas Besonderes.

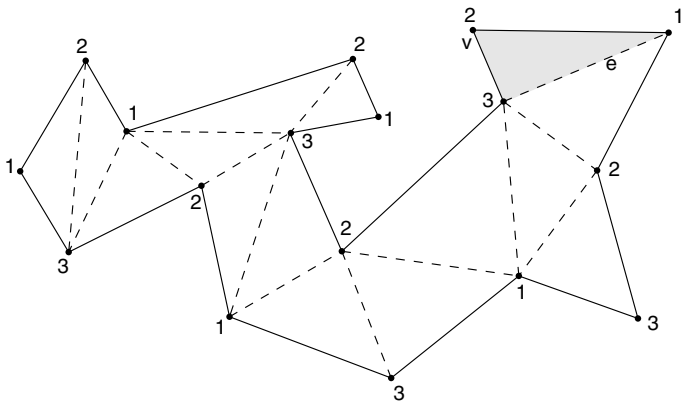


Abb. 4.31 Eine 3-Färbung der Triangulation eines Polygons.

Der Beweis ist induktiv: Ist P selbst ein Dreieck, stimmt die Behauptung. Andernfalls läßt sich nach Theorem 4.16 ein Ohr in der Triangulation finden wie etwa das in Abbildung 4.31 grau schraffierte Dreieck, das mit den übrigen nur eine einzelne Kante e gemeinsam hat. Per Induktion läßt sich der Rest 3-färben, und wenn wir das entfernte Dreieck wieder einfügen, färben wir die neue Ecke v mit der Farbe, die noch nicht für die Ecken von Kante e verbraucht sind.

Die drei Ecken eines jeden Dreiecks haben nach erfolgter 3-Färbung natürlich unterschiedliche Farben; es kommen deshalb bei jedem Dreieck alle drei Farben vor. Wenn wir also eine Farbe i auswählen und an jeder mit i gefärbten Ecke von P einen Wächter postieren, sind alle Dreiecke – und damit das gesamte Polygon – bewacht.

Unter den drei Farben muß es mindestens eine geben, mit der $\leq \frac{n}{3}$ viele Knoten gefärbt sind; weil diese Knotenzahl ganzzahlig ist, muß sie dann sogar $\leq \lfloor \frac{n}{3} \rfloor$ sein. Dort postieren wir die Wachen. □

Der Beweis hat auch gezeigt: Wenn man wirklich $\lfloor \frac{n}{3} \rfloor$ Wächter einsetzen will, kann man sie sogar an den Wänden stationieren.

Übungsaufgabe 4.15 Angenommen, man hat eine Gruppe von Wächtern so am Rand des Polygons stationiert, daß sie den gesamten Rand bewachen können. Können die Wächter dann auch schon das Innere von P bewachen?

4.4 Der Kern eines einfachen Polygons

Am Ende von Abschnitt 4.3 hatten wir diskutiert, wie viele Wächter man aufstellen muß, um einen Raum mit polygonalem Grundriß zu überwachen. Wir interessieren uns jetzt für solche Polygone, bei denen bereits *ein* Wächter ausreicht. Ein einfaches Polygon P heißt *sternförmig*, wenn es einen Punkt p in P gibt mit

sternförmig

$$vis(p) = P.$$

Die Gesamtheit aller dieser Punkte nennen wir den *Kern* von P, d. h.

Kern

$$ker(P) = \{p \in P; vis(p) = P\}.$$

Genau dann ist P sternförmig, wenn $ker(P) \neq \emptyset$ gilt. Beispiel (ii) in Abbildung 4.32 zeigt, daß es sternförmige Polygone gibt, deren Form nicht sofort an einen Stern erinnert.

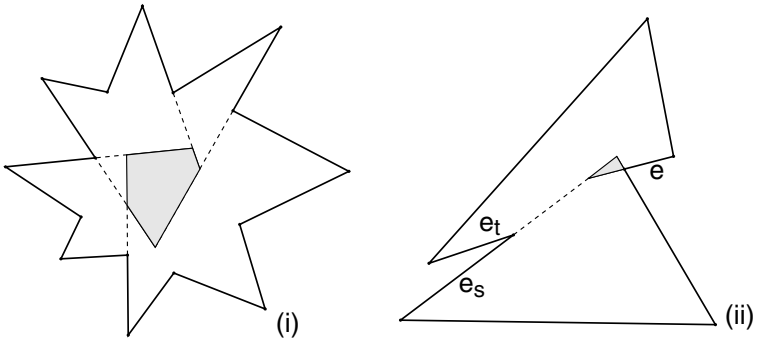


Abb. 4.32 Zwei sternförmige Polygone und ihre Kerne.

Natürlich ist jedes konvexe Polygon auch sternförmig; aus Übungsaufgabe 1.8 auf Seite 22 folgt

$$P \text{ ist konvex } \iff ker(P) = P.$$

Trivialerweise ist jedes Sichtbarkeitspolygon $vis(p)$ sternförmig und enthält p in seinem Kern.

Übungsaufgabe 4.16 Man zeige, daß für jeden Punkt p in einem einfachen Polygon P die folgende Aussage gilt:

$$ker(P) \subseteq ker(vis(p)).$$

In diesem Abschnitt geht es uns darum, zu gegebenem Polygon P schnell seinen Kern zu berechnen oder festzustellen, daß $ker(P)$ leer ist.

Ziel: Berechnung von $ker(P)$

4.4.1 Die Struktur des Problems

Wir beginnen mit einer Charakterisierung des Kerns als Durchschnitt von Halbebenen. Der Rand von P sei wieder gegen den Uhrzeigersinn orientiert. Für eine Kante e von P bezeichne $H^+(e)$ die abgeschlossene Halbebene zur Linken von e und $H^-(e)$ die offene Halbebene zur Rechten. In der Nähe der Kante e ist also das Innere von P in $H^+(e)$ enthalten; siehe Abbildung 4.33.

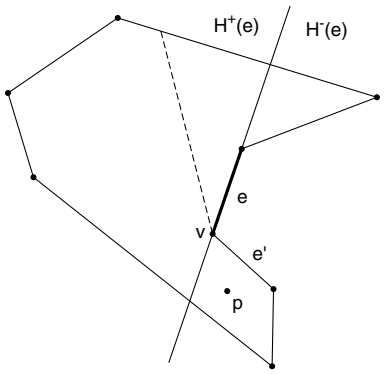


Abb. 4.33 Der Punkt p kann die Kante e nicht sehen, weil er in $H^-(e)$ liegt.

Lemma 4.22

$$ker(P) = \bigcap_{e \text{ Kante von } P} H^+(e)$$

Beweis. Daß der Kern in jeder Halbebene $H^+(e)$ enthalten ist, wird durch Abbildung 4.33 verdeutlicht: Wenn ein Punkt $p \in H^-(e)$ überhaupt zu P gehört, kann er die Kante e gewiß nicht sehen. Umgekehrt sei p in jeder Halbebene $H^+(e)$ enthalten. Dann liegt p zumindest im Polygon P, denn andernfalls könnte p mindestens eine Kante e „von außen“ sehen und läge deshalb in $H^-(e)$. Läge p nicht im Kern von P, wäre $vis(p)$ echte Teilmenge von P, und es gäbe eine spitze Ecke v von P, von deren beiden Kanten e, e' der Punkt p nur e' sehen kann, e aber nicht. Dann wäre $p \in H^-(e)$, im Widerspruch zur Annahme. □

Kern in Zeit $O(n \log n)$

Da nach Lemma 4.22 der Kern von P Durchschnitt von n bekannten Halbebenen ist, läßt er sich nach Übungsaufgabe 4.3 auf Seite 171 in Zeit $O(n \log n)$ berechnen. Das genügt uns aber nicht! Wir wollen die Tatsache ausnutzen, daß die n Halbebenen nicht beliebig sind, sondern von den Kanten eines einfachen Polygons herrühren, um zu einer noch effizienteren Lösung zu gelangen.

Bei Betrachtung von Abbildung 4.32 fällt auf, daß manche Kanten e gar nicht zum Kern von P beitragen, weil ihre Halbebenen $H^+(e)$ ihn im Inneren enthalten. Solche Kanten heißen *unwesentlich*. Im Gegensatz dazu nennen wir eine Kante *wesentlich*, wenn ihre Verlängerung zum Rand des Kerns beiträgt.

wesentliche und unwesentliche Kanten

Übungsaufgabe 4.17 Sind alle Kanten unwesentlich, die nicht an einer spitzen Ecke von P sitzen?

Wir werden im folgenden eine Teilmenge der Menge aller Kanten von P bestimmen, die eine schöne Struktur besitzt und alle für $ker(P)$ wesentlichen Kanten enthält. Zunächst betrachten wir die in Abschnitt 4.2 eingeführten *Drehwinkel* zwischen den Kanten auf dem Rand von P. Sei

Drehwinkel

$$\alpha_{\max} := \max_{i \neq j} \alpha_{i,j} = \alpha_{s,t}$$

der maximale Drehwinkel zwischen zwei verschiedenen Kanten; er werde auf dem Weg von e_s nach e_t angenommen. Abbildung 4.34 zeigt ein Beispiel. Wir bezeichnen mit a den Startpunkt von e_s und mit b den Endpunkt von e_t.

Im folgenden bezeichne allgemein $\alpha(e, f)$ den Drehwinkel, den der Rand von P auf dem Weg von Kante e zur Kante f beschreibt; es ist also $\alpha(e_i, e_k) = \alpha_{i,k}$.

Zunächst machen wir uns klar, daß der Kern von P sicher leer ist, wenn $\alpha_{\max}$ zu groß ist.

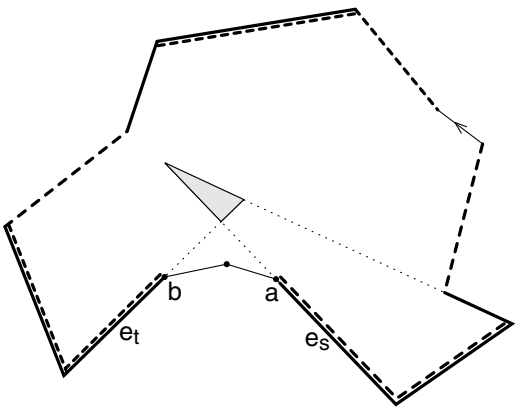


Abb. 4.34 Der größtmögliche Drehwinkel beträgt hier $\frac{5}{2}\pi$; er wird auf dem Weg von e_s nach e_t angenommen.

Lemma 4.23 *Ist der maximale Drehwinkel $\alpha_{\max}$ eines einfachen Polygons P größer gleich 3π, so hat P einen leeren Kern.*

leerer Kern bei Drehwinkel $\geq 3\pi$

Beweis. Wenn sich der Polygonrand auf dem Weg von e_s nach e_t um mehr als 3π linksherum dreht, so muß er sich beim Weiterlaufen von e_t nach e_s um mindestens π rechtsherum drehen, denn nach Lemma 4.17 beträgt bei einer vollständigen Umkreisung der Drehwinkel genau 2π. Es gibt also auf dem Rand von P eine Kantenfolge $e_i, \ldots, e_j$ mit

$$\alpha(e_i, e_j) \leq -\pi,$$

und wir betrachten eine solche mit minimaler Länge. Dann ist jede Kante e in dieser Folge gegenüber e_i rechtsherum verdreht. Wäre nämlich e gegenüber e_i linksherum verdreht, hätten wir $\alpha(e_i, e) \geq 0$. Dann hätte die Teilfolge ab e erst recht einen Drehwinkel von

$$\alpha(e, e_j) = \alpha(e_i, e_j) - \alpha(e_i, e) \leq \alpha(e_i, e_j) \leq -\pi,$$

im Widerspruch zur Minimalität von $e_i, \ldots, e_j$.

Aus der Minimalität folgt auch, daß e_j die erste Kante hinter e_i mit Drehwinkel $< -\pi$ ist. Insgesamt sind also alle Kanten zwischen e_i und e_j in Bezug auf e_i rechtsherum gedreht, aber höchstens um eine halbe Drehung. Wie Abbildung 4.35 verdeutlicht, kann deshalb keine dieser Kanten die innere Halbebene $H^+(e_i)$ betreten. Der gemeinsame Eckpunkt von e_{j-1} und e_j besitzt eine zu e_i parallele Tangente. Sie trennt die innere Halbebene $H^+(e_i)$ vom Durchschnitt

$$H^+(e_{j-1}) \cap H^+(e_j).$$

Da schon diese drei inneren Halbebenen einen leeren Durchschnitt haben, ist erst recht der Kern von P leer. □

ab jetzt $\alpha_{\max} < 3\pi$

Ab jetzt nehmen wir an, daß der maximale Drehwinkel von P kleiner als 3π ist. Daraus folgt:

Korollar 4.24 *Sei der maximale Drehwinkel von P kleiner als 3π. Dann gilt für je zwei Kanten e_i, e_k die Abschätzung $-\pi < \alpha_{i,k}$.*

Beweis. Wegen Lemma 4.17 ist

$$\alpha_{i,k} = 2\pi - \alpha_{k,i} \geq 2\pi - \alpha_{\max} > -\pi.$$ □

Abbildung 4.36 zeigt schematisch, wie solch ein Polygon aussieht.

Nun definieren wir eine wohlstrukturierte Menge von Kanten, die alle für den Kern wesentlichen Kanten enthält. Sie besteht aus zwei *Teilfolgen*

$$\begin{aligned} F &= (f_0 = e_s, f_1, f_2, \ldots, f_k = e_t) \\ B &= (b_0 = e_t, b_1, b_2, \ldots, b_l = e_s) \end{aligned}$$

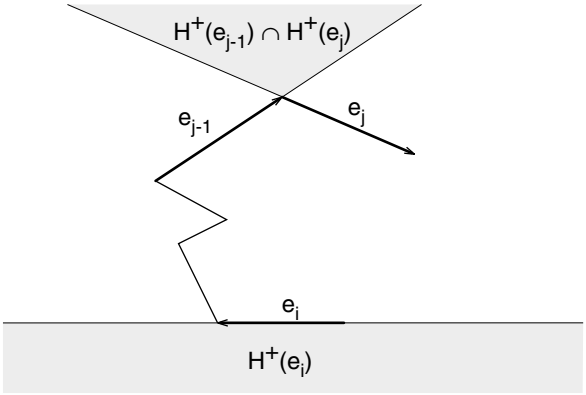


Abb. 4.35 Solange die Kanten gegenüber e_i nur rechtsherum verdreht sind, aber höchstens um eine halbe Drehung, können sie die Halbebene $H^+(e_i)$ nicht betreten.

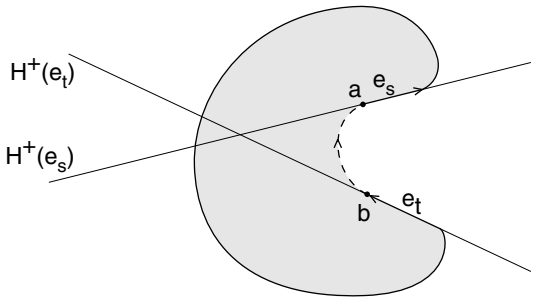


Abb. 4.36 Schematischer Verlauf eines Polygons mit maximalem Drehwinkel $< 3\pi$ zwischen e_s und e_t.

der „maximalen“ Kantenfolge $e_s, e_{s+1}, \ldots, e_t$ mit $\alpha_{\max} = \alpha_{s,t}$.[20]

Die Kantenfolge F wird rekursiv definiert; wir starten mit e_s, durchlaufen den Polygonrand gegen den Uhrzeigersinn und nehmen jeweils die nächste Kante in die Folge auf, die sich weiter linksherum dreht als ihre Vorgängerinnen: Kantenfolge F

$$f_0 := e_s$$

$$f_{i+1} := \begin{cases} \text{erste Kante } e \text{ hinter } f_i \text{ mit } \alpha(f_i, e) > 0, \\ \quad \text{falls } f_i \text{ noch vor } e_t \text{ liegt}, \\ \text{undefiniert sonst.} \end{cases}$$

In Abbildung 4.34 sind die zu F gehörenden Kanten mit durchgezogenen dicken Linien dargestellt. Wegen $\alpha(e_s, e_t) = \alpha_{\max}$ ist

[20] Sollte es mehrere Kantenfolgen geben, deren Drehwinkel $\alpha_{\max}$ beträgt, wählen wir die kürzeste von ihnen aus.

die Kante e_t auf jeden Fall dabei und stellt das letzte Element der Folge F dar.

Leider ist nicht jede wesentliche Kante bereits in F enthalten, wie Beispiel (ii) in Abbildung 4.32 auf Seite 195 zeigt: Die Verlängerung von e berandet den Kern, aber e kommt in F nicht vor.

Aus diesem Grund wird ganz symmetrisch eine zweite Kantenfolge B definiert; man startet mit e_t, durchläuft den Rand von P im Uhrzeigersinn und nimmt jede Kante in die Folge B auf, die sich weiter rechtsherum dreht als ihre Vorgängerinnen. In bezug auf die Standardorientierung von ∂P gegen den Uhrzeigersinn bedeutet das:

Kantenfolge B

$$b_0 := e_t$$

$$b_{i+1} := \begin{cases} \text{letzte Kante } e \text{ vor } b_i \text{ mit } \alpha(e, b_i) > 0, \\ \quad \text{falls } b_i \text{ noch hinter } e_s \text{ liegt,} \\ \text{undefiniert sonst.} \end{cases}$$

Die Kanten aus B sind in Abbildung 4.34 gestrichelt dargestellt; das letzte Folgenglied ist e_s. Die Definitionen von F und B hängen eng zusammen: Wenn P^* das Spiegelbild von P bedeutet, ist $B(P)$ das Spiegelbild von $F(P^*)$.

Abbildung 4.34 zeigt, daß manche Kanten von P sowohl zu F als auch zu B gehören, andere dagegen zu keiner von beiden Folgen. Die nächste Aussage ist von entscheidender Bedeutung:

$F \cup B$ enthält alle wesentlichen Kanten

Theorem 4.25 *Sei P ein einfaches Polygon mit maximalem Drehwinkel $< 3\pi$. Dann gehört jede für den Kern wesentliche Kante von P zur Folge F oder zur Folge B.*

e_0 zwischen e_s und e_t

Beweis. Sei e_0 eine Kante, die weder zu F noch zu B gehört. Wir unterscheiden zwei Fälle. Im ersten Fall liegt e_0 auf dem Randstück von e_s nach e_t. Sei f_i die letzte Kante aus F vor e_0. Weil erst f_{i+1} wieder gegenüber f_i linksherum gedreht ist, gilt $\alpha(f_i, e) \leq 0$ für jede Kante e ab f_i bis e_0 einschließlich. Wegen Korollar 4.24 ist außerdem $-\pi < \alpha(f_i, e)$; jedes e bis e_0 einschließlich ist also gegenüber f_i rechtsherum gedreht, aber höchstens um eine halbe Umdrehung. Wie schon beim Beweis von Lemma 4.23 bemerkt, muß dieses Randstück ganz in der Halbebene $H^-(f_i)$ enthalten sein, und jedem e bis e_0 einschließlich ist nur der in Abbildung 4.37 (i) eingezeichnete Winkelbereich erlaubt.

Nun betrachten wir die erste Kante b_j aus B hinter e_0. Ganz analog gilt

$$-\pi < \alpha(e_0, b_j) \leq 0$$

und unter nochmaliger Anwendung von Korollar 4.24 auch

$$-\pi < \alpha(f_i, b_j) = \alpha(f_i, e_0) + \alpha(e_0, b_j) \leq 0.$$

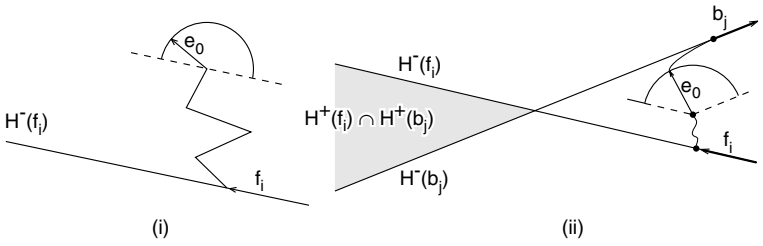


Abb. 4.37 Wenn f_i die letzte Kante aus F vor e_0 und b_j die erste Kante aus B hinter e_0 sind, muß e_0 in $H^-(f_i) \cap H^-(b_j)$ und dem in (ii) dargestellten Winkelbereich enthalten sein.

Deshalb liegen f_i und b_j so zueinander, wie in Abbildung 4.37 (ii) eingezeichnet. Außerdem gelten dieselben Überlegungen für b_j wie vorher für f_i. Daher ist die Kante e_0 *im Durchschnitt* von $H^-(b_j)$ mit $H^-(f_i)$ enthalten, und ihre Steigung liegt in dem in Figur (ii) dargestellten Winkelbereich.

Dann kann aber die Gerade durch e_0 den Keil $H^+(f_i) \cap H^+(b_j)$ nicht schneiden! Weil der Kern von P in diesem Keil enthalten ist, kann e_0 nur unwesentlich sein.

Im zweiten Fall liegt e_0 auf dem Randstück von e_t nach e_s, das in Abbildung 4.36 gestrichelt eingezeichnet ist. Hier gelten wegen der Maximalität von $\alpha_{\max} = \alpha_{s,t}$ und Korollar 4.24 für jede solche Kante e die Abschätzungen

e_0 zwischen e_t und e_s

$$-\pi < \alpha(e_t, e) \leq 0$$
$$-\pi < \alpha(e, e_s) \leq 0,$$

und wir können die Argumentation von oben auf e_t, e_s statt auf f_i, b_j anwenden; wieder ist e_0 unwesentlich. □

4.4.2 Ein optimaler Algorithmus

Was ist nun mit Theorem 4.25 gewonnen? Wir stehen doch immer noch vor dem Problem, den Durchschnitt von $O(n)$ Halbebenen zu berechnen!

Im Unterschied zu früher sind die Geraden durch die Kanten aus F und B, die die Halbebenen definieren, jetzt aber *nach Winkeln geordnet*. Wir können deshalb Korollar 4.12 auf Seite 174 anwenden und die Durchschnitte

geordnet geht es schneller!

$$\bigcap_{e \in F} H^+(e) \text{ und } \bigcap_{e \in B} H^+(e)$$

jeweils in Zeit $O(n)$ berechnen (durch Reduktion auf die Berechnung der konvexen Hüllen sortierter Punktmengen). Als Ergebnis entstehen dabei zwei konvexe Mengen[21] mit $O(n)$ vielen Kanten. Deren Durchschnitt läßt sich nach Theorem 2.19 auf Seite 93 in Zeit $O(n)$ konstruieren, zum Beispiel per *sweep*.

Zunächst muß aber der maximale Drehwinkel $\alpha_{\max}$ bestimmt werden. Dieses Problem ist ein alter Bekannter aus Kapitel 2: die Bestimmung einer maximalen Teilsumme! Nur stehen die Zahlen jetzt nicht in einem Array, sondern in einer zyklisch verketteten Liste. Der Algorithmus vor Theorem 2.2 auf Seite 56 läßt sich aber leicht an diese Situation anpassen und erlaubt uns, sowohl $\alpha_{\max}$ als auch ein Randstück $e_s, e_{s+1}, \ldots, e_t$, bei dem $\alpha_{\max}$ angenommen wird, in Zeit $O(n)$ zu bestimmen. Die Folgen F und B lassen sich dann ebenfalls in linearer Zeit gewinnen, durch je einen Durchlauf von e_s nach e_t und zurück.

fertig!

Damit haben wir folgenden Satz:

Theorem 4.26 *Der Kern eines einfachen Polygons mit n Ecken läßt sich in Zeit $O(n)$ und linearem Speicherplatz berechnen.*

Der hier vorgestellte Ansatz zur Berechnung des Kerns geht im wesentlichen auf die Arbeit von Cole und Goodrich [35] zurück. Der hier dargestellte Beweis von Lemma 4.23 wurde von Schulz [127] angegeben. Implementiert wurde unser Algorithmus im Applet

`http://www.geometrylab.de/VisPolygon/`

Ein direkterer, aber technisch etwas komplizierterer Algorithmus zur Berechnung des Kerns stammt von Lee und Preparata [91]; er findet sich auch im Buch von Preparata und Shamos [120].

Sichtbarkeitsrelation

Viele der in diesem Kapitel behandelten Probleme beruhten auf der *Sichtbarkeitsrelation* zwischen Punkten. Man kann diesen Begriff auch auf ganz andere Weise definieren; siehe Munro et al. [108]. Zu den Sichtbarkeitsproblemen gehören auch die Fragen, welcher Punkt einer Szene von einem Strahl als erster getroffen wird und welche Teile einer komplexen dreidimensionalen Umgebung von einem Beobachterstandpunkt aus sichtbar sind. Diese Probleme sind als *ray shooting* sowie *hidden line elimination* bzw. *hidden surface removal* bekannt; siehe zum Beispiel de Berg [37].

ray shooting

[21] Eine oder beide können leer sein.

Lösungen der Übungsaufgaben

Übungsaufgabe 4.1 Die Inklusion $ch(S) \subseteq ch(P)$ ist trivial, denn wegen $S \subset P$ ist $ch(P)$ eine konvexe Menge, die die Punkte aus S enthält.

Zum Nachweis von $ch(P) \subseteq ch(S)$: Die konvexe Hülle der Ecken von P ist in $ch(S)$ enthalten, enthält den ganzen Rand von P und damit auch das Innere.

Übungsaufgabe 4.2 Eine Kante e in der Kontur von L nach O könnte sich höchstens mit einer Kante f zwischen R und U kreuzen. Nach Definition der Kontur können aber die in Abbildung 4.38 eingezeichneten Quadranten keinen Punkt aus S enthalten. Also kann eine solche Kreuzung nicht existieren.

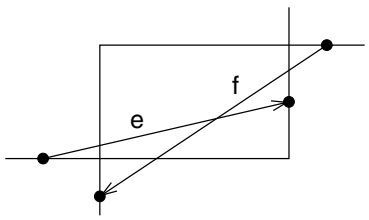


Abb. 4.38 Weil keiner der Quadranten einen Punkt aus S enthalten kann, ist eine solche Kreuzung nicht möglich.

Übungsaufgabe 4.3 Wir zerlegen die Menge der Halbebenen in zwei gleich große Teilmengen und berechnen rekursiv deren Durchschnitte. Als Ergebnis erhalten wir zwei konvexe Mengen, die durch polygonale Ketten mit höchstens n Kanten berandet sind. Deren Durchschnitt läßt sich mit einem *sweep* in Zeit $O(n)$ berechnen, wie wir uns beim Beweis von Theorem 2.19 für konvexe Polygone als Spezialfall überlegt hatten. Insgesamt ergibt sich eine Laufzeit in $O(n \log n)$.

Übungsaufgabe 4.4 Nein! Ihre Geraden sind nicht senkrecht und schneiden deshalb die Y-Achse. Das Stück der Y-Achse unterhalb des tiefsten Schnittpunkts gehört zum Durchschnitt der unteren Halbebenen. Man kann sich auch auf Theorem 4.10 berufen; schließlich muß jede konvexe Hülle mindestens eine untere Kante besitzen.

Übungsaufgabe 4.5 Der Algorithmus ist dem Vorgehen im Zweidimensionalen sehr ähnlich. Zu Beginn wählen wir einen Punkt z im Innern von $ch(S_4)$. Wenn der Punkt p_i eingefügt werden soll, verfügen wir über ein Array A, daß uns für jeden Index

$j \geq i$ einen Zeiger auf dasjenige Dreieck[22] auf der Oberfläche von $ch(S_{i-1})$ liefert, welches vom Segment zp_j geschnitten wird, bzw. **nil** im Fall $p_j \in ch(S_i)$. Den Tangentenpunkten bei der Konstruktion der zweidimensionalen konvexen Hülle entspricht hier der *Horizont* (auch *Silhouette* genannt), also die geschlossene polygonale Kette der – von p_i aus gesehen – äußersten Kanten von $ch(S_{i-1})$. Vom Dreieck $A[i]$ beginnend, finden wir die h Kanten des Horizonts durch Breitensuche. Die Anzahl der dabei besuchten Dreiecke von $ch(S_{i-1})$ ist proportional zur Anzahl der Ecken von $ch(S_{i-1})$, die nicht mehr auf der Oberfläche von $ch(S_i)$ liegen werden, plus h. Um nun $ch(S_i)$ zu konstruieren, wird jede Ecke des Horizonts mit p_i verbunden. Also entspricht h dem Grad von p_i in dem kreuzungsfreien Graph, der durch die Dreiecke von $\partial ch(S_i)$ gebildet wird. Aus Korollar 1.2 – angewendet auf den zu unserer Triangulation dualen Graphen – folgt, daß h im Mittel kleiner als 6 ist, denn p_i ist ja aus den Punkten von S_i zufällig ausgewählt.

Horizont

Silhouette

Jetzt muß noch das Array A aktualisiert werden. Für jeden Punkt p_j, für den $A[j]$ auf eines der verschwundenen Dreiecke zeigte, testen wir, welches der h neuen Dreiecke von zp_j geschnitten wird. Die Wahrscheinlichkeit dafür, daß ein p_j hiervon betroffen ist, entspricht der Wahrscheinlichkeit, daß p_i einer der drei Eckpunkte des Dreiecks $A[j]$ in $\partial ch(S_i)$ ist, also $\frac{3}{i}$. Der Rest des Beweises folgt dem zweidimensionalen Fall.

Übungsaufgabe 4.6 Nein; ein Gegenbeispiel ist in Abbildung 4.39 gezeigt.

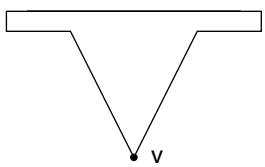


Abb. 4.39 Dieses Polygon hat keine Diagonale mit Endpunkt v.

Übungsaufgabe 4.7 Beweis durch Induktion über die Anzahl n der Ecken von P. Für Dreiecke stimmen beide Behauptungen. Sei also $n \geq 4$. Dann wählen wir aus der in Rede stehenden Triangulation eine Diagonale aus und betrachten die beiden Teilpolygone. Weil keines von ihnen mehr als $n-1$ Ecken hat, läßt sich die Induktionsvoraussetzung anwenden, und direktes Nachrechnen liefert die Behauptung.

[22] Wir gehen davon aus, daß keine 4 Punkte auf einer Ebene liegen, so daß die Oberfläche der konvexen Hülle aus lauter Dreiecken besteht.

Übungsaufgabe 4.8 Nein; siehe Abbildung 4.40.

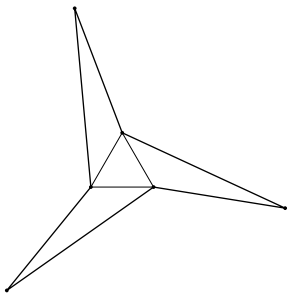


Abb. 4.40 Die einzig mögliche Triangulation dieses Polygons enthält ein Dreieck mit drei Nachbarn.

Übungsaufgabe 4.9 Ja. Für die in Abbildung 4.41 gezeigte Kurve ergibt sich beim Durchlauf gegen den Uhrzeigersinn eine Gesamtdrehung von $(1 + a - b)2\pi$, und a, b können beliebig ≥ 0 gewählt werden.

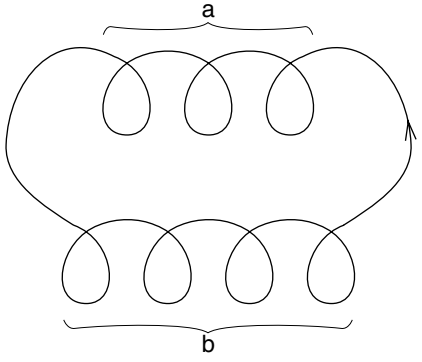


Abb. 4.41 Für $a = b \geq 1$ erhält man eine Kurve, die nicht einfach ist und trotzdem eine Gesamtdrehung von 2π aufweist.

Übungsaufgabe 4.10 Für Dreiecke sind beide Aussagen klar. Den allgemeinen Fall beweist man induktiv durch Abschneiden eines Ohrs einer Triangulation von P.

Übungsaufgabe 4.11 Man wähle einen beliebigen Strahl C, der von p nach ∞ läuft, und teste nacheinander die n Kanten von P auf Schnitt mit C. Der am dichtesten bei p gelegene Schnittpunkt ist von p aus sichtbar.

Übungsaufgabe 4.12 Wir ziehen in Abbildung 4.25 den Punkt W so weit nach unten, daß er unterhalb des Strahls C liegt. Die

hierdurch entstehende Blockade der Sicht von p auf T wird ignoriert, weil wir den Rand von Q bis T ja im schnellen Vorlauf durcheilen. Das Stück kurz hinter T wird also zunächst in den Stapel S eingefügt.

Übungsaufgabe 4.13 Von jeder spitzen Ecke von P gehen zwei Sichtsegmente ins Innere von P, die die beiden anliegenden Kanten verlängern. Also wird jede Sichtregion durch Teile von Sichtsegmenten und durch konvexe Kantenfolgen von P berandet. Folglich sind die Regionen konvex.

Übungsaufgabe 4.14 Wenn das Polygon überhaupt spitze Ecken hat, ist von jedem Punkt p aus eine spitze Ecke sichtbar. (Sähe p nur konvexe Ecken, also solche mit Innenwinkel $\leq \pi$, wäre der gesamte Rand von p aus sichtbar, weil sich Teile des Randes nur hinter spitzen Ecken verstecken können; dann wäre aber das Polygon konvex!) Also genügt es, an jeder spitzen Ecke einen Wächter aufzustellen. Ist das Polygon jedoch konvex, wird *ein* Wächter benötigt, der sich an beliebiger Position aufhalten darf.

Übungsaufgabe 4.15 Nein! Abbildung 4.42 zeigt ein Gegenbeispiel.

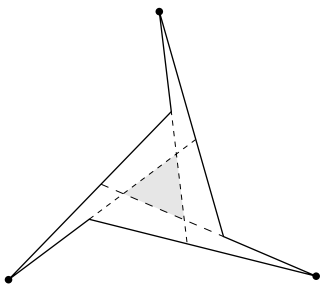


Abb. 4.42 Die drei Wächter in den äußeren Ecken bewachen zwar den gesamten Rand des Polygons, aber nicht den grauen Bereich im Inneren.

Übungsaufgabe 4.16 Es liegt nahe, so zu argumentieren: „Jeder Punkt q, der ganz P sieht, kann erst recht $vis(p)$ sehen, denn $vis(p)$ ist ja in P enthalten!“ Dieses Argument übersieht, daß das Polygon $vis(p)$ Kanten besitzt, die in P nicht vorkommen und im Prinzip die Sicht von q einschränken könnten.

Sei also $q \in ker(P)$. Dann sieht q insbesondere p, ist also selbst in $vis(p)$ enthalten. Sei x ein beliebiger Punkt aus $vis(p)$; weil q im Kern von P liegt, wird das Liniensegment qx jedenfalls nicht von einer Kante von P geschnitten. Nun hat $vis(p)$ außerdem noch

künstliche Kanten, die durch spitze Ecken entstehen. Würde eine solche Kante das Segment qx schneiden, wäre entweder q oder x von p aus „hinter der Ecke“ und darum nicht sichtbar. Beides ist unmöglich: q sieht auch p, und x liegt nach Voraussetzung in $vis(p)$!

Weil das Segment qx von keiner Kante von $vis(p)$ geschnitten wird, ist es in $vis(p)$ enthalten. Da also q jeden Punkt x in $vis(p)$ sehen kann, folgt $q \in ker(vis(p))$.

Übungsaufgabe 4.17 Nein, wie man am Beispiel eines konvexen Polygons sehen kann. Richtig ist aber die Gleichung

$$ker(P) = P \cap \bigcap_{\substack{e \text{ Kante einer spitzen} \\ \text{Ecke von } P}} H^+(e),$$

die sich aus dem Beweis von Lemma 4.22 ergibt.

5

Voronoi-Diagramme

5.1 Einführung

Der Gegenstand dieses und des folgenden Kapitels unterscheidet sich von den übrigen Teilen dieses Buchs und generell von einem Großteil der Informatik dadurch, daß seine Geschichte bis ins 17. Jahrhundert zurückreicht.

In seinem Buch [43] über die Prinzipien der Philosophie äußert Descartes[1] 1644 die Vermutung, das Sonnensystem bestehe aus Materie, die um die Fixsterne herumwirbelt. Er illustriert sein Modell mit Abbildung 5.1. Sie zeigt eine Zerlegung des Raums in konvexe Regionen. Im Zentrum einer jeden Region befindet sich ein Fixstern; die Materiewirbel sind durch gepunktete Linien von verschiedenen Seiten dargestellt.

Descartes' Modell

Mag diese Theorie auch unzutreffend sein, so enthält sie doch den Keim der folgenden wichtigen Vorstellung: Gegeben ist ein Raum (hier der $\mathbb{R}^3$) und in ihm eine Menge S von Objekten (die Fixsterne), die auf ihre Umgebung irgendeine Art von Einfluß ausüben (z. B. Anziehung). Dann kann man die folgende *Zerlegung des Raums in Regionen* betrachten: Zu jedem Objekt p in S bildet man die Region $VR(p, S)$ derjenigen Punkte des Raums, für die der von p ausgeübte Einfluß am stärksten ist (bei Descartes sind das die Wirbel).

Zerlegung des Raums in Regionen

Die Bedeutung dieses Ansatzes liegt in seiner Allgemeinheit: Die Begriffe *Raum*, *Objekt* und *Einfluß* sind ja zunächst variabel! Es ist daher nicht erstaunlich, daß dieser Ansatz unter verschiedenen Namen in vielen Wissenschaften außerhalb der Mathematik und Informatik wiederentdeckt wurde, zum Beispiel in der Biologie, Chemie, Kristallographie, Meteorologie und Physiologie.

[1] Nach ihm wurde später das kartesische Koordinatensystem benannt.

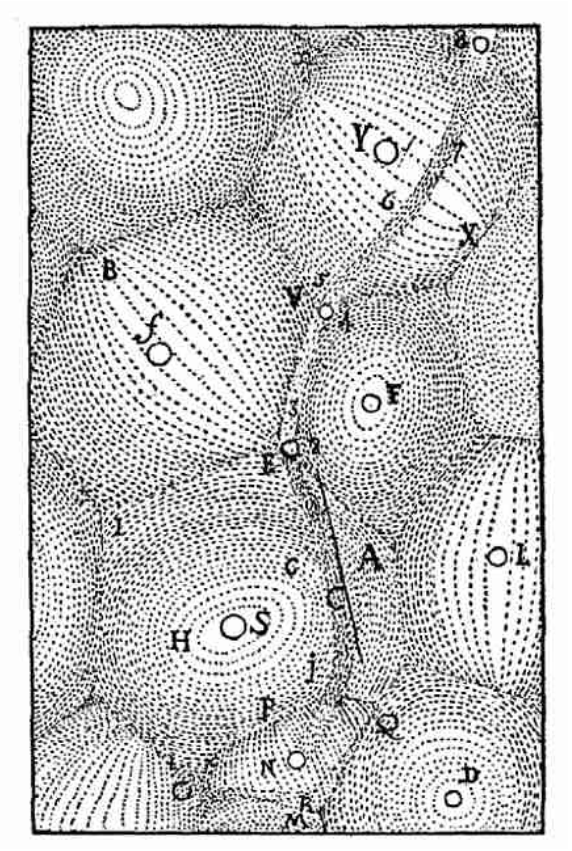


Abb. 5.1 Die Zerlegung des Sonnensystems in Wirbel nach Descartes.

Dirichlet-Zerlegungen

Voronoi-Diagramme

In der Mathematik haben erst Dirichlet und dann Voronoi bei ihren Arbeiten über quadratische Formen diesen Ansatz systematisch untersucht. Ihre Objekte waren regelmäßig angeordnete Punkte im $\mathbb{R}^d$, und der Einfluß, den p auf einen Punkt x des $\mathbb{R}^d$ ausübt, ist umgekehrt proportional zum *Abstand* $|px|$. Die entstehenden Zerlegungen wurden *Dirichlet-Zerlegungen* und *Voronoi-Diagramme* genannt; die letzte Bezeichnung hat sich in der Algorithmischen Geometrie durchgesetzt und wird auch im folgenden verwendet.

In der Algorithmischen Geometrie wurde das Voronoi-Diagramm zuerst von Shamos und Hoey [135] betrachtet.

Wer sich für die Geschichte des Voronoi-Diagramms und seine vielfältigen Anwendungen innerhalb und außerhalb der Informatik interessiert, sei auf das Buch von Okabe et al. [112] und die Übersichtsartikel von Aurenhammer [9] und von Aurenhammer

und Klein [10] hingewiesen.

Wir beschäftigen uns in diesem Kapitel mit den strukturellen Eigenschaften des Voronoi-Diagramms und mit seiner herausragenden Rolle bei der Lösung geometrischer Distanzprobleme. In Kapitel 6 geht es dann um schnelle Algorithmen zur Konstruktion des Voronoi-Diagramms von n Punkten in der Ebene. Erstaunlicherweise eignen sich dazu die meisten der uns inzwischen bekannten algorithmischen Paradigmen: randomisierte inkrementelle Konstruktion, geometrische Transformation, *sweep* und *divide-and-conquer* lassen sich zum Aufbau des Voronoi-Diagramms verwenden. Soviel sei schon verraten: Sie alle führen zu optimalen Algorithmen mit Laufzeit $O(n \log n)$ und linearem Platzbedarf.

Ausblick

5.2 Definition und Struktur des Voronoi-Diagramms

In diesem Abschnitt ist der in der Einführung erwähnte *Raum* der $\mathbb{R}^2$, und die *Objekte* sind die Elemente einer n-elementigen Menge $S = \{p, q, r, \ldots\}$ von Punkten im $\mathbb{R}^2$. Als Maß für den *Einfluß*, den ein Punkt $p = (p_1, p_2)$ aus S auf einen Punkt $x = (x_1, x_2)$ ausübt, verwenden wir den *euklidischen Abstand*[2]

euklidischer Abstand

$$|px| = \sqrt{|p_1 - x_1|^2 + |p_2 - x_2|^2}.$$

Zunächst erinnern wir an den Bisektor von zwei Punkten $p, q \in S$, wie er schon in Kapitel 1 auf Seite 25 eingeführt wurde; es handelt sich um die Menge

$$B(p, q) = \{x \in \mathbb{R}^2; |px| = |qx|\}$$

aller Punkte des $\mathbb{R}^2$, die zu p und q denselben Abstand haben.

Der *Bisektor* $B(p, q)$ besteht aus genau den Punkten x auf der Mittelsenkrechten des Liniensegments pq. Diese Mittelsenkrechte zerlegt $\mathbb{R}^2$ in zwei offene Halbebenen; eine von ihnen ist

Bisektor $B(p, q)$

$$D(p, q) = \{x \in \mathbb{R}^2; |px| < |qx|\},$$

die andere

$$D(q, p) = \{x \in \mathbb{R}^2; |px| > |qx|\}.$$

Offenbar ist p in $D(p, q)$ und q in $D(q, p)$ enthalten. Wir nennen

$$VR(p, S) = \bigcap_{q \in S \setminus \{p\}} D(p, q)$$

[2] Im folgenden Sinn: je kleiner der Abstand, desto größer der Einfluß.

Voronoi-Region

die *Voronoi-Region* von p bezüglich S.[3] Als Durchschnitt von $n-1$ offenen Halbebenen ist die Voronoi-Region von p offen und konvex, aber nicht notwendig beschränkt.

Übungsaufgabe 5.1 Kann es vorkommen, daß die Voronoi-Region $VR(p, S)$ nur aus dem Punkt p besteht?

Zwei verschiedene Punkte $p, q \in S$ haben disjunkte Voronoi-Regionen, denn kein x kann gleichzeitig zu $D(p, q)$ und $D(q, p)$ gehören.

Die Voronoi-Region $VR(p, S)$ von p besteht aus allen Punkten der Ebene, denen p näher ist als irgendein anderer Punkt aus S. Denkt man sich sämtliche Voronoi-Regionen aus der Ebene entfernt, bleiben daher genau diejenigen Punkte des $\mathbb{R}^2$ übrig, die keinen eindeutigen, sondern zwei oder mehr nächste Nachbarn in S besitzen. Wir nennen diese Punktmenge das *Voronoi-Diagramm* $V(S)$ von S.

Voronoi-Diagramm $V(S)$

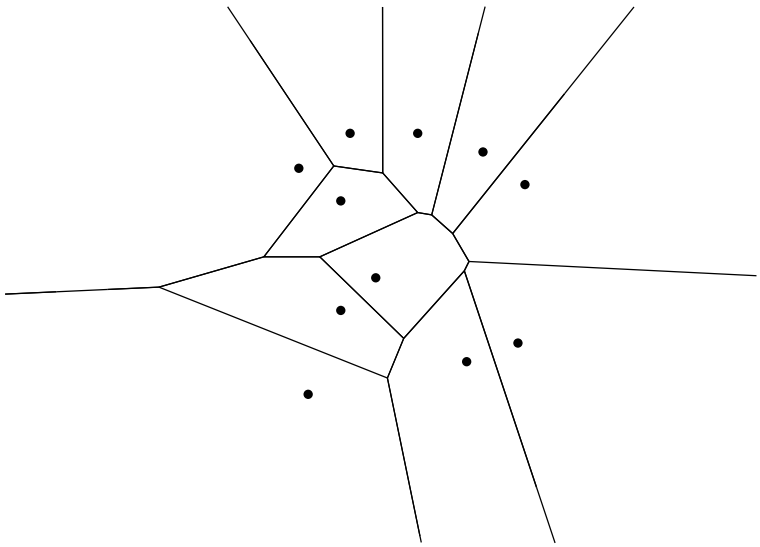


Abb. 5.2 Ein Voronoi-Diagramm von elf Punkten.

Abbildung 5.2 zeigt ein Voronoi-Diagramm von elf Punkten. Es zerlegt die Ebene in elf konvexe Gebiete: zu jedem Punkt aus S seine Voronoi-Region. Ein Punkt x auf dem gemeinsamen Rand

[3] Besteht S nur aus den beiden Punkten p und q, gilt daher $D(p, q) = VR(p, S)$.

von zwei Voronoi-Regionen $VR(p,S)$ und $VR(q,S)$ muß zum Bisektor $B(p,q)$ von p und q gehören, denn es ist

$$\overline{VR(p,S)} \cap \overline{VR(q,S)} \subseteq \overline{D(p,q)} \cap \overline{D(q,p)} = B(p,q).$$

Weil die Voronoi-Regionen konvex sind, kann für je zwei Punkte p und q aus S höchstens *ein einziges* Stück des Bisektors $B(p,q)$ zum gemeinsamen Rand gehören. Solch ein gemeinsames Randstück nennen wir eine *Voronoi-Kante*, wenn es aus mehr als einem Punkt besteht. Endpunkte von Voronoi-Kanten heißen *Voronoi-Knoten*.

Voronoi-Kante

Voronoi-Knoten

Es gibt einen einfachen Weg, um die Rolle eines jeden Punktes x der Ebene im Voronoi-Diagramm $V(S)$ zu bestimmen: Wir beobachten, wie sich ein Kreis $C(x)$ mit Mittelpunkt x langsam vergrößert. Zu irgendeinem Zeitpunkt wird der Rand von $C(x)$ zum ersten Mal auf einen oder mehrere Punkte aus S treffen; dies sind gerade die nächsten Nachbarn von x in S. Nun kommt es auf die Anzahl der getroffenen Punkte an; siehe Abbildung 5.3.

wozu gehört Punkt x?

Lemma 5.1 *Sei x ein Punkt in der Ebene, und sei $C(x)$ der sich von x ausbreitende Kreis. Dann gilt:*

$$\begin{aligned}
&C(x) \text{ trifft zuerst nur auf } p &&\Longleftrightarrow x \text{ liegt in der Voronoi-Region von } p,\\
&C(x) \text{ trifft zuerst nur auf } p,q &&\Longleftrightarrow x \text{ liegt auf der Voronoi-Kante zwischen den Regionen von } p \text{ und } q,\\
&C(x) \text{ trifft zuerst genau auf } p_1,\ldots,p_k \text{ mit } k \geq 3 &&\Longleftrightarrow x \text{ ist ein Voronoi-Knoten, an den die Regionen von } p_1,\ldots,p_k \text{ angrenzen.}
\end{aligned}$$

Im letzten Fall entspricht die Ordnung der Punkte $p_1,\ldots,p_k$ auf dem Rand von $C(x)$ der Ordnung ihrer Voronoi-Regionen um x.

Beweis. Die erste Aussage besagt, daß die Voronoi-Region eines Punktes p aus genau denjenigen Punkten x besteht, die p als einzigen nächsten Nachbarn haben. Das hatten wir schon nach der Definition der Voronoi-Regionen festgestellt. Wir brauchen deshalb das Lemma nur noch für alle Punkte x mit zwei oder mehr nächsten Nachbarn in S zu beweisen.[4]

ein nächster Nachbar

Angenommen, p und q sind die einzigen nächsten Nachbarn von x. Dann gilt

zwei nächste Nachbarn

$$\begin{aligned}
x &\in B(p,q) \subseteq \overline{D(p,q)} \text{ und}\\
x &\in D(p,r) \text{ für alle } r \neq p,q,
\end{aligned}$$

[4] D. h. für alle Punkte von $V(S)$.

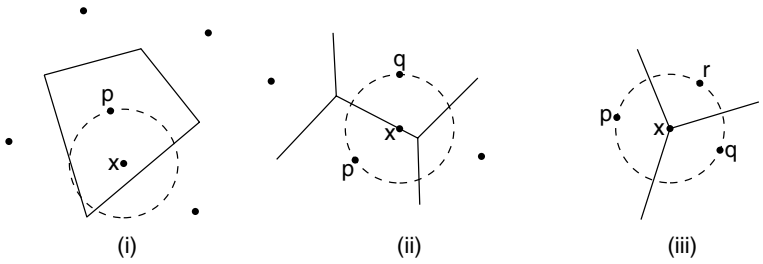


Abb. 5.3 In (i) gehört x zur Region von p, in (ii) liegt x auf einer Kante des Voronoi-Diagramms, und in (iii) ist x ein Voronoi-Knoten.

und es folgt

$$\begin{aligned} x &\in \overline{D(p,q)} \cap \bigcap_{r \neq p,q} D(p,r) \\ &\subseteq \bigcap_{r \neq p} \overline{D(p,r)} = \overline{\bigcap_{r \neq p} D(p,r)} \\ &= \overline{VR(p,S)}. \end{aligned}$$

Bei der vorletzten Gleichung haben wir ausgenutzt, daß es sich bei den Mengen $D(p,r)$ um offene Halbebenen handelt, die alle den Punkt p enthalten. Liegt nämlich y im Durchschnitt ihrer Abschlüsse, so auch das Liniensegment py; bis auf seinen Endpunkt y ist es sogar im Durchschnitt der offenen Halbebenen enthalten. Also liegt y im Abschluß des Durchschnitts.[5]

Ganz analog ergibt sich, daß x auch im Abschluß der Region von q liegt.

Dieselbe Überlegung läßt sich statt für x für jeden Punkt auf dem Bisektor $B(p,q)$ durchführen, der so nahe bei x liegt, daß auch er p, q als einzige nächste Nachbarn hat. Folglich liegt x im Innern eines Segments von $B(p,q)$, das zum gemeinsamen Rand der Regionen von p und q gehört, d. h. auf einer Voronoi-Kante.

drei oder mehr nächste Nachbarn

Schließlich nehmen wir an, daß x drei oder mehr nächste Nachbarn $p_1, p_2, \ldots, p_k$ in S besitzt, die in dieser Reihenfolge auf dem Rand von $C(x)$ liegen; siehe Abbildung 5.4. Wie oben folgt zunächst

$$x \in B(p_i, p_j) \quad \text{und} \quad x \in \overline{VR(p_i, S)}$$

für alle i und j mit $1 \leq i \neq j \leq k$. Der Punkt x kann nicht im Abschluß weiterer Regionen liegen, denn aus

$$x \in \overline{VR(r,S)}$$

[5] Im allgemeinen ist aber der Durchschnitt von Abschlüssen endlich vieler Mengen eine echte Obermenge vom Abschluß ihres Durchschnitts.

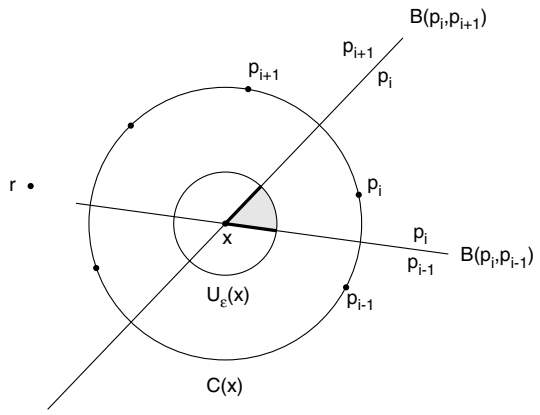


Abb. 5.4 Die Umgebung eines Punktes x im dritten Fall von Lemma 5.1.

folgt

$$x \in \overline{VR(r,S)} \cap \overline{VR(p_1,S)} \subset B(r,p_1),$$

also kann r nur ein nächster Nachbar von x, d. h. eines der p_i sein. Es gibt daher eine Umgebung $U_\varepsilon(x)$, in der nur die Voronoi-Regionen der p_i vertreten sind.

Wegen

$$VR(p_i,S) \subseteq D(p_i,p_{i-1}) \cap D(p_i,p_{i+1})$$

ist die Region von p_i innerhalb von $U_\varepsilon(x)$ in dem in Abbildung 5.4 grau dargestellten „Tortenstück“ enthalten. Diese Überlegung gilt für *jeden* Punkt p_i. Andererseits kann kein Punkt $r \notin \{p_1,\ldots,p_k\}$ etwas von der Torte $U_\varepsilon(x)$ abbekommen. Also muß die gesamte Torte – bis auf die Schnitte – zwischen den p_i aufgeteilt sein. Das bedeutet: Jede Voronoi-Region $VR(p_i,S)$ füllt ihr Tortenstück

Tortenstück

$$U_\varepsilon(x) \cap D(p_i,p_{i-1}) \cap D(p_i,p_{i+1})$$

voll aus!

Für die Punkte aus Abbildung 5.4 sieht das Voronoi-Diagramm innerhalb von $U_\varepsilon(x)$ also so aus wie in Abbildung 5.5. Insbesondere enthält jeder Bisektor $B(p_i,p_{i+1})$ eine Voronoi-Kante mit Endpunkt x; daher ist x ein Voronoi-Knoten.

Wir können also an der Zahl der nächsten Nachbarn eindeutig feststellen, ob ein Punkt x im Innern einer Region enthalten ist, ob er im Innern einer Voronoi-Kante liegt oder ob er ein Voronoi-Knoten ist. Damit ist Lemma 5.1 bewiesen. □

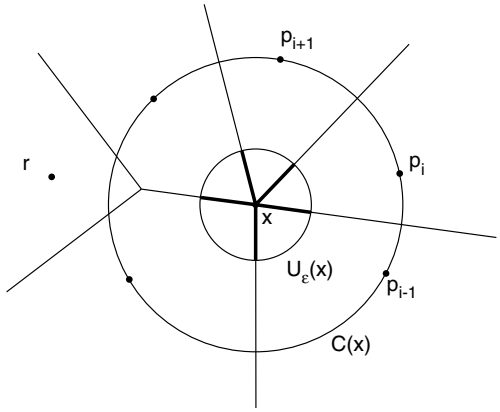


Abb. 5.5 In der Umgebung $U_\varepsilon(x)$ treffen die Voronoi-Regionen der Punkte p_i wie Tortenstücke aufeinander.

Wir haben das Voronoi-Diagramm als die Menge aller Punkte x mit zwei oder mehr nächsten Nachbarn in S definiert. Daneben gibt es noch andere Definitionen, die im Fall der euklidischen Metrik hierzu äquivalent sind:

Übungsaufgabe 5.2 Man zeige, daß für das Voronoi-Diagramm folgende äquivalente Darstellungen gelten:

$$\begin{aligned} V(S) &= \bigcup_{p \in S} \partial\, VR(p,S) \\ &= \bigcup_{p,q \in S,\ p \neq q} \overline{VR(p,S)} \cap \overline{VR(q,S)}. \end{aligned}$$

Hierbei bezeichnet wieder ∂A den Rand einer Menge A.

Knotengrad 3 bei 3 Punkten auf Kreisrand

Wenn von den Punkten aus S nie mehr als drei auf einem gemeinsamen Kreisrand liegen, so haben alle Voronoi-Knoten den Grad 3.

Während man durch drei nicht-kolineare Punkte in der Ebene stets einen Kreis legen kann, geht das für vier beliebige Punkte im allgemeinen nicht. Man verlangt daher manchmal, daß die Punkte aus S sich in *allgemeiner Lage* befinden sollen, um solche Phänomene auszuschließen. Wir werden von dieser leicht mißzuverstehenden Redeweise keinen Gebrauch machen und lieber explizit sagen, welche Eigenschaften von S vorausgesetzt werden.

globale Eigenschaften von $V(S)$

Nachdem wir beim Beweis von Lemma 5.1 die lokalen Eigenschaften des Voronoi-Diagramms in der Nähe eines Punktes x untersucht haben, wenden wir uns nun den *globalen Eigenschaften*

des Voronoi-Diagramms zu. Bei einem Blick auf Abbildung 5.2 fällt auf, daß manche Punkte *unbeschränkte Voronoi-Regionen* besitzen. Hier gilt eine interessante Beziehung, die man durch Experimentieren mit dem Applet

unbeschränkte Voronoi-Regionen

```
http://www.geometrylab.de/VoroGlide/
```

selbst schnell herausfinden kann:

Lemma 5.2 *Genau dann hat ein Punkt $p \in S$ eine unbeschränkte Voronoi-Region, wenn er auf dem Rand der konvexen Hülle von S liegt.*

konvexe Hülle

Beweis. Angenommen, $VR(p, S)$ ist unbeschränkt. Dann existiert ein anderer Punkt q in S, so daß $V(S)$ ein unbeschränktes Stück des Bisektors $B(p, q)$ als Kante enthält. Wir betrachten einen Punkt x auf dieser Kante und den Kreis $C(x)$ durch p und q mit Mittelpunkt x; siehe Abbildung 5.6. Wenn wir mit x auf dieser Kante nach rechts gegen ∞ wandern, wird der in der Halbebene R enthaltene Teil von $C(x)$ ständig größer und erreicht irgendwann jeden Punkt in R. Läge also ein Punkt $r \in S$ in der Halbebene R, würde der Rand von $C(x)$ irgendwann auf r stoßen. Nach Lemma 5.1 wäre dieser Punkt x dann ein Endpunkt der Voronoi-Kante aus $B(p, q)$; diese war aber als unbeschränkt vorausgesetzt! Also müssen alle Punkte aus $S \setminus \{p, q\}$ in der *linken* Halbebene L liegen. Dann gehört das Liniensegment pq zum Rand der konvexen Hülle $ch(S)$.

Umgekehrt: Sind p und q zwei benachbarte Ecken von $ch(S)$ und liegt S links von der Senkrechten durch p und q, kann $R \cap C(x)$ für kein $x \in B(p, q)$ einen Punkt aus S enthalten. Aber auch $L \cap C(x)$ enthält keine Punkte aus S, wenn x nur weit genug rechts liegt. Folglich bildet ein unbeschränktes Stück von $B(p, q)$ eine Voronoi-Kante, und auch $VR(p, S)$ und $VR(q, S)$ sind unbeschränkt. □

Übungsaufgabe 5.3 Man zeige, daß das Voronoi-Diagramm $V(S)$ immer zusammenhängend ist, wenn die Punkte aus S nicht auf einer gemeinsamen Geraden liegen.

Man kann das technische Problem eines nicht zusammenhängenden Voronoi-Diagramms vermeiden, indem man sich vorstellt, der „interessante“ Teil von $V(S)$ sei von einem einfachen geschlossenen Weg Γ umschlossen, der so weit außen verläuft, daß er nur unbeschränkte Voronoi-Kanten kreuzt, die dort abgeschnitten werden; siehe Abbildung 5.7. Wir sprechen dann vom *beschränkten* Voronoi-Diagramm $V_0(S)$.

beschränktes Diagramm $V_0(S)$

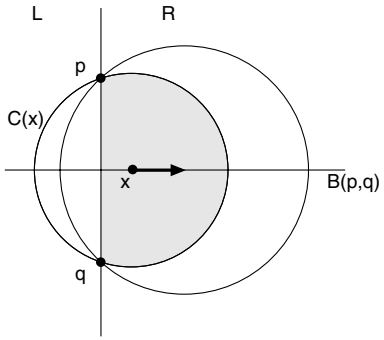


Abb. 5.6 Wenn der Punkt x sich auf dem Bisektor $B(p,q)$ nach rechts bewegt, schrumpft der Durchschnitt des Kreises $C(x)$ mit der linken Halbebene L, während $C(x) \cap R$ wächst.

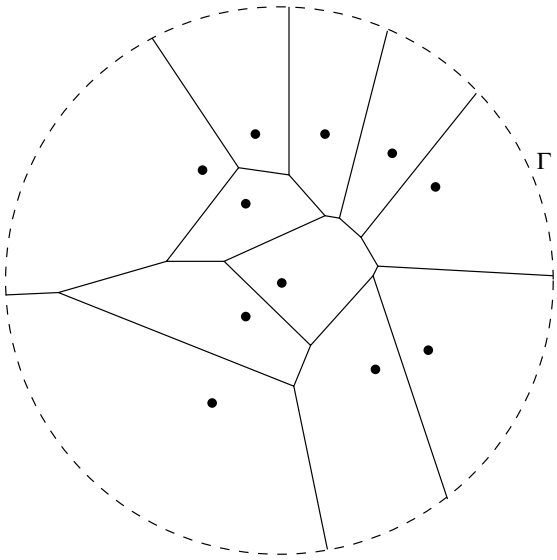


Abb. 5.7 Ein beschränktes Voronoi-Diagramm $V_0(S)$.

Hier haben wir nun einen kreuzungsfreien geometrischen Graphen vor uns, wie er in Kapitel 1 auf Seite 13 eingeführt wurde. $V_0(S)$ hat $n+1$ Flächen, nämlich eine Voronoi-Region für jeden Punkt $p \in S$ und das Äußere von Γ. Jeder Knoten hat mindestens den Grad 3; für die echten Voronoi-Knoten im Innern von Γ ergibt sich das aus Lemma 5.1, und die Knoten auf Γ, die durch Abschneiden einer nach ∞ führenden Voronoi-Kante entstehen, haben trivialerweise den Grad 3. Also folgt aus Korollar 1.2 auf Seite 15 zunächst für $V_0(S)$, aber damit auch für $V(S)$:

Theorem 5.3 *Das Voronoi-Diagramm von n Punkten in der Ebene hat $O(n)$ viele Knoten und Kanten. Der Rand einer Voronoi-Region besteht im Mittel aus höchstens sechs Kanten.*

Zur Speicherung des beschränkten Voronoi-Diagramms eignen sich die in Abschnitt 1.2.2 auf Seite 19 eingeführten Datenstrukturen. Dabei braucht der gedachte Weg Γ natürlich nicht explizit dargestellt zu werden; wir behandeln jedes Stück von Γ wie eine gewöhnliche Kante, lassen aber die Koordinaten der Endpunkte fort.

Bei dieser Darstellung kann man bequem im Uhrzeigersinn um den geschlossenen Weg Γ herumwandern und alle unbeschränkten Voronoi-Regionen besuchen; nach Lemma 5.2 gehören sie gerade zu denjenigen Punkten aus S, die auf dem Rand der konvexen Hülle liegen. Damit haben wir die erste nützliche Anwendung des Voronoi-Diagramms erkannt:

Anwendung: konvexe Hülle

Theorem 5.4 *Aus dem Voronoi-Diagramm $V(S)$ einer n-elementigen Punktmenge S läßt sich in Zeit $O(n)$ die konvexe Hülle von S ableiten.*

Obwohl wir uns erst in Kapitel 6 mit der Konstruktion des Voronoi-Diagramms befassen wollen, halten wir schon einmal fest, was sich hieraus und aus Lemma 4.2 auf Seite 157 als untere Schranke ergibt.

Korollar 5.5 *Das Voronoi-Diagramm von n Punkten in der Ebene zu konstruieren hat die Zeitkomplexität $\Omega(n \log n)$.*

5.3 Anwendungen des Voronoi-Diagramms

Jetzt wollen wir ein paar konkrete Distanzprobleme betrachten, die sich mit Hilfe des Voronoi-Diagramms effizient lösen lassen.

5.3.1 Das Problem des nächsten Postamts

Gegeben seien n Postämter, dargestellt als Punkte p_i in der Ebene. Für einen beliebigen Anfragepunkt x soll schnell festgestellt werden, welches der Postämter am nächsten zu x liegt.

Es gibt in der Algorithmischen Geometrie ein allgemeines Verfahren, um Anfrageprobleme wie dieses zu lösen: Man bereitet sich auf die Anfrage vor, indem man diejenigen möglichen Anfrageobjekte zu Klassen zusammenfaßt, für die die Antwort dieselbe ist.[6]

[6] Natürlich ist dies eine *Äquivalenzrelation*.

Wenn dann eine konkrete Anfrage x beantwortet werden muß, genügt es festzustellen, zu welcher Klasse x gehört. Die Antwort kann dann direkt der Beschreibung der Klasse entnommen werden.

Ortsansatz

Dieses Vorgehen nennt man den *Ortsansatz* (im Englischen: *locus approach*). Ein Beispiel hatten wir bereits auf Seite 190 bei der Diskussion der Sichtregionen gesehen.

In unserem Postamtproblem bestehen die Klassen gerade aus denjenigen Punkten x, für die dasselbe Postamt das nächstgelegene ist, also aus den Voronoi-Regionen $VR(p_i, S)$! Nur die Punkte x, die auf dem Voronoi-Diagramm $V(S)$ liegen, haben mehrere nächste Postämter; wir könnten sie zum Beispiel stets dem Postamt p_i zuordnen, das den kleinsten Index i hat.

Lokalisierungsproblem

Damit bleibt das *Lokalisierungsproblem*, zu gegebenem Voronoi-Diagramm $V(S)$ und einem beliebigen Anfragepunkt x schnell die Voronoi-Region zu bestimmen, in der x enthalten ist. Im Englischen ist diese Aufgabe als *point location problem* bekannt.

Streifenmethode

Eine ganz einfache Lösung besteht in der sogenannten *Streifenmethode*. Dabei wird durch jeden Knoten des unbeschränkten Voronoi-Diagramms $V(S)$ eine waagerechte Gerade gezogen. Hierdurch wird die Ebene in parallele Streifen zerlegt; siehe Abbildung 5.8. Wenn nun für einen Anfragepunkt $x = (x_1, x_2)$ seine Voronoi-Region bestimmt werden soll, führen wir zunächst eine binäre Suche nach x_2 in den Y-Koordinaten der Waagerechten durch und bestimmen damit den Streifen, der x enthält. Innerhalb eines Streifens können keine zwei Voronoi-Kanten aufeinandertreffen, denn dort wäre sonst ein Voronoi-Knoten mit einer weiteren waagerechten Geraden. Also können wir eine binäre Suche nach x in den Kantensegmenten des Streifens ausführen, in dem x enthalten ist.

Nach Theorem 5.3 enthält $V(S)$ nur $O(n)$ viele Knoten und Kanten, und eine Kante kann in einem Streifen mit höchstens *einem* Segment vertreten sein. Also erfordern die beiden binären Suchprozesse jeweils nur $O(\log n)$ viel Zeit, und wir erhalten folgende Lösung des Postamtproblems:

nächstes Postamt in Zeit $O(\log n)$

Theorem 5.6 *Zu n Punkten in der Ebene kann man mit Hilfe des Voronoi-Diagramms eine Struktur aufbauen, mit der sich für einen beliebigen Anfragepunkt x der nächstgelegene der n Punkte in Zeit $O(\log n)$ bestimmen läßt.*

Nicht optimal sind der Speicherplatzbedarf und die Vorbereitungszeit der beschriebenen Methode: Eine Voronoi-Kante kann ja viele waagerechte Streifen schneiden und in jedem ein Kantensegment erzeugen, das dort gespeichert werden muß. Die folgende

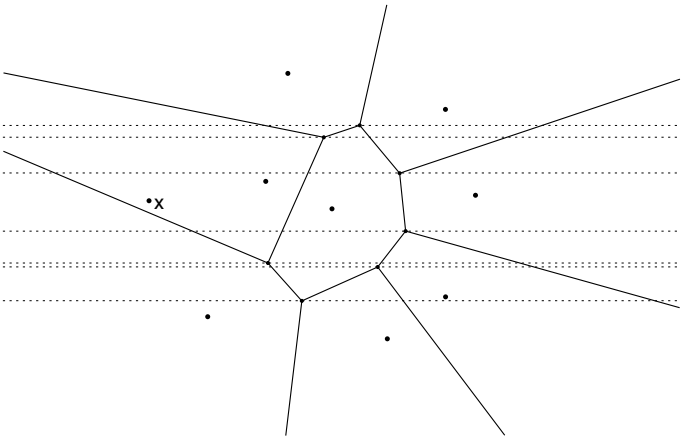


Abb. 5.8 Hat man durch jeden Knoten eine Waagerechte gelegt, läßt sich ein Anfragepunkt x schnell lokalisieren.

Aufgabe zeigt, daß der Aufwand für die Konstruktion der Suchstruktur deshalb quadratisch in der Anzahl der Punkte sein kann.

Übungsaufgabe 5.4 Man zeige, daß die Voronoi-Kanten eines Diagramms von n Punkten in $\Theta(n^2)$ Segmente zerfallen können, wenn man durch die Voronoi-Knoten waagerechte Geraden legt.

Es gibt aber effizientere Ansätze: Ist $V(S)$ vorhanden, kann man auch in linearer Zeit und linearem Speicherplatz eine Suchstruktur konstruieren, mit deren Hilfe sich ein beliebiger Anfragepunkt in Zeit $O(\log n)$ im Voronoi-Diagramm lokalisieren läßt; siehe Edelsbrunner [51] und Seidel [132].

5.3.2 Die Bestimmung aller nächsten Nachbarn

Erinnern wir uns an das am Anfang auf Seite 2 vorgestellte Problem, zu jedem Punkt einer ebenen Punktmenge S seinen nächsten Nachbarn in S zu bestimmen! Das Voronoi-Diagramm verhilft uns nun zu einer effizienten Lösung. Sie beruht auf folgendem Sachverhalt:

alle nächsten Nachbarn

Lemma 5.7 *Sei $S = P \cup Q$ eine Zerlegung der endlichen Punktmenge S in zwei disjunkte, nicht-leere Teilmengen P und Q. Seien $p_0 \in P$ und $q_0 \in Q$ so gewählt, daß*

$$|p_0 q_0| = \min_{p \in P,\, q \in Q} |pq|$$

gilt. Dann haben die Regionen von p_0 und q_0 im Voronoi-Diagramm $V(S)$ eine gemeinsame Kante, und diese wird vom Liniensegment p_0q_0 geschnitten.

Beweis. Andernfalls enthielte das Liniensegment p_0q_0 einen Punkt z aus dem Abschluß $\overline{VR(r,S)}$ der Voronoi-Region eines Punktes $r \neq p_0, q_0$. Angenommen, r gehört zu Q; der Fall $r \in P$ ist symmetrisch.

Da z im Abschluß der Region von r liegt, ist $|rz| \leq |q_0z|$. Folglich gilt

$$\begin{aligned} |p_0r| &\leq |p_0z| + |zr| && \text{(Dreiecksungleichung)} \\ &\leq |p_0z| + |zq_0| = |p_0q_0| \leq |p_0r| && \text{(Minimalität von } |p_0q_0|\text{)}. \end{aligned}$$

Also muß überall die Gleichheit gelten, und es folgt $|p_0r| = |p_0q_0|$ und $|zr| = |zq_0|$. Wenn aber p_0 tatsächlich auf dem Bisektor $B(q_0, r)$ liegt, ist das Innere des Liniensegments q_0p_0 und damit auch der Punkt z ganz in $D(q_0, r)$ enthalten, so daß $|zq_0| < |zr|$ gelten muß. Widerspruch! □

Wenden wir für einen beliebigen Punkt $p \in S$ das Lemma 5.7 auf $P = \{p\}$ und $Q = S \setminus \{p\}$ an, ergibt sich folgende Aussage:

Korollar 5.8 *Jeder nächste Nachbar von p in S sitzt im Voronoi-Diagramm in einer Nachbarzelle, d. h. in einer Voronoi-Region, die mit $VR(p, S)$ eine gemeinsame Kante besitzt.*

Übungsaufgabe 5.5 Man gebe einen direkten Beweis für Korollar 5.8, der ohne Benutzung von Lemma 5.7 auskommt.

Also können wir die Bestimmung der nächsten Nachbarn sehr effizient anhand des Voronoi-Diagramms vornehmen.

Theorem 5.9 *Ist das Voronoi-Diagramm $V(S)$ vorhanden, läßt sich in Zeit $O(n)$ für alle $p \in S$ ihr nächster Nachbar*[7] *in S bestimmen.*

Beweis. Nach Korollar 5.8 genügt es, für jeden Punkt $p \in S$ alle Punkte q in den Nachbarregionen von $VR(p, S)$ aufzusuchen und *unter diesen* den Punkt (oder alle Punkte) mit minimalem Abstand zu p zu berichten. Dabei muß man insgesamt zweimal über jede Voronoi-Kante $\subseteq B(p, q)$ schauen: einmal bei der Bestimmung des nächsten Nachbarn von p und das zweite Mal, wenn der nächste Nachbar von q ermittelt wird. Weil es nach Theorem 5.3 nur $O(n)$ viele Kanten in $V(S)$ gibt, folgt die Behauptung. □

[7] Sollte es in S Punkte mit mehreren nächsten Nachbarn geben, können wir sie alle in Zeit $O(n)$ ausgeben.

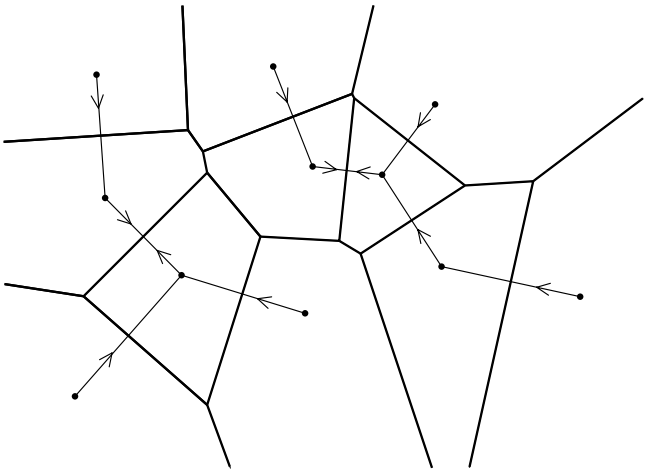


Abb. 5.9 Die nächsten Nachbarn befinden sich in benachbarten Voronoi-Regionen.

In Abbildung 5.9 sind noch einmal die 11 Punkte aus Abbildung 1.1 auf Seite 3 zu sehen, ihre nächsten Nachbarn und das Voronoi-Diagramm der Punkte.

Hat man für jeden Punkt aus S einen nächsten Nachbarn in S bestimmt, läßt sich daraus in linearer Zeit der minimale Abstand und ein *dichtestes Punktepaar* ermitteln. Damit ergibt sich eine Alternative zu dem in Abschnitt 2.3.1 vorgestellten Sweep-Verfahren, wenn man das Voronoi-Diagramm schon hat.

dichtestes Punktepaar

Korollar 5.10 *Ein dichtestes Paar von n Punkten in der Ebene läßt sich aus ihrem Voronoi-Diagramm in Zeit $O(n)$ bestimmen.*

5.3.3 Der minimale Spannbaum

Angenommen, ein Netzbetreiber möchte n Orte miteinander verkabeln, die als Punktmenge S in der Ebene dargestellt werden. Dabei soll (ohne Berücksichtigung der Leistung des Netzes) die Gesamtlänge der verlegten Kabelstrecken minimiert werden.

Ein möglicher Ansatz besteht darin, die Kabel längs eines *minimalen Spannbaums* (engl.: *minimum spanning tree*) von S zu verlegen, also eines Graphen mit Knotenmenge S, bei dem die Summe der Längen aller Kanten minimal ist.

minimaler Spannbaum

Abbildung 5.10 zeigt elf Punkte in der Ebene und ihren minimalen Spannbaum. Wie läßt er sich effizient berechnen?

In den meisten Büchern über Algorithmen und Datenstrukturen wird der *Algorithmus von Kruskal* vorgestellt, der für einen

Algorithmus von Kruskal

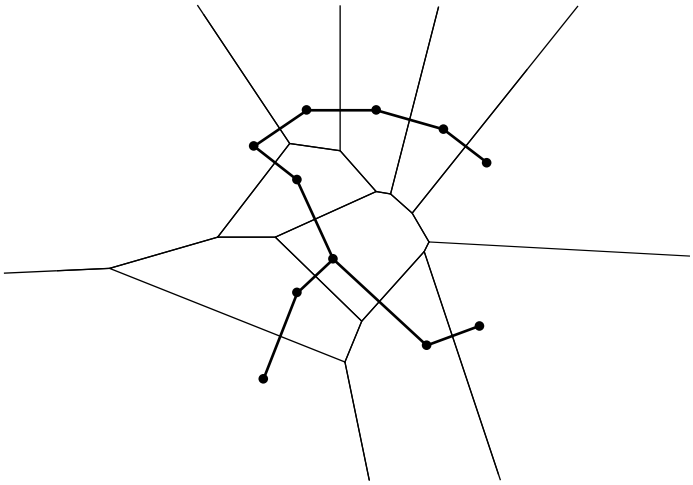


Abb. 5.10 Elf Punkte in der Ebene, ihr Voronoi-Diagramm und ihr minimaler Spannbaum.

beliebigen Graphen $G = (E, V)$ mit gewichteten Kanten in Zeit $O(|E| \log |E|)$ einen spannenden Baum über den Knoten V von G berechnet, bei dem die Summe aller Kantengewichte minimal ist.

Man verwaltet dabei einen Wald von Bäumen über disjunkten Teilmengen der Knotenmenge V; am Anfang enthält jede Teilmenge nur einen einzigen Knoten. Nun werden die Kanten nach aufsteigenden Gewichten sortiert zur Hand genommen. Wenn eine Kante zwei Teilbäume miteinander verbindet, wird sie eingefügt, und die Teilbäume werden verschmolzen. Andernfalls wird sie verworfen.

Kruskal auf allen Segmenten braucht $O(n^2 \log n)$ Zeit

Wir können dieses Verfahren direkt auf unser Problem anwenden, wenn wir als G folgenden Graphen zugrunde legen: Die Knoten sind die n Punkte aus S, die Kanten alle $n(n-1)/2$ vielen Liniensegmente zwischen den Punkten. Damit ergäbe sich eine Laufzeit von $O(n^2 \log n)$.

Geht es vielleicht besser? Betrachten wir noch einmal Abbildung 5.10, in der ja auch das Voronoi-Diagramm der elf Punkte eingezeichnet ist. Es fällt auf, daß alle Kanten des minimalen Spannbaums zwischen solchen Punkten verlaufen, deren Voronoi-Regionen sich eine Kante teilen. Ein Zufall? Nein! Denn der Algorithmus von Kruskal fügt nur solche Liniensegmente ein, die eine kürzeste Verbindung zwischen zwei disjunkten Teilmengen von S schaffen;[8] Lemma 5.7 garantiert aber, daß die Endpunkte solcher Liniensegmente benachbarte Voronoi-Regionen haben.

[8] Alle kürzeren Segmente sind ja schon vorher eingefügt worden, ohne daß die Teilmengen zusammengewachsen sind.

Statt Kruskals Verfahren also auf alle $\Theta(n^2)$ vielen Liniensegmente mit Endpunkten in S anzuwenden, genügt es, die $O(n)$ vielen Segmente zwischen den Voronoi-Nachbarn als Kantenmenge E zu definieren und das Verfahren auf den Graphen $G = (V, E)$ anzuwenden.

Kruskal auf $V(S)$ in $O(n \log n)$ Zeit

Theorem 5.11 *Ist das Voronoi-Diagramm einer Menge S von n Punkten in der Ebene gegeben, läßt sich daraus in Zeit $O(n \log n)$ ein minimaler Spannbaum für S ableiten.*

Cheriton und Tarjan [28] haben gezeigt, wie man diese Zeitschranke zu $O(n)$ verbessern kann; siehe auch Preparata und Shamos [120].

Übungsaufgabe 5.6 Sei S eine endliche Punktmenge in der Ebene. Angenommen, die Abstände aller Punktepaare in S sind paarweise verschieden.

(i) Man beweise: Ist q der nächste Nachbar von p in S, ist pq eine Kante des minimalen Spannbaums von S.

(ii) Man zeige, daß der minimale Spannbaum von S eindeutig bestimmt ist.

Der minimale Spannbaum hat eine sehr nützliche Anwendung auf das *Problem des Handlungsreisenden* (*traveling salesman problem*). Dieses lautet wie folgt: Gegeben sind n Punkte in der Ebene, gesucht ist der kürzeste Rundweg, der alle Punkte einmal besucht. Man kann zeigen, daß eine optimale Lösung dieses Problems NP-hart ist, also vermutlich nicht in polynomialer Zeit gefunden werden kann. Dieses und zahlreiche andere Probleme finden sich in dem Buch von Garey und Johnson [62].

Problem des Handlungsreisenden

Angenommen, der Handlungsreisende berechnet einen minimalen Spannbaum der zu besuchenden Punkte und bewegt sich um den Baum herum, wie in Abbildung 5.11 dargestellt. Damit fährt er gar nicht so schlecht!

Theorem 5.12 *Der Rundweg um einen minimalen Spannbaum ist weniger als doppelt so lang wie ein optimaler Rundweg.*

Beweis. Sei t die Länge des minimalen Spannbaums und W ein optimaler Rundweg[9] mit Länge W_{opt}. Lassen wir aus W eine Kante e zwischen zwei Punkten aus S fort, entsteht ein Spannbaum von S mit Länge kleiner als W_{opt}. Er kann aber nicht kürzer sein als der minimale Spannbaum.

[9] Der optimale Rundweg ist polygonal: Verliefe er zwischen zwei Punkten nicht geradlinig, könnte man ihn dort verkürzen.

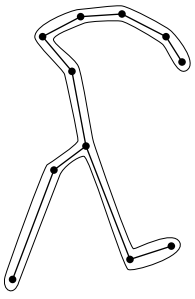


Abb. 5.11 Ein möglicher Rundweg für den Handlungsreisenden.

Folglich gilt

$$t \leq \text{Länge von } W \text{ ohne } e < W_{opt}.$$

Weil der Rundweg um den Spannbaum die Länge $2t$ besitzt, folgt die Behauptung. □

An diesem Ergebnis ist folgendes bemerkenswert: Ein Problem, dessen optimale Lösung praktisch unmöglich oder extrem aufwendig ist, kann trotzdem einfache Lösungen besitzen, die zwar nicht optimal sind, deren Güte sich vom Optimum aber höchstens um einen konstanten Faktor unterscheiden. Solche *Approximationslösungen* sind für die Praxis besonders interessant.

Approximationslösungen

Kommen wir noch einmal auf das eingangs formulierte Problem zurück, bei einem Netzwerk zwischen vorgegebenen Orten die Gesamtlänge des Kabels zu minimieren. Erstaunlicherweise ist der minimale Spannbaum der Orte gar nicht immer die optimale Lösung! Oft läßt sich durch *Einführung künstlicher Knoten* die Gesamtlänge verringern; siehe Abbildung 5.12. Solche zusätzlichen Knoten heißen *Steinerpunkte*, den entstehenden Baum nennt man einen *Steinerbaum* von S. Man kann zeigen, daß ein minimaler Spannbaum höchstens 3/2-mal so lang ist wie ein minimaler Steinerbaum. Leider ist aber die Berechnung minimaler Steinerbäume NP-hart; siehe Garey und Johnson [61]. Eine Fülle von Informationen über spannende Bäume und Netzwerke findet sich in Narasimhan und Smid [109].

Steinerpunkte

5.3.4 Der größte leere Kreis

Angenommen, jemand möchte seinen Wohnsitz in einem bestimmten Gebiet A wählen, möglichst weit entfernt von gewissen Störquellen (Flugplätze, Kraftwerke etc.).

Wenn wir das Gebiet A als ein konvexes Polygon mit m Ecken modellieren und die Störquellen als Elemente einer Punktmenge

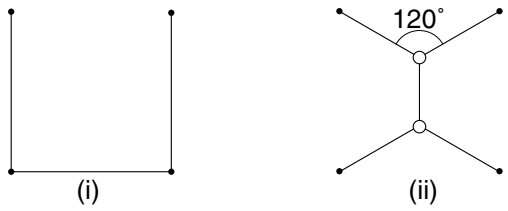


Abb. 5.12 Für die vier Ecken des Einheitsquadrats hat der minimale Spannbaum die Länge 3. Durch Einführung von zwei Steinerpunkten ergibt sich in (ii) die Länge $1 + \sqrt{3} \approx 2.732$.

$S = \{p_1, \ldots, p_n\}$, besteht das Problem offenbar darin, unter allen Kreisen mit Mittelpunkt in A einen solchen mit größtem Radius zu finden, der keinen Punkt aus S im Innern enthält. Anders gesagt: Gesucht wird ein *maximaler leerer Kreis* mit Mittelpunkt in A. Drei Beispiele sind in Abbildung 5.13 gezeigt.

maximaler leerer Kreis

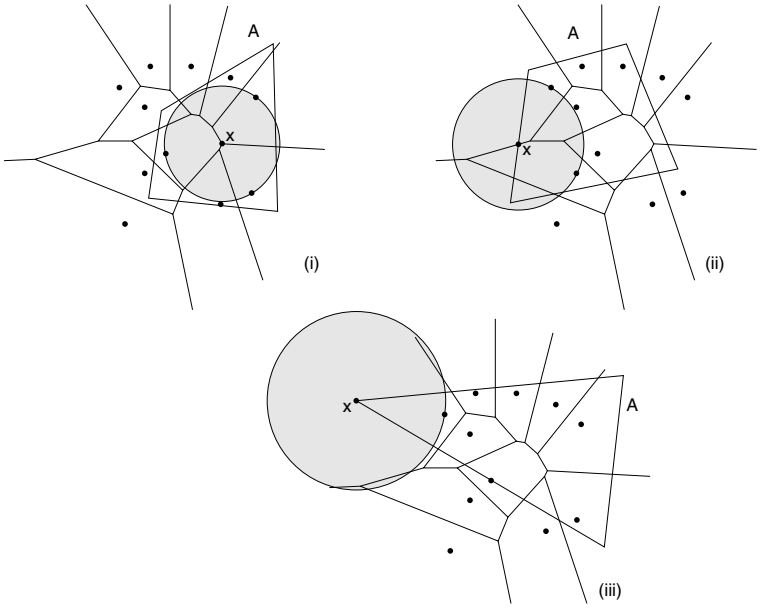


Abb. 5.13 Maximale leere Kreise mit Mittelpunkten in konvexen Polygonen A.

Lemma 5.13 *Sei $C(x)$ der größte Kreis mit im Polygon A liegendem Mittelpunkt x, der keinen Punkt aus S in seinem Innern enthält. Dann ist x ein Voronoi-Knoten von $V(S)$, ein Schnitt-*

punkt einer Voronoi-Kante mit dem Rand von A oder ein Eckpunkt von A.

Beweis. Sei $x \in A$ beliebig und $C(x)$ ein Kreis um x, der kein $p \in S$ enthält. Wir zeigen, daß wir $C(x)$ vergrößern und dabei gegebenenfalls verschieben können, bis eine der drei genannten Eigenschaften auf den Mittelpunkt x zutrifft. Daraus folgt dann, daß der größte leere Kreis selbst auch eine dieser Eigenschaften hat.

wachsender Kreis

Zunächst wird $C(x)$ bei festem x so lange vergrößert, bis der Rand mindestens einen Punkt aus S berührt. Liegen nun gleich drei Punkte auf $\partial C(x)$, ist x ein Voronoi-Knoten; siehe Lemma 5.1. Sind es nur zwei, etwa p und q, liegt x auf einer Voronoi-Kante von $V(S)$, die Teil vom Bisektor $B(p, q)$ ist. In diesem Fall schieben wir x entlang $B(p, q)$ fort vom Schnittpunkt mit pq und halten gleichzeitig den Kontakt von $\partial C(x)$ mit p, q aufrecht, bis der Rand des Kreises einen dritten Punkt aus S berührt oder x den Rand von A erreicht. Wenn schließlich $\partial C(x)$ zu Anfang nur einen einzigen Punkt aus S enthält, halten wir diesen Kontakt aufrecht und verschieben x unter Vergrößerung von $C(x)$ so lange, bis x eine Ecke von A erreicht oder bis mindestens ein weiterer Punkt von $\partial C(x)$ berührt wird. □

Die drei möglichen Lagen für größte leere Kreise sehen wir in Abbildung 5.13.

Damit wird eine effiziente Bestimmung des „idealen Wohnsitzes“ möglich, wie sie zuerst von Chew und Drysdale [30] beschrieben wurde.

Theorem 5.14 *Den am weitesten von n Störquellen entfernten Punkt in einem konvexen Bereich mit m Ecken kann man in Zeit $O(m + n)$ bestimmen, wenn das Voronoi-Diagramm der Störquellen schon vorhanden ist.*

Beweis. Die Kandidaten für den idealen Wohnsitz werden der Reihe nach inspiziert und der beste, d. h. der mit dem größten Abstand zu den Störquellen in S, ausgewählt.

Wir beginnen mit den $O(n)$ vielen Voronoi-Knoten von $V(S)$; für jeden von ihnen wird der Abstand zu einer der Störquellen bestimmt, deren Regionen sich dort treffen.

Dann müssen die Schnittpunkte des Randes von A mit den Voronoi-Kanten inspiziert werden. Nebenbei stellen wir für jeden Eckpunkt von A die Voronoi-Region $VR(p, S)$ fest, in der er enthalten ist, und bestimmen den Abstand zu p. Wir werden zeigen, daß ein Zeitaufwand in $O(n + m)$ hierfür ausreicht. Bis auf die Klärung dieser Frage, die wir in Lemma 5.15 nachholen werden, ist damit Theorem 5.14 bewiesen. □

Es bleibt also noch ein Teilproblem zu lösen: die Bestimmung der Schnittpunkte, die der Rand eines konvexen Polygons A mit m Ecken mit einem Voronoi-Diagramm von n Punkten hat. Weil jede Kante von $V(S)$ von dem konvexen Rand von A höchstens zweimal geschnitten werden kann, gibt es höchstens $O(n)$ viele Schnittpunkte – aber wie finden wir sie schnell?

Schnittpunkte von konvexem Polygon A mit $V(S)$

Das andere Teilproblem, nämlich die Voronoi-Region der Eckpunkte von A zu bestimmen, läßt sich dabei miterledigen.

Sei $a_1, a_2, \ldots, a_m$ eine Numerierung der Ecken von A gegen den Uhrzeigersinn. Zu Beginn bestimmen wir den ersten Punkt h, den der Strahl von a_1 in Richtung von a_2 im Voronoi-Diagramm $V(S)$ trifft; hierzu testen wir alle $O(n)$ viele Voronoi-Kanten. Damit ist auch die Voronoi-Region bekannt, die den Eckpunkt a_1 enthält.

Zwei Fälle sind möglich, je nachdem ob der Strahl von a_1 zuerst den Punkt h oder zuerst a_2 erreicht; siehe Abbildung 5.14 (i) und (ii).[10]

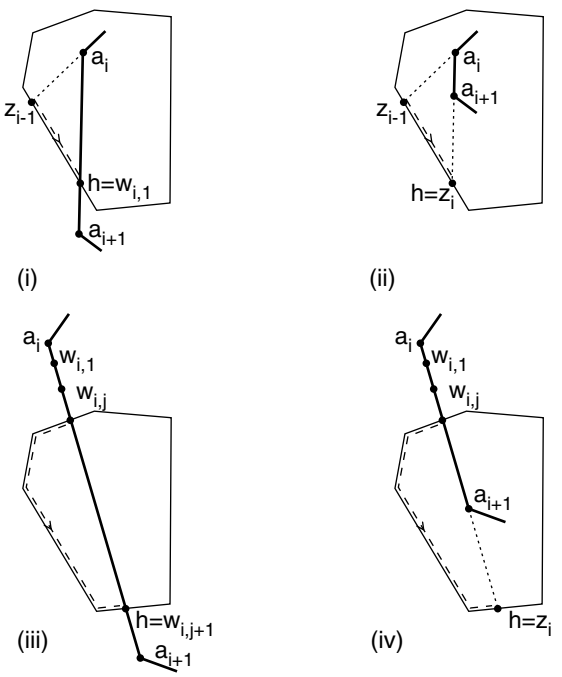


Abb. 5.14 Die vier möglichen Situationen bei der Verfolgung einer konvexen polygonalen Kette durch eine konvexe Region.

In (i) wechselt die aktuelle Kante des Polygons A am Punkt $h = w_{i,1}$ in eine neue Voronoi-Region, und der Punkt $w_{i,1}$ ist

[10] Am Anfang muß man $i = 1$ setzen und den Punkt z_{i-1} ignorieren.

als Schnittpunkt mit $V(S)$ zu berichten. Wie es dann weitergeht, zeigen (iii) und (iv): Man bestimmt den Punkt h auf dem Rand der neuen Region, der von dem Strahl von a_i getroffen wird. Dazu durchläuft man den Rand der Voronoi-Region vom Eintrittspunkt $w_{i,j}$ aus gegen den Uhrzeigersinn und testet jede Kante auf Schnitt mit dem Strahl, bis h gefunden ist. Liegt h von a_i aus vor a_{i+1}, wie in (iii), ist $h = w_{i,j+1}$ ein weiterer Schnittpunkt von $V(S)$ mit der aktuellen Kante von A, die hier wiederum in eine neue Region wechselt. Kommt dagegen a_{i+1} zuerst, wie in (iv), endet die aktuelle Kante von A in der aktuellen Voronoi-Region. Wir müssen dann den Strahl betrachten, der von a_{i+1} in Richtung von a_{i+2} läuft, und seinen Treffpunkt auf dem Rand suchen.

Das bringt uns in eine der in (i) und (ii) gezeigten Situationen. Vom Punkt z_{i-1} aus, in dem die Verlängerung der Vorgängerkante auf den Rand der Voronoi-Region traf, suchen wir gegen den Uhrzeigersinn weiter, bis wir den Treffpunkt h des aktuellen Strahls finden. Er kann vor oder hinter a_{i+1} liegen, und entsprechend fahren wir fort.

Verfolgung des Randes

So wird der Rand des Polygons A vollständig umlaufen; dabei werden auch die Regionen der Eckpunkte bestimmt.

Wieviel Arbeit hat uns die Verfolgung des Randes von A durch das Voronoi-Diagramm gemacht?

Lemma 5.15 *Sei A ein konvexes Polygon mit m Ecken und S eine Menge von n Punkten. Dann kann man die Schnittpunkte von ∂A mit dem Voronoi-Diagramm $V(S)$ und die Voronoi-Regionen der Eckpunkte von A in Zeit $O(m+n)$ bestimmen.*

Beweis. Am Anfang wird $O(n)$ viel Zeit auf die Bestimmung des ersten Trefferpunktes h verwendet. Danach werden nur noch die in Abbildung 5.14 gezeigten vier Schritte ausgeführt. In jedem Schritt ist ein Teil des Randes einer Voronoi-Region abzusuchen; diese Stücke sind gestrichelt eingezeichnet.

Den Suchaufwand in den Schritten (iii) und (iv) können wir durch die Anzahl der durchlaufenen Voronoi-Knoten abschätzen; man beachte, daß die durchlaufenen Randstücke stets mindestens einen Voronoi-Knoten enthalten.

Anders in (i) und (ii): Hier können viele Punkte z_i auf derselben Voronoi-Kante liegen! Doch zu jedem z_i gehört ein eindeutiger Eckpunkt des Polygons A, so daß wir bei diesen Schritten den Suchaufwand insgesamt durch die Anzahl der durchlaufenen Voronoi-Knoten plus $O(m)$ abschätzen können.

Wegen der Konvexität von A wird jedes Randstück einer Region von $V(S)$, also auch jeder darin enthaltene Voronoi-Knoten, insgesamt höchstens einmal durchlaufen. Also liegt der Gesamtaufwand in $O(m+n)$, wie behauptet. □

Dieses Prinzip der Verfolgung eines Polygonrandes durch ein Voronoi-Diagramm und die soeben angewandte Abrechnungsmethode werden uns in Kapitel 6 wiederbegegnen, wenn wir die *divide-and-conquer*-Technik auf die Konstruktion des Voronoi-Diagramms anwenden.

Beim Beweis von Lemma 5.15 haben wir übrigens keinen Gebrauch davon gemacht, daß es sich bei $V(S)$ um ein Voronoi-Diagramm handelt; der Ansatz funktioniert ebensogut für jede Zerlegung der Ebene, solange die Regionen konvex sind.

Neben der in Theorem 5.14 behandelten Aufgabe gibt es zahlreiche andere *Standortprobleme*, zu deren Lösung zum Teil ganz andere Methoden benötigt werden; siehe z. B. Hamacher [71].

Standortprobleme

Wir konnten hier nur wenige Anwendungen des Voronoi-Diagramms aufführen. Es eignet sich z. B. auch dazu, in einer Menge von Objekten die jeweils nah zusammenliegenden in einer Gruppe, einem sogenannten *Cluster*, zusammenzufassen; siehe Dehne und Noltemeier [39].

Cluster

5.4 Die Delaunay-Triangulation

Bereits in Abschnitt 5.3.2 war des öfteren von Paaren von Punkten aus S die Rede, deren Voronoi-Regionen in $V(S)$ an eine gemeinsame Kante angrenzen. Nach unserer Definition des dualen Graphen in Abschnitt 1.2.2 definiert jedes solche Punktepaar eine Kante in $V(S)^*$, dem Dualen des Voronoi-Diagramms[11] von S.

Dieser zu $V(S)$ duale Graph hat selbst einige überraschende und nützliche Eigenschaften, die in den folgenden Abschnitten vorgestellt werden.

5.4.1 Definition und elementare Eigenschaften

Sei also S wieder eine Menge von n Punkten in der Ebene und sei $V(S)$ ihr Voronoi-Diagramm.

Zwei Punkte $p, q \in S$ werden durch das Liniensegment pq miteinander verbunden, wenn ihre Voronoi-Regionen $VR(p, S)$ und $VR(q, S)$ an eine gemeinsame Voronoi-Kante in $V(S)$ angrenzen. Die durch diese Festsetzung entstehende Menge von Liniensegmenten heißt die *Delaunay-Zerlegung* $DT(S)$ der Punktmenge S. Ein Beispiel ist in Abbildung 5.15 gezeigt. Das Liniensegment pq heißt eine *Delaunay-Kante*.

Delaunay-Zerlegung

Delaunay-Kante

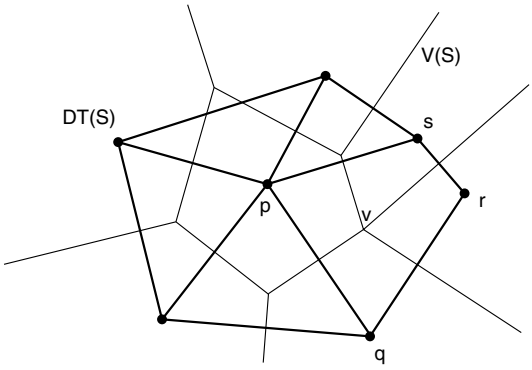


Abb. 5.15 Sieben Punkte in der Ebene, ihr Voronoi-Diagramm und ihre Delaunay-Zerlegung.

Übungsaufgabe 5.7

(i) Man zeige, daß zwei Punkte $p, q \in S$ genau dann durch eine Delaunay-Kante miteinander verbunden sind, wenn es einen Kreis gibt, der nur p und q auf seinem Rand enthält und keinen Punkt aus S in seinem Innern.

(ii) Kann eine Delaunay-Kante pq auch andere Voronoi-Regionen durchqueren als die von p oder q?

Insbesondere ist jede Kante der konvexen Hülle von S eine Delaunay-Kante; auch der gesamte minimale Spannbaum von S ist ein Teilgraph von $DT(S)$, wie wir in Abschnitt 5.3.3 gesehen haben.

Weil zwei Voronoi-Regionen ihrer Konvexität wegen an höchstens *eine* gemeinsame Kante angrenzen können, enthält der duale Graph $V(S)^*$ des Voronoi-Diagramms höchstens eine Kante zwischen je zwei Punkten $p, q \in S$. Außerdem können sich wegen Übungsaufgabe 5.7 (i) zwei Delaunay-Kanten nicht kreuzen. Das bedeutet:

$DT(S) = V(S)^*$

Lemma 5.16 *Die Delaunay-Zerlegung ist eine kreuzungsfreie geometrische Realisierung des dualen Graphen des Voronoi-Diagramms.*

von $V(S)$ nach $DT(S)$ und zurück in $O(n)$ Zeit

Hieraus folgt insbesondere, daß sich die Delaunay-Zerlegung in linearer Zeit aus dem Voronoi-Diagramm berechnen läßt und umgekehrt.

[11] Auch $V(S)$ kann als kreuzungsfreier geometrischer Graph aufgefaßt werden; wir nehmen einfach an, die unbeschränkten Kanten endeten in einem gemeinsamen Knoten ∞.

Auch $DT(S)$ ist eine wichtige Struktur mit interessanten Eigenschaften.

Zunächst einmal folgt aus der Definition des dualen Graphen, daß jede beschränkte Fläche von $DT(S)$ genau so viele Kanten hat, wie bei ihrem zugehörigen Knoten in $V(S)$ zusammenlaufen. Ist die Punktmenge S also so beschaffen, daß keine vier Punkte auf einem Kreisrand liegen, sind die beschränkten Flächen von $DT(S)$ allesamt Dreiecke!

keine vier Punkte auf einem Kreis

Aus diesem Grund heißt $DT(S)$ auch die *Delaunay-Triangulation* der Punktmenge S. Auch wir wollen im folgenden diese weiter verbreitete Bezeichnung verwenden, auch dann, wenn S die genannte Bedingung nicht erfüllt und $DT(S)$ deswegen Flächen mit mehr als drei Kanten enthält. Hierbei kann es sich nur um konvexe Polygone handeln, deren Ecken auf einem Kreis liegen und die sich leicht „nachtriangulieren“ lassen; siehe etwa die Fläche des Voronoi-Knotens v in Abbildung 5.15.

Delaunay-Triangulation

Allgemein versteht man unter einer *Triangulation einer Punktmenge* S eine maximale Menge von Liniensegmenten mit Endpunkten in S, von denen sich je zwei höchstens in ihren Endpunkten schneiden. Die Kanten der konvexen Hülle $ch(S)$ gehören zu jeder Triangulation von S.

Triangulation einer Punktmenge

Das in Abschnitt 4.2 erwähnte Problem, ein einfaches Polygon P mit Eckpunktmenge S zu triangulieren, kann als Spezialfall einer Triangulierung von S angesehen werden, bei der alle Kanten von P verwendet werden müssen und ansonsten nur solche Liniensegmente mit Endpunkten in S verwendet werden dürfen, die Diagonalen von P sind.

Triangulation eines Polygons

Abbildung 5.16 zeigt, daß es für Punktmengen ganz verschiedene Triangulationen gibt. Sie enthalten aber alle dieselbe Anzahl von Dreiecken! Wie die Delaunay-Triangulation beschaffen ist, kann man mit dem Applet

`http://www.geometrylab.de/VoroGlide/`

herausfinden.

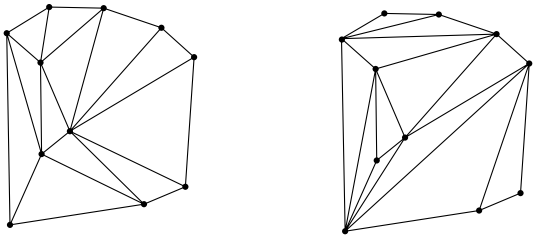


Abb. 5.16 Zwei Triangulationen derselben Punktmenge.

Übungsaufgabe 5.8 Sei S eine Menge von n Punkten in der Ebene. Angenommen, die konvexe Hülle $ch(S)$ hat r Ecken. Man zeige, daß jede Triangulation von S genau $2(n-1)-r$ viele Dreiecke enthält.

5.4.2 Die Maximalität der kleinsten Winkel

einschränkende Annahmen

In diesem Abschnitt sei S eine Menge von n Punkten in der Ebene, von denen keine vier auf einem gemeinsamen Kreisrand und keine drei auf einer Geraden liegen.

Bei der Betrachtung von Abbildung 5.16 fällt auf, daß in der rechten Triangulation zahlreiche spitze Dreiecke mit kleinen Winkeln auftreten, während die linke Triangulation in dieser Hinsicht ausgeglichener wirkt.

Es gibt zahlreiche Anwendungen für Triangulationen, bei denen kleine Winkel in den Dreiecken unerwünscht sind, zum Beispiel aus numerischen Gründen. Hier ist die Delaunay-Triangulation optimal, wie wir jetzt zeigen werden.

Für eine Triangulation T einer Punktmenge S bezeichne

$$w(T) = (\alpha_1, \alpha_2, \ldots, \alpha_{3d})$$

die aufsteigend sortierte Folge der Innenwinkel aller d Dreiecke in T. Ist T' eine andere Triangulation von S, kann man die Winkelfolgen $w(T)$ und $w(T')$ bezüglich der lexikographischen Ordnung miteinander vergleichen; es gilt immer dann

lexikographischer Winkelvergleich

$$w(T) < w(T'),$$

wenn der kleinste Winkel in T kleiner ist als der kleinste Winkel in T', aber auch, wenn für eine Zahl j mit $1 \leq j \leq 3d-1$ die j kleinsten Winkel in beiden Triangulationen gleich sind und erst der $(j+1)$-kleinste Winkel von T kleiner als sein Gegenstück in T' ist.

Theorem 5.17 *Sei S eine Menge von n Punkten in der Ebene, von denen keine vier auf einem gemeinsamen Kreisrand liegen. Dann hat die Delaunay-Triangulation unter allen Triangulationen von S die größte Winkelfolge; hierdurch ist $DT(S)$ eindeutig bestimmt.*

größte Winkelfolge

Hierfür benötigen wir die folgende Charakterisierung von $DT(S)$, deren Beweis analog zur Lösung von Übungsaufgabe 5.7 ist:

Lemma 5.18 *Drei Punkte p, q, r aus S definieren genau dann ein Dreieck von $DT(S)$, wenn der eindeutig bestimmte Kreis durch p, q, r – wir nennen ihn im folgenden den Umkreis des Dreiecks $tria(p, q, r)$ – keinen Punkt aus S im Innern enthält.*

Charakterisierung der Delaunay-Dreiecke

Beweis (von Theorem 5.17). Sei T eine Triangulation von S. Wenn kein Umkreis eines Dreiecks von T einen anderen Punkt aus S im Innern enthält, kommen wegen Lemma 5.18 alle Dreiecke von T auch in der Delaunay-Triangulation vor. Dann folgt sofort $T = DT(S)$, weil ja nach Übungsaufgabe 5.8 alle Triangulationen von S dieselbe Anzahl von Dreiecken enthalten.

Andernfalls gibt es mindestens ein Dreieck D in T, dessen Umkreis einen Punkt t aus S im Innern enthält. Solch ein Punkt kann natürlich nicht im Dreieck selbst liegen, sonst wäre T keine Triangulation; er liegt also zwischen dem Kreisrand und einer der Dreieckskanten.

Wir wählen unter allen solchen Paaren (D, t) eines aus, bei dem der Winkel α, unter dem t die Endpunkte „seiner“ Dreieckskante sieht, maximal ist. In Abbildung 5.17 ist $D = tria(p, q, r)$, und pr ist die dem Punkt t zugewandte Kante von D.

Wahl eines maximalen Kandidaten

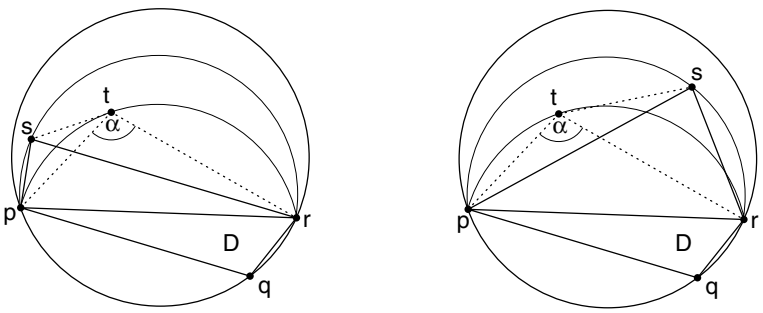


Abb. 5.17 Der Winkel zwischen ts und tr bzw. zwischen tp und ts ist größer als α.

Wir behaupten, daß auch $tria(p, r, t)$ zu T gehört! Zunächst einmal ist wegen der Lage von t und q klar, daß pr keine Kante der konvexen Hülle von S ist. Sollte also der Punkt t mit pr kein Dreieck von T bilden, müßte ein anderer Punkt s das tun, der auf derselben Seite von pr liegt wie t.

Der Punkt s kann nur außerhalb des Kreises durch p, r und t liegen, weil er sonst auch im Umkreis von D läge und die Punkte p, r in einem größeren Winkel als α sähe.[12] Also liegt t im Umkreis, aber nicht im Inneren, von $tria(p, r, s)$. Dann sieht t aber

[12] Das folgt aus dem Satz des Thales auf Seite 24.

die Endpunkte der ihm zugewandten Dreieckskante – entweder sr oder ps – in einem Winkel $> \alpha$, was ebenfalls der Maximalität von α widerspricht.

Damit haben wir gezeigt, daß doch $tria(p, r, t)$ das zu D benachbarte Dreieck in der Triangulation T ist.

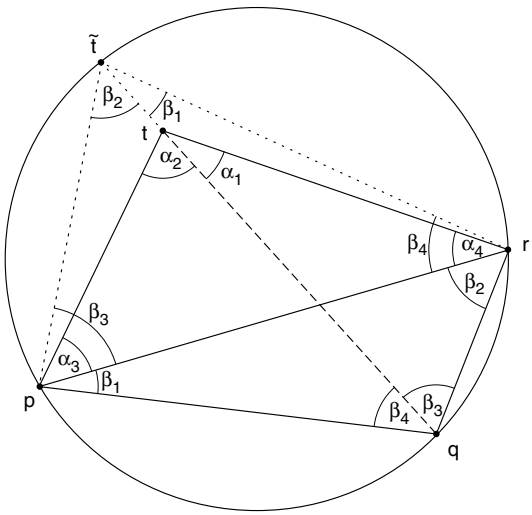


Abb. 5.18 Über der Sehne qr sind die Winkel β_1 in den Punkten $\widetilde{t}$ und p gleich; das gilt entsprechend auch für die anderen Winkel $\beta_2, \beta_3, \beta_4$.

Nun behaupten wir: Wenn man in T die Kante pr durch qt ersetzt,[13] entsteht eine Triangulation T' von S mit

$$w(T) < w(T').$$

Damit wäre dann alles bewiesen: Für jede von $DT(S)$ verschiedene Triangulation T läßt sich $w(T)$ noch vergrößern. Weil es nur endlich viele Triangulationen von S gibt, muß der Vergrößerungsprozeß mit $DT(S)$ enden.

Es bleibt also noch der Beweis der obigen Behauptung. Daß T' wieder eine Triangulation von S ist, liegt auf der Hand. Betrachten wir also die in Abbildung 5.18 eingezeichneten Winkel.

Wenn man das Liniensegment qt bis zu einem Punkt $\widetilde{t}$ auf dem Rand des Umkreises von $tria(p, q, r)$ fortsetzt, kann man viermal den Satz des Thales anwenden, und zwar auf $\beta_1, \beta_2, \beta_3, \beta_4$; vergleiche Abbildung 1.14 auf Seite 25. Danach sind zum Beispiel in Abbildung 5.18 in den Dreiecken $tria(p, q, \widetilde{t})$ und $tria(p, q, r)$ die Winkel β_2 an den Punkten $\widetilde{t}$ und r identisch.

[13] Im Englischen nennt man diesen Vorgang einen *edge flip*.

Außerdem gilt

$$\alpha_1 > \beta_1 \quad \text{und} \quad \alpha_2 > \beta_2$$
$$\alpha_3 < \beta_3 \quad \text{und} \quad \alpha_4 < \beta_4.$$

Mit der Originalkante pr entstehen in $tria(p, q, r)$ und $tria(r, t, p)$ in unsortierter Reihenfolge die Innenwinkel

$$\beta_1, \beta_2, \alpha_3, \alpha_4, \alpha_1 + \alpha_2, \beta_3 + \beta_4.$$

Ersetzt man pr durch die Kante qt, haben $tria(p, q, t)$ und $tria(q, r, t)$ die Innenwinkel

$$\alpha_1, \alpha_2, \beta_3, \beta_4, \beta_1 + \alpha_3, \beta_2 + \alpha_4.$$

Jeder von ihnen ist echt größer als ein Winkel aus der oberen Liste; zum Beispiel ist $\beta_3 > \alpha_3$ und $\beta_2 + \alpha_4 > \beta_2$; alle übrigen Winkel bleiben beim Übergang von T zur Triangulation T' unverändert. Insbesondere wird also der kleinste Winkel der oberen Liste durch einen größeren Winkel ersetzt. Daraus folgt $w(T) < w(T')$. □

Wegen der in Theorem 5.17 beschriebenen Eigenschaft der Maximalität der kleinsten Winkel wird die Delaunay-Triangulation zum Beispiel bei der Approximation von räumlichen Flächen durch Dreiecksnetzwerke verwendet; siehe zum Beispiel Abramowski und Müller [2].

5.5 Verallgemeinerungen

Bisher haben wir nur das Voronoi-Diagramm von Punkten im $\mathbb{R}^2$ unter der euklidischen Metrik betrachtet. In Anbetracht seiner zahlreichen nützlichen Anwendungen ist es nicht erstaunlich, daß dieser Ansatz auf verschiedene Weisen verallgemeinert wurde. Zwei solche Verallgemeinerungen werden im folgenden vorgestellt.

5.5.1 Allgemeinere Abstandsbegriffe

In Abschnitt 5.3.1 hatten wir auf Seite 219 bei der Bestimmung des nächstgelegenen Postamts angenommen, daß die Entfernung vom Anfrageort x zu einem Postamt p_i durch die Luftlinienentfernung, d. h. durch den euklidischen Abstand $|p_i x|$, adäquat beschrieben wird.

Luftlinienabstand nicht immer geeignet

In einer realen Umgebung ist diese Annahme nicht immer erfüllt. Die Einwohner von Manhattan können sich zum Beispiel nur in einem orthogonalen Straßengitter bewegen; siehe Abbildung 5.19. Für sie beträgt die Entfernung zwischen zwei Punkten $p = (p_1, p_2)$ und $q = (q_1, q_2)$ daher

$$L_1(p, q) = |p_1 - q_1| + |p_2 - q_2|.$$

Manhattan-Metrik

Man nennt L_1 deshalb auch die *Manhattan-Metrik*. Diese Metrik ist ein Mitglied der Familie der *Minkowski-Metriken*

$$L_i(p, q) = \sqrt[i]{|p_1 - q_1|^i + |p_2 - q_2|^i}, \quad 1 \leq i < \infty,$$

sowie

$$L_\infty(p, q) = \max(|p_1 - q_1|, |p_2 - q_2|).$$

Offenbar ist L_2 die euklidische Metrik. Diese Definitionen lassen sich für beliebiges $d > 2$ auf den $\mathbb{R}^d$ verallgemeinern.

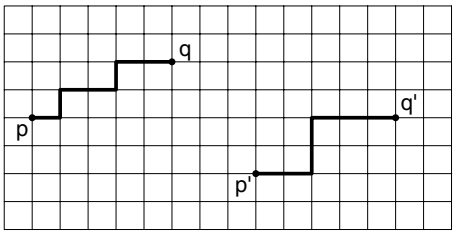


Abb. 5.19 Entfernungen in der Manhattan-Metrik.

Die Funktionen L_i gehören zu einer Menge spezieller Metriken, die in der Mathematik *Normen* heißen und in der Algorithmischen Geometrie *symmetrische konvexe Distanzfunktionen* genannt werden.

Sie lassen sich rein geometrisch folgendermaßen beschreiben. Sei C eine kompakte, konvexe Menge in der Ebene, die den Nullpunkt im Innern enthält; dieser Punkt wird im folgenden auch als das *Zentrum* von C bezeichnet. Um den Abstand von p nach $q \neq p$ bezüglich C zu definieren, wird C zunächst um den Vektor p verschoben. Der Strahl von p durch q schneidet den Rand von C in genau einem Punkt q'; siehe Abbildung 5.20. Nun definiert man

$$d_C(p, q) = \frac{|pq|}{|pq'|};$$

diese Zahl gibt den Faktor an, um den die nach p verschobene Kopie von C skaliert werden müßte, damit ihr Rand den Punkt q enthält. Wir setzen außerdem noch $d_C(p, p) = 0$.

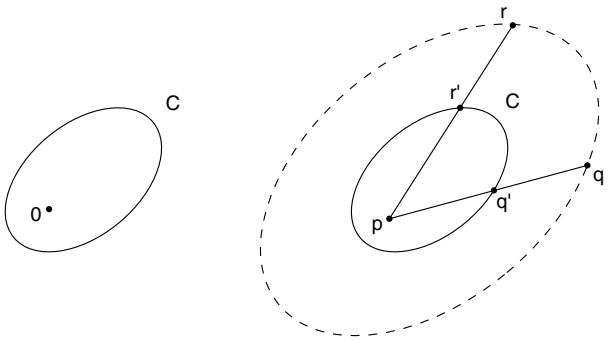


Abb. 5.20 Zur Definition einer konvexen Distanzfunktion; für jeden Punkt r auf dem Rand der skalierten Kopie von C gilt $\frac{|pr|}{|pr'|} = \frac{|pq|}{|pq'|}$.

konvexe Distanzfunktion

Die Abbildung d_C heißt eine *konvexe Distanzfunktion*; sie erfüllt $d_C(p,q) \geq 0$, und $d_C(p,q) = 0$ gilt genau dann, wenn $p = q$ ist. Man kann zeigen, daß wegen der Konvexität von C auch die Dreiecksungleichung[14]

$$d_C(p,r) \leq d_C(p,q) + d_C(q,r)$$

gilt. Die Symmetriebedingung $d_C(p,q) = d_C(q,p)$ für Metriken ist aber nur erfüllt, wenn C in bezug auf den Nullpunkt *symmetrisch* ist.

Übungsaufgabe 5.9 Sei $C \subset \mathbb{R}^2$ kompakt und konvex mit dem Nullpunkt im Innern, und sei D die aus C durch Spiegelung am Nullpunkt hervorgehende Menge. Man zeige, daß für alle Punkte p, q die Beziehung $d_C(p,q) = d_D(q,p)$ gilt.

Nach Definition ist C gerade der *Einheitskreis*

$$\{x \in \mathbb{R}^2;\ d_C(x,0) \leq 1\}$$

von d_C. Außerdem ist d_C invariant unter Translationen, d. h. es ist $d_C(p,q) = d_C(0, q-p)$.

Übungsaufgabe 5.10

(i) Man bestimme die Einheitskreise von L_1 und L_∞.

(ii) Sei R der Einheitskreis von L_1. Man zeige, daß $L_1 = d_R$ ist.

Offenbar kann man zu jedem noch so kleinen euklidischen Kreis K um den Nullpunkt einen Stauchungsfaktor t finden, so daß die

[14] Siehe z. B. Barner und Flohr [13].

gestauchte Menge tC ganz in K enthalten ist und umgekehrt. Das bedeutet: Die euklidische Metrik und die konvexe Distanzfunktion d_C induzieren dieselbe Topologie im $\mathbb{R}^2$, vgl. Kapitel 1 auf Seite 9.

Kann man nun Voronoi-Diagramme auch bezüglich einer konvexen Distanzfunktion d_C definieren? Ja, mit ein wenig Vorsicht. Für zwei Punkte p, q können wir die Mengen $D_C(p,q)$, $D_C(q,p)$ und $B_C(p,q)$ genauso definieren wie auf Seite 211, indem wir statt der euklidischen Metrik überall d_C einsetzen.

Bisektoren unter d_C

Schauen wir uns zunächst die Bisektoren unter der konvexen Distanzfunktion d_C an; aus ihnen sind die Voronoi-Diagramme ja aufgebaut. Beides kann man sich mit den Applets unter

`http://www.geometrylab.de/ConvexDistFkt/`

anschauen.

Jeder Weg von einem Punkt p zu einem anderen Punkt q muß unterwegs den d_C-Bisektor $B_C(p,q)$ kreuzen. Denn wenn der Punkt x den Weg entlangwandert, verändert sich der Abstand $d_C(p,x)$ stetig von 0 zu $d_C(p,q)$, während $d_C(x,q)$ sich von $d_C(p,q)$ zu 0 verändert; nach dem Zwischenwertsatz muß irgendwo $d_C(p,x) = d_C(x,q)$ sein. Also zerlegt auch hier $B_C(p,q)$ die Ebene in die beiden Mengen $D_C(p,q)$ und $D_C(q,p)$, die offen sind und p bzw. q enthalten. Insbesondere ist $B_C(p,q)$ unbeschränkt.

Zerlegung der Ebene

Um den Bisektor $B_C(p,q)$ zu konstruieren, stellt man sich zwei an p und q zentrierte Kopien des Einheitskreises C vor, die mit gleicher Geschwindigkeit wachsen. Alle Punkte im Durchschnitt ihrer Ränder sind von p und q gleich weit entfernt, gehören also zu $B_C(p,q)$.

Manhattan-Metrik

In Abbildung 5.21 sind der Einheitskreis und mögliche Bisektoren der Manhattan-Metrik L_1 dargestellt. Figur (ii) zeigt den typischen Fall, der immer dann eintritt, wenn p und q ein echtes Rechteck aufspannen, das kein Liniensegment und kein Quadrat ist. Der Bisektor besteht aus zwei Halbgeraden und einem Liniensegment mit Winkel $\pi/4$ zu den Koordinatenachsen. Wenn man Figur (ii) um eine Vierteldrehung dreht, ergibt sich ein anderer möglicher Fall.

Haben p und q identische X- oder Y-Koordinaten, so stimmt ihr L_1-Bisektor mit dem euklidischen Bisektor überein. Ein gravierender Unterschied tritt aber bei der L_1-Metrik auf, wenn p und q diagonale Ecken eines Quadrats sind, wie in Abbildung 5.21 (iii) dargestellt. In diesem Fall enthält ihr Bisektor zwei volle Viertelebenen!

flächiger Bisektor

Man kann dieser Besonderheit dadurch begegnen, daß man so tut, als sei das Quadrat ein liegendes Rechteck; die senkrechten Ränder der Viertelebenen und das diagonale Segment werden zur Bisektorkurve $B_C^*(p,q)$ erklärt, die Reste der beiden Viertelebenen

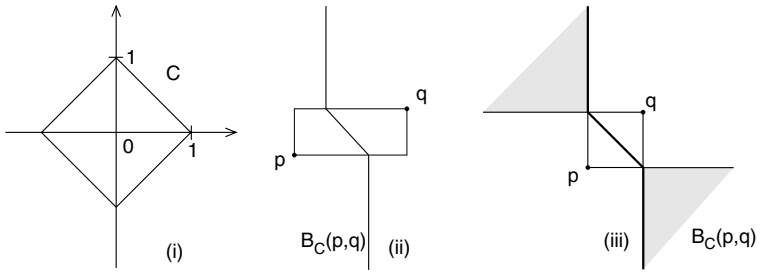


Abb. 5.21 Einheitskreis und Bisektoren unter L_1.

werden den Regionen von p und q zugeschlagen.[15] So entstehen offene Gebiete $D_C^*(p,q)$ und $D_C^*(q,p)$, die $D_C(p,q)$ und $D_C(q,p)$ echt umfassen und von $B_C^*(p,q)$ separiert werden.

Wenn p und q nicht gerade die diagonalen Ecken eines Quadrats sind, setzt man $D_C^*(p,q) = D_C(p,q)$ und $B_C^*(p,q) = B_C(p,q)$.

Als Voronoi-Region von p kann man dann

$$VR_C^*(p,S) = \bigcap_{q \in S \setminus \{p\}} D_C^*(p,q)$$

definieren; zur Definition des Voronoi-Diagramms eignen sich die in Übungsaufgabe 5.2 angegebenen Darstellungen.

Im Umgang mit diesen Diagrammen ist aber Vorsicht geboten: Zum Beispiel ist die dritte Aussage in Lemma 5.1 jetzt nicht mehr richtig, wie die folgende Übungsaufgabe zeigt!

Übungsaufgabe 5.11 Man gebe drei Punkte p, q, r auf dem Rand eines L_1-Kreises $C(x)$ an, dessen Zentrum x im L_1-Voronoi-Diagramm von p, q, r *keinen* Voronoi-Knoten bildet.

Das Phänomen flächiger Bisektorstücke in $B_C(p,q)$ tritt aber nur auf, wenn der Einheitskreis C stellenweise abgeplattet ist, d. h. Liniensegmente in seinem Rand enthält, und wenn außerdem p und q auf einer Geraden liegen, die parallel zu solch einem Randsegment von C ist.

Man nennt C – und dann auch d_C – *streng konvex*, wenn ∂C keine Liniensegmente enthält. Für streng konvexe Distanzfunktionen können die Bisektoren keine Flächenstücke enthalten, wie folgendes Lemma zeigt.

streng konvex

[15] Eine andere Methode, die für beliebige Metriken funktioniert, wird in Klein [86] vorgeschlagen.

Lemma 5.19 *Sei C eine kompakte, streng konvexe Menge in der Ebene mit dem Nullpunkt im Innern. Dann ist jeder Bisektor $B_C(p,q)$ bezüglich der konvexen Distanzfunktion d_C homöomorph*[16] *zu einer Geraden.*

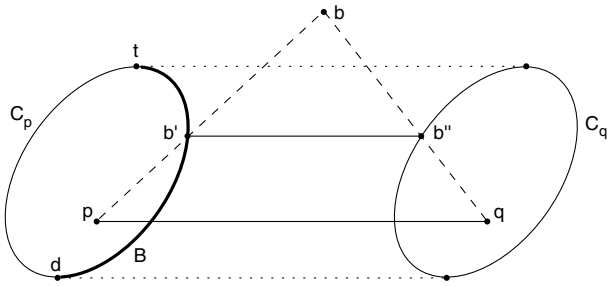


Abb. 5.22 Ist der Einheitskreis C streng konvex, so ist der Bisektor von zwei Punkten immer ein Weg.

Beweis. Wir betrachten zwei in p und q zentrierte Kopien C_p, C_q von C, wie in Abbildung 5.22 dargestellt. Sollten sie sich schneiden, werden sie beide so lange um denselben Faktor zentrisch gestaucht, bis sie disjunkt sind. Sei b ein Punkt der Ebene außerhalb von C_p und C_q; mit b' und b'' bezeichnen wir die Durchschnitte von pb und qb mit den Rändern von C_p und C_q. Es gilt:

$$\begin{aligned} b \in B_C(p,q) &\iff d_C(p,b) = d_C(q,b) \\ &\iff \frac{|bp|}{|b'p|} = \frac{|bq|}{|b''q|} \qquad (*) \\ &\iff b'b'' \text{ ist parallel zu } pq. \end{aligned}$$

Tatsächlich müssen die beiden Punkte b', b'' auf den einander zugewandten Randstücken von C_p und C_q liegen, wie in Abbildung 5.22 dargestellt. Läge einer von ihnen auf der Rückseite, wären die Strahlen von p durch b' und von q durch b'' parallel und könnten sich daher nicht in b schneiden; lägen beide auf der Rückseite, liefen die Strahlen sogar auseinander.

Seien umgekehrt b', b'' zwei Punkte auf den einander zugewandten Seiten von C_p und C_q, für die $b'b''$ parallel zu pq ist. Dann schneiden sich die Strahlen von p durch b' und von q durch b'' in einem Punkt b, der $(*)$ erfüllt und deshalb zu $B_C(p,q)$ gehört.

Seien $t, d \in \partial C_p$ die Berührpunkte der oberen und unteren gemeinsamen Tangenten von C_p und C_q, und sei B der C_q zugewandte Teil von ∂C_p ohne t und d. Man kann zeigen, daß B – wie

[16] D. h., es gibt eine bijektive Abbildung von $B_C(p,q)$ auf die Y-Achse, die in beiden Richtungen stetig ist.

jedes offene Intervall (d, t) in $\mathbb{R}^1$ – homöomorph zu einer Geraden ist.

Die oben eingeführte Abbildung

$$\begin{array}{rcl} B_C(p,q) & \longrightarrow & B \\ b & \longmapsto & b' \end{array}$$

ist, wie wir gesehen haben, surjektiv: Zu gegebenem $b' \in B$ läßt sich ja immer ein b'' im zugewandten Teil von ∂C_q finden, so daß $b'b''$ parallel mit pq ist.

Die Abbildung $b \longmapsto b'$ ist aber auch injektiv! Gäbe es nämlich zwei Punkte $b_1, b_2 \in B_C(p,q)$ mit $b_1' = b_2'$, müßten die Punkte b_1'' und b_2'' auf dem Rand von C_q verschieden sein. Weil $b_1'b_1''$ und $b_1'b_2''$ parallel zu pq sind, müßten b_1', b_1'', und b_2'' auf einer Geraden liegen. Weil b_1'' und b_2'' auf dem C_p *zugewandten* Teil des Randes von C_q liegen, zwischen den Berührpunkten der oberen und unteren Tangente, muß der Rand von C_q zwischen b_1'' und b_2'' gerade verlaufen, was der strengen Konvexität widerspricht; siehe Abbildung 5.23.

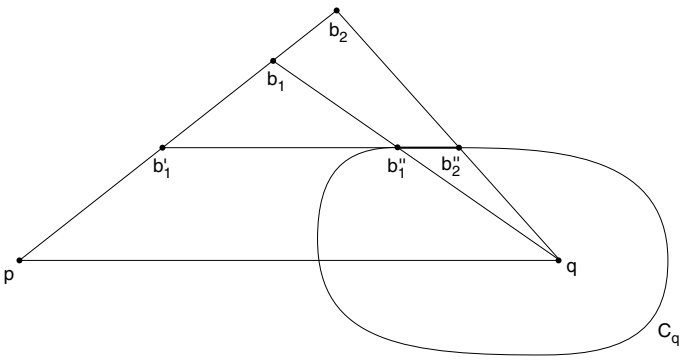


Abb. 5.23 Wenn b_1'', $b_2'' \in \partial C_q$ auf der C_p zugewandten Seite und mit b_1' auf einer zu pq parallelen Geraden liegen, enthält ∂C_q das Liniensegment $b_1''b_2''$.

Man überlegt sich leicht, daß die Zuordnung $b \longmapsto b'$ in beide Richtungen stetig ist. Damit haben wir eine Homöomorphie zwischen dem Bisektor $B_C(p,q)$ und dem Randstück B. □

ab jetzt: d_C streng konvex

Ab jetzt nehmen wir an, daß d_C streng konvex ist, und brauchen dann nach Lemma 5.19 zwischen $VR_C^*(p,S)$ und $VR_C(p,S)$ nicht zu unterscheiden.

Voronoi-Regionen nicht mehr konvex

Der Hauptunterschied zur euklidischen Metrik besteht bei konvexen Distanzfunktionen darin, daß die Voronoi-Regionen im allgemeinen nicht konvex sind; schließlich sind die Bisektoren ja im allgemeinen keine Geraden.

Für die verlorene Konvexität gibt es aber einen Ersatz!

Lemma 5.20 *Sei d_C eine konvexe Distanzfunktion. Dann ist jede Voronoi-Region $VR_C(p, S)$ sternförmig und enthält p im Kern.*[17]

Beweis. Sei $x \in VR_C(p, S)$; angenommen, es gibt einen Punkt $y \in px$, der nicht zur d_C-Voronoi-Region von p gehört. Dann liegt y entweder in der Region eines anderen Punktes $q \in S$ oder auf dem Voronoi-Diagramm selbst, also auf einem Bisektor $B_C(p, q)$. In beiden Fällen folgt $d_C(p, y) \geq d_C(q, y)$, also

$$\begin{aligned} d_C(p, x) &= d_C(p, y) + d_C(y, x) \\ &\geq d_C(q, y) + d_C(y, x) \geq d_C(q, x), \end{aligned}$$

ein Widerspruch zur Annahme $x \in VR_C(p, S)$! □

Bei diesem Beweis haben wir ausgenutzt, daß für drei konsekutive Punkte a, b, c auf einer Geraden die Dreiecksungleichung zur Gleichheit wird, also

$$d_C(a, b) + d_C(b, c) = d_C(a, c)$$

gilt; dies folgt unmittelbar aus der Definition der konvexen Distanzfunktionen und zeigt die enge Verwandtschaft zur euklidischen Metrik auf.

Aus Lemma 5.20 folgt insbesondere:

Voronoi-Regionen zusammenhängend

Korollar 5.21 *Jede Voronoi-Region bezüglich einer konvexen Distanzfunktion ist zusammenhängend.*

Streng konvexe, kompakte Mengen C, wie sie hier als Einheitskreise von konvexen Distanzfunktionen auftreten, haben eine wichtige Eigenschaft mit normalen euklidischen Kreisen gemein.

nur zwei Schnittpunkte zwischen zwei Kreisen

Lemma 5.22 *Sei C kompakt und streng konvex, und sei $C' \neq C$ eine skalierte und verschobene Kopie von C. Dann schneiden sich die Ränder ∂C und $\partial C'$ in höchstens zwei Punkten.*

Beweis. Sei c der Schnittpunkt der beiden äußeren Tangenten an C und C', wobei wir annehmen, daß C kleiner als C' ist; siehe Abbildung 5.24 (sollten C und C' gleich groß sein, so kann der folgende Beweis ähnlich geführt werden). Seien t, d und t', d' jeweils die oberen und unteren Tangentenpunkte auf ∂C und $\partial C'$.

Durch diese Tangentenpunkte zerfallen die Ränder von C und C' in je zwei Stücke A, B und A', B'; hierbei seien die Bezeichnungen wie in Abbildung 5.24. Das Randstück A' entsteht dadurch, daß man für jeden Punkt $a \in A$ den Vektor von c nach a um den

[17] Zum Begriff *Kern* siehe Seite 195.

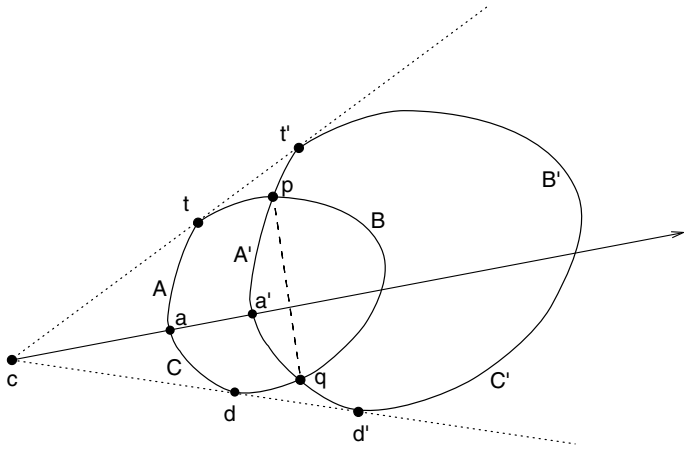


Abb. 5.24 Zwei skalierte Kopien einer streng konvexen, kompakten Menge können höchstens zwei gemeinsame Randpunkte besitzen.

Faktor $\frac{|t'c|}{|tc|} = \frac{|d'c|}{|dc|}$ verlängert; folglich können sich A und A' nicht schneiden. Dasselbe gilt für B und B'. Erst recht ist $A \cap B' = \emptyset$. Alle Schnittpunkte von ∂C mit $\partial C'$ müssen also in $A' \cap B$ liegen. Seien p und q die – von c aus gesehen – am weitesten links und rechts gelegenen Schnittpunkte von A' und B. Wegen der strengen Konvexität verläuft das Liniensegment pq bis auf seine Endpunkte ganz im Innern von C und C'; folglich trennt es die Randstücke A' und B voneinander, so daß sie zwischen p und q keinen dritten Schnittpunkt haben können. □

Jetzt können wir eine Eigenschaft von konvexen Distanzfunktionen beweisen, die wir im euklidischen Fall bereits kennen:

Korollar 5.23 *Sei d_C eine streng konvexe Distanzfunktion. Dann können zwei Voronoi-Regionen $VR_C(p, S)$ und $VR_C(q, S)$ an höchstens eine gemeinsame Voronoi-Kante angrenzen.*

Beweis. Angenommen, es gibt zwei solche Kanten, wie in Abbildung 5.25 (i) gezeigt. Weil die Regionen von p und q nach Korollar 5.21 zusammenhängend sind, müssen ihre Ränder zwischen den Voronoi-Knoten v und w eine oder mehrere andere Regionen einschließen. Eine von ihnen gehöre zum Punkt r.

Wenn wir von S zur Teilmenge $\{p, q, r\}$ übergehen, werden die Voronoi-Regionen dieser drei Punkte höchstens größer. Also sieht $V_C(\{p, q, r\})$ so aus, wie in Abbildung 5.25 (ii) dargestellt. Hier schneiden sich die Bisektoren $B_C(p, r)$ und $B_C(r, q)$ in *zwei*

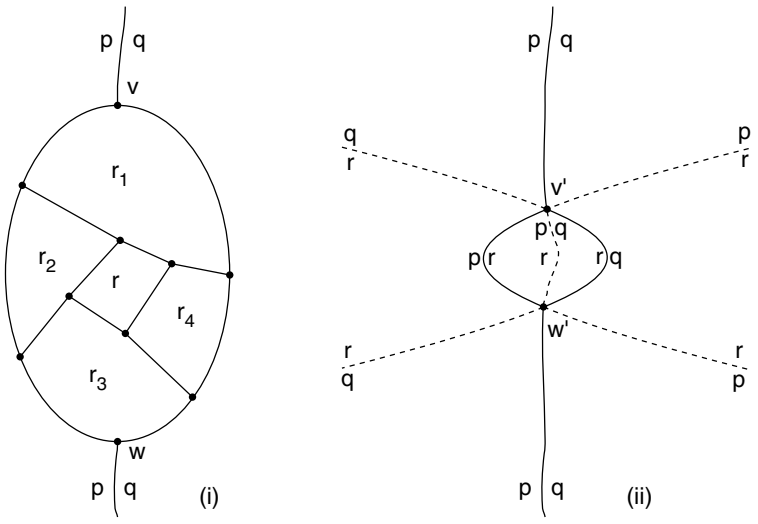


Abb. 5.25 Zwei Bisektoren $B_C(p,r)$ und $B_C(r,q)$ einer streng konvexen Distanzfunktion können sich nicht in zwei Punkten schneiden!

Punkten v' und w'. Es gilt daher

$$d_C(p,v') = d_C(q,v') = d_C(r,v'),$$

woraus nach Übungsaufgabe 5.9 für das Spiegelbild D von C

$$d_D(v',p) = d_D(v',q) = d_D(v',r)$$

folgt. Also gibt es eine skalierte Kopie von D mit Mittelpunkt v', deren Rand durch p,q und r läuft. Ebenso gibt es eine Kopie mit Mittelpunkt w', die p,q,r in ihrem Rand enthält. Nach Lemma 5.22 können sich aber zwei Kopien von D, auch wenn sie unterschiedlich skaliert sind, höchstens in zwei Punkten, nicht aber in p,q,r schneiden – ein Widerspruch! □

Zusammenfassung

Damit haben wir die Struktur von Voronoi-Diagrammen bezüglich streng konvexer Distanzfunktionen in der Ebene geklärt: Sie haben im Grunde dieselben Eigenschaften wie das euklidische Voronoi-Diagramm, nur sind die Bisektoren gekrümmt.[18]

Diese Ähnlichkeit endet abrupt in höheren Dimensionen! In der Ebene gibt es zu drei Punkten höchstens einen Kreis, der die Punkte auf seinem Rand enthält; diese Aussage gilt nach Lemma 5.22 für skalierte Kopien eines streng konvexen Einheitskreises

[18] Trotzdem sind für eine konkrete Punktmenge S die Voronoi-Diagramme $V(S)$ und $V_C(S)$ in der Regel als Graphen verschieden!

ebenso, wie sie für normale euklidische Kreise gilt. Im $\mathbb{R}^3$ gibt es analog zu vier Punkten, die nicht gerade alle auf einem Kreisrand liegen, höchstens eine euklidische Kugel, deren Oberfläche die Punkte enthält. Diese Aussage ist für skalierte Kopien von beliebigen kompakten, streng konvexen Mengen $C \subset \mathbb{R}^3$ aber falsch; siehe Icking et al. [80]!

große Unterschiede im $\mathbb{R}^3$

Nicht immer sind konvexe Distanzfunktionen allgemein genug, um die reale Welt adäquat zu beschreiben. Wer etwa in der Stadt Karlsruhe lebt, findet ein Straßennetz vor, das im wesentlichen aus geraden Stücken besteht, die wie Strahlen zum zentral gelegenen Schloß führen, und aus (Teilen von) Kreisbögen um das Schloß herum. Die Entfernung zwischen zwei Punkten p und q ist hier *nicht* invariant unter einer simultanen Verschiebung von p und q und kann deshalb nicht durch eine konvexe Distanzfunktion dargestellt werden; siehe Abbildung 5.26.

Karlsruhe-Metrik

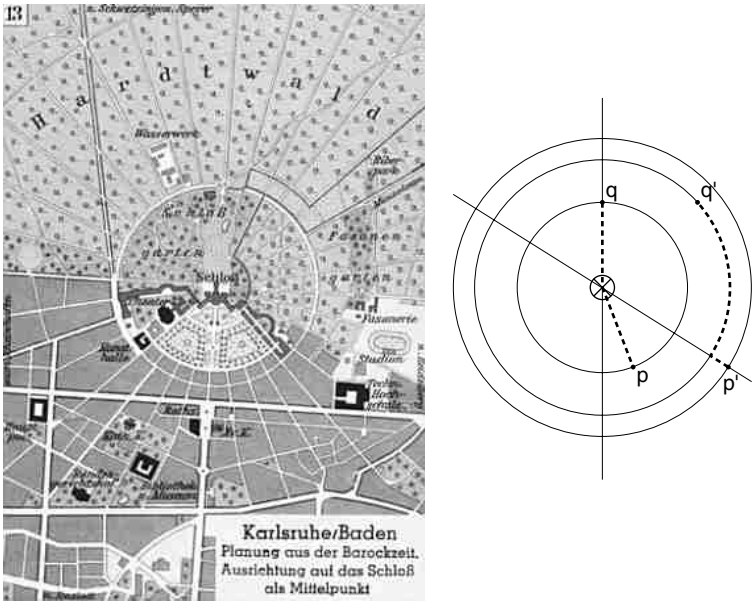


Abb. 5.26 Links ist der Stadtplan von Karlsruhe [46] zu sehen;[20] rechts sind die verschobenen Punkte p' und q' weiter voneinander entfernt als p von q.

Zur Modellierung solcher Umgebungen lassen sich Metriken in der Ebene verwenden. Auf beliebigen Metriken beruhende

Voronoi-Diagramme können aber recht merkwürdig aussehen: Ihre Regionen brauchen zum Beispiel nicht zusammenhängend zu sein.

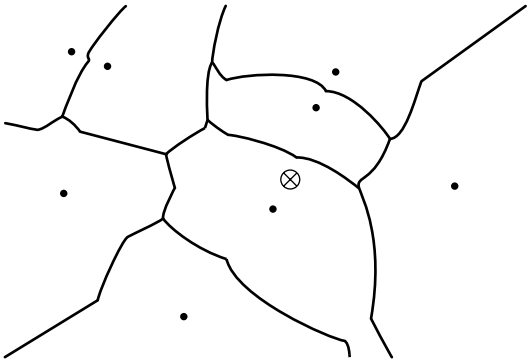


Abb. 5.27 Das Voronoi-Diagramm von acht Punkten in der Karlsruhe-Metrik.

hübsche Metrik

Es gibt eine große Menge von Metriken, bei denen man mehr Glück hat, das sind die sogenannten *hübschen Metriken*. Hierzu zählen alle konvexen Distanzfunktionen, aber auch die oben eingeführte Karlsruhe-Metrik. Abbildung 5.27 zeigt ein Beispiel für ein Voronoi-Diagramm von acht Punkten.

Wer sich speziell für die Karlsruhe-Metrik oder überhaupt für Metriken und ihre Voronoi-Diagramme interessiert, sei auf Klein [86] verwiesen.

5.5.2 Voronoi-Diagramme von Liniensegmenten

Bisher haben wir nur Voronoi-Diagramme von Punkten betrachtet; die Definition läßt sich aber auf andere geometrische Objekte ausdehnen. In diesem Abschnitt betrachten wir das Voronoi-Diagramm einer Menge S von n Liniensegmenten unter der euklidischen Metrik. Dazu müssen wir zunächst unseren Abstandsbegriff auf Punkte x und Liniensegmente l fortsetzen. Wir definieren

Abstand Punkt-Liniensegment

$$|xl| = \min_{y \in l} |xy|.$$

Das Minimum wird in dem zu x nächsten Punkt $y_x \in l$ angenommen; Abbildung 5.28 zeigt die beiden Möglichkeiten.

Der Abstand zwischen einem Punkt und einer Geraden wird genauso definiert; offenbar kann hierbei nur Fall (i) aus Abbildung 5.28 auftreten.

[20] Für die Genehmigung zum Abdruck des Bildes danken wir dem Westermann Schulbuch Verlag GmbH.

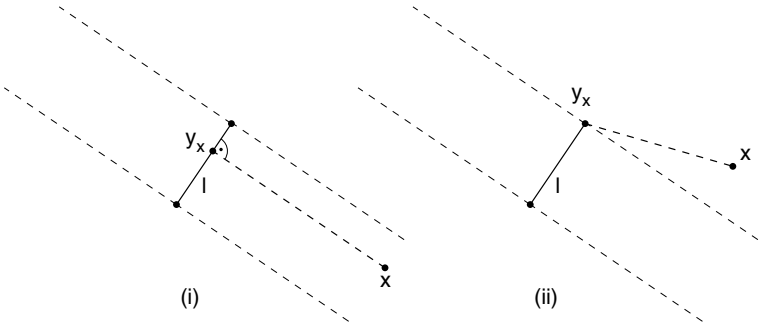


Abb. 5.28 Der zu x dichteste Punkt y_x auf einem Liniensegment l.

Zuerst bestimmen wir die Gestalt der Bisektoren

$$B(l_1, l_2) = \{x \in \mathbb{R}^2;\ |xl_1| = |xl_2|\}$$

von zwei Liniensegmenten. Dazu dient folgende Vorübung:

Bisektor von Punkt und Gerade

Übungsaufgabe 5.12 Sei g eine Gerade und p ein Punkt. Wie sieht der Bisektor von g und p aus?

Unser nächster Schritt besteht in der Konstruktion des Bisektors $B(l, p)$ für ein Liniensegment $l = ab$ und einen Punkt $p \notin l$. Dazu betrachten wir den zur Geraden g durch ab senkrechten Streifen, dessen Ränder durch a und b laufen; siehe Abbildung 5.29. Auf jeder Seite des Streifens liegt eine Halbebene.

Bisektor von Punkt und Liniensegment

Liegt p auf der Geraden g, ist $B(l, p)$ der Bisektor von p und dem zu p benachbarten Endpunkt von l. Andernfalls schneiden $B(a, p)$ und $B(b, p)$ jeweils den Streifenrand durch a bzw. durch b. Für die Schnittpunkte v und w gilt

$$\begin{aligned} &|vp| = |va| = |vl| \\ \text{und}\quad &|wp| = |wb| = |wl|; \end{aligned}$$

folglich liegen v, w auf $B(l, p)$. Zwischen ihnen erstreckt sich ein Teil der Parabel $B(g, p)$ (siehe Übungsaufgabe 5.12), der in den Halbebenen durch Halbgeraden aus $B(a, p)$ und $B(b, p)$ fortgesetzt wird.

Nun können wir schon ein wenig mehr über die Bisektoren von Liniensegmenten sagen.

Lemma 5.24 *Der Bisektor von zwei disjunkten Liniensegmenten ist eine Kurve, die aus Parabelsegmenten, Liniensegmenten und zwei Halbgeraden besteht.*

Bisektor von zwei Liniensegmenten

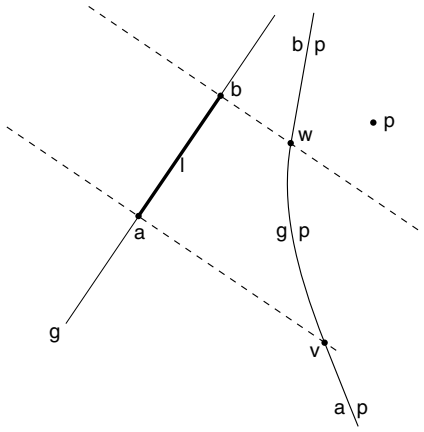


Abb. 5.29 Der Bisektor $B(l,p)$ eines Punktes p und eines Liniensegmentes l besteht aus drei Segmenten: einem Parabelstück und zwei Halbgeraden.

Beweis. Seien l_1 und l_2 zwei disjunkte Liniensegmente. Wir nehmen an, ihre senkrechten Streifen schneiden sich; andernfalls sind die Liniensegmente parallel, und die Betrachtung wird einfacher.

Wenn ein Teil von $B(l_1,l_2)$ im Durchschnitt der beiden Streifen verläuft, hat jeder Bisektorpunkt dort je einen inneren Punkt auf l_1 und auf l_2 zum nächsten Nachbarn, entsprechend Abbildung 5.28 (i). Hier stimmt deshalb $B(l_1,l_2)$ lokal mit einer *Winkelhalbierenden* von l_1 und l_2 überein. In Abbildung 5.30 gibt es zwei solche Stücke; sie sind dort mit $l_2|l_1$ beschriftet.

Während der Bisektor innerhalb des Streifens von l_i und außerhalb des Streifens von l_j verläuft, stimmt er mit der Parabel $B(l_i,e)$ überein, wobei e der näher gelegene Endpunkt von l_j ist. In Abbildung 5.30 trifft dies für die mit $d|l_1, c|l_1, d|l_1$ beschrifteten Bisektorstücke zu.

Außerhalb von beiden Streifen ist $B(l_1,l_2)$ Teil des Bisektors von zwei Endpunkten; siehe die mit $d|a$ und $d|b$ beschrifteten Halbgeraden in Abbildung 5.30.

Wenn man mit irgendeinem Stück von $B(l_1,l_2)$ startet, kann man den Bisektor von da aus in beide Richtungen eindeutig bis ins Unendliche weiterverfolgen: Wann immer man den Rand eines Streifens erreicht, hat man stetig Anschluß an das nächste Stück. Bei den unbeschränkten Endstücken kann es sich nur um Halbgeraden handeln, denn die Parabelstücke stoßen irgendwann an ihre Streifenränder, und der im Durchschnitt der Streifen enthaltene Teil von $B(l_1,l_2)$ ist beschränkt. □

Damit wissen wir, daß die Bisektoren von Liniensegmenten

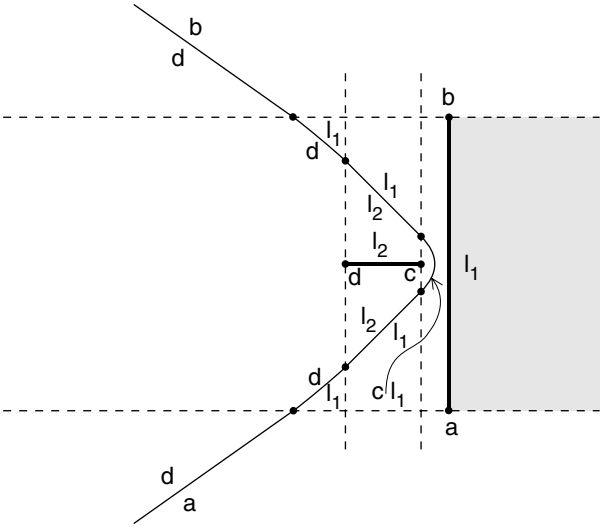


Abb. 5.30 Hier besteht der Bisektor der beiden Liniensegmente aus sieben Stücken.

Kurven sind, also keine flächigen Stücke enthalten, wie sie bei der L_1-Metrik vorkamen. Jetzt können wir problemlos das Voronoi-Diagramm $V(S)$ einer Menge von n Liniensegmenten definieren, analog zu Abschnitt 5.2 ab Seite 211. Abbildung 5.31 zeigt ein Voronoi-Diagramm von sieben Liniensegmenten.

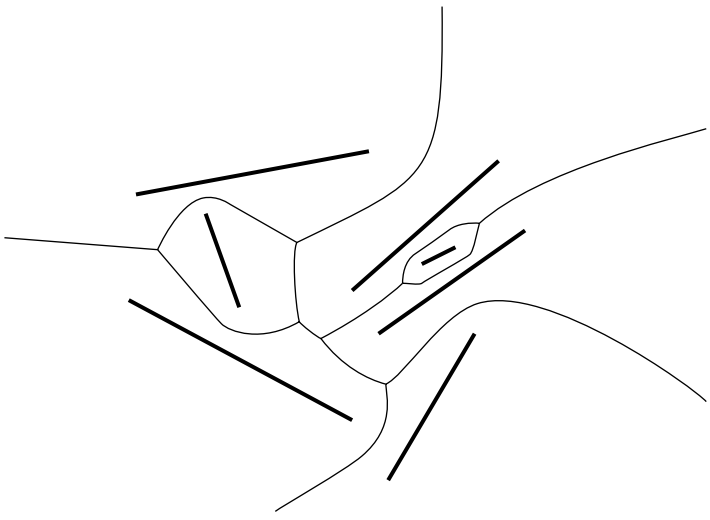


Abb. 5.31 Ein Voronoi-Diagramm von sieben Liniensegmenten.

verallgemeinerte Sternförmigkeit

Es hat einige wesentliche Eigenschaften mit dem Diagramm von Punkten gemeinsam: Zum Beispiel sind auch die Voronoi-Regionen von Liniensegmenten in einem verallgemeinerten Sinne sternförmig.

Lemma 5.25 *Sei S eine Menge von Liniensegmenten und $l \in S$. Dann enthält die Voronoi-Region $VR(l, S)$ mit jedem Punkt x auch das Liniensegment xy_x von x zum nächsten Punkt y_x auf l.*

Beweis. Angenommen, ein Punkt $z \in xy_x$ gehört zum Abschluß einer Region eines anderen Liniensegments l'. Dann wäre $|lz| \geq |l'z|$, und für den zu z nächsten Punkt y'_z auf l' würde folgen

$$\begin{aligned} |lx| = |y_x x| &= |y_x z| + |zx| \\ &= |lz| + |zx| \\ &\geq |l'z| + |zx| \\ &= |y'_z z| + |zx| \geq |y'_z x| \geq |l'x|, \end{aligned}$$

im Widerspruch zur Annahme $x \in VR(l, S)$. □

Daraus folgt sofort:

Korollar 5.26 *Die Voronoi-Regionen von Liniensegmenten sind zusammenhängende Mengen.*

In der *globalen Struktur* gibt es aber einen fundamentalen Unterschied zwischen dem Voronoi-Diagramm von Punkten und dem von Liniensegmenten: Zwei Regionen können mehr als eine gemeinsame Randkante besitzen. Abbildung 5.31 enthält ein Beispiel.

Zum Abschluß wollen wir noch bestimmen, aus *wie vielen* Stücken der Bisektor von zwei Liniensegmenten bestehen kann; daß es sich dabei nur um Stücke von Parabeln und Geraden handeln kann, hatten wir schon in Lemma 5.24 festgestellt.

Anzahl der Bisektorstücke

Lemma 5.27 *Seien $l_1 = ab$ und $l_2 = cd$ zwei disjunkte Liniensegmente in der Ebene. Dann besteht ihr Bisektor $B(l_1, l_2)$ aus maximal sieben Stücken.*

Beweis. Wir verwenden die Bezeichnungen aus Abbildung 5.30. Jeder der beiden Streifen wird durch sein Liniensegment in zwei Halbstreifen zerlegt. Aus Lemma 5.25 folgt, daß $B(l_1, l_2)$ jeden Halbstreifen von l_i höchstens einmal betreten und wieder verlassen kann: Die Liniensegmente zwischen den Bisektorpunkten und ihren nächsten Nachbarn auf l_i dürfen $B(l_1, l_2)$ ja nicht kreuzen!

Nun betrachten wir die konvexe Hülle der vier Endpunkte $\{a, b, c, d\}$ der beiden Liniensegmente. Weil l_1 und l_2 disjunkt sind,

muß mindestens eines von ihnen eine Kante von $ch(\{a, b, c, d\})$ sein.

In Abbildung 5.30 liegt nur l_1 auf dem Rand der konvexen Hülle. Jeder Punkt im Halbstreifen rechts von l_1 liegt näher an l_1 als an irgendeinem Punkt von l_2; folglich kann der Bisektor $B(l_1, l_2)$ den rechten Halbstreifen von l_1 nicht betreten. Er kann jeden der drei anderen Halbstreifen betreten und wieder verlassen und dabei sechs Streifenränder überqueren. Also besteht $B(l_1, l_2)$ in diesem Fall aus maximal sieben Stücken.

Liegen sogar beide Liniensegmente auf dem Rand von $ch(\{a, b, c, d\})$ wie in Abbildung 5.32, sind zwei Halbstreifen für $B(l_1, l_2)$ verboten, und es kann nur vier Überquerungen von Streifenrändern geben, also nur fünf Bisektorstücke. □

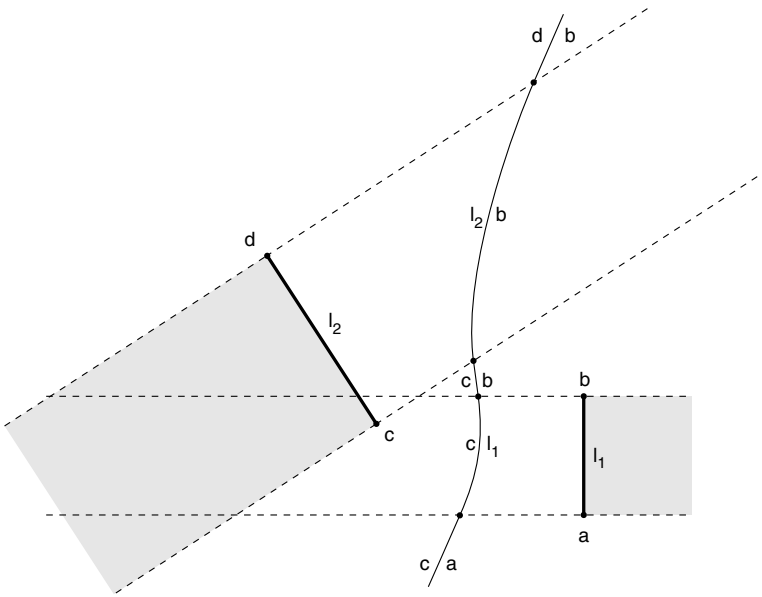


Abb. 5.32 Wenn die konvexe Hülle von $\{a, b, c, d\}$ jeden der vier Punkte enthält, kann der Bisektor $B(l_1, l_2)$ aus höchstens fünf Stücken bestehen.

Übungsaufgabe 5.13 Kann der Bisektor von zwei Liniensegmenten aus sieben Stücken bestehen, von denen keines im Durchschnitt der beiden Streifen verläuft?

Lemma 5.27 sagt uns, daß eine Voronoi-Kante aus maximal sieben Stücken bestehen kann; die Punkte, an denen solche Stücke zusammenstoßen, betrachten wir nicht als Voronoi-Knoten.

In Analogie zu Theorem 5.3 auf Seite 219 folgt hieraus und aus Korollar 5.26:

Theorem 5.28 *Das Voronoi-Diagramm von n disjunkten Liniensegmenten in der Ebene hat $O(n)$ viele Knoten und Kanten. Jede Kante besteht aus $O(1)$ vielen Stücken. Der Rand einer Voronoi-Region enthält im Mittel höchstens sechs Kanten.*

Eine weitreichende Verallgemeinerung der hier betrachteten Situation, nämlich das Voronoi-Diagramm beliebiger gekrümmter Kurvenstücke, wurde von Alt et al. [4] untersucht.

5.5.3 Anwendung: Bewegungsplanung für Roboter

Das Voronoi-Diagramm von Liniensegmenten hat eine wichtige Anwendung, die für die Bewegungsplanung von Robotern interessant ist.

kreisförmiger Roboter

Gegeben sei ein *kreisförmiger Roboter*, der von einem Startpunkt s zu einem Zielpunkt t bewegt werden soll.[21] Dabei sind Kollisionen mit n Hindernissen zu vermeiden, die durch Liniensegmente dargestellt werden, d. h. zu keinem Zeitpunkt darf das Innere des Kreises eines der Liniensegmente schneiden. Wir nehmen an, daß die Hindernisse paarweise disjunkt sind oder an den Enden zusammenstoßen.[22] Die Szene wird von vier Liniensegmenten eingeschlossen; siehe Abbildung 5.33.

kollisionsfreie Bewegung

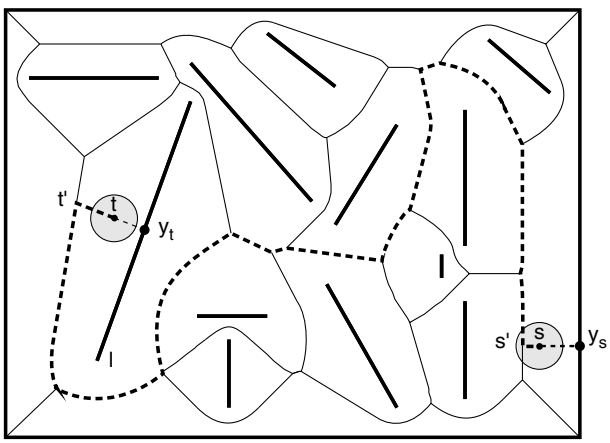


Abb. 5.33 Ein kreisförmiger Roboter soll von s nach t bewegt werden, ohne daß er zwischendurch mit einem der Hindernisse kollidiert.

[21] Die Positionsangaben beziehen sich immer auf den Kreismittelpunkt.
[22] Unsere Ergebnisse aus Abschnitt 5.5.2 gelten auch dafür.

Auf den ersten Blick könnte man meinen, zur Lösung dieser Aufgabe seien unendlich viele Wege von s nach t zu inspizieren und für jeden Weg festzustellen, ob er zu einer Kollision führt. Doch das Voronoi-Diagramm $V(S)$ der Liniensegmente hilft uns, hieraus ein *endliches Problem* zu machen, das sehr effizient gelöst werden kann.

Die Idee ist recht einfach: Während der Roboter sich durch eine Lücke zwischen zwei Liniensegmenten l_1 und l_2 hindurchbewegt, sollte er in jeder Position x seine Sicherheitsabstände

$$|xl_i| = \min_{y \in l_i} |xy|, \ i = 1, 2$$

maximieren. Dieses Ziel wird erreicht, wenn der Roboter stets *denselben Abstand* zu l_1 und zu l_2 hat! Das heißt: Er sollte dem Bisektor $B(l_1, l_2)$ folgen, zumindest so lange, bis sein Abstand zu einem anderen Hindernis l_3 kleiner wird als $|xl_i|$. Der Roboter sollte sich also im wesentlichen längs der Kanten des Voronoi-Diagramms $V(S)$ der Liniensegmente bewegen.

$V(S)$ maximiert Sicherheitsabstand

Wegen der vier umschließenden Liniensegmente ist $V(S)$ zusammenhängend.

Wenn Start- und Zielpunkt beide auf $V(S)$ liegen, läßt sich das Bahnplanungsproblem folgendermaßen lösen: Wir teilen die Voronoi-Kanten, die s und t enthalten, und machen s und t dadurch zu Knoten des Graphen $V(S)$. Dann bestimmen wir für jede Kante e, die Teil eines Bisektors $B(l_1, l_2)$ ist, den kleinsten Abstand

Start und Ziel auf $V(S)$

$$d_e = \min_{b \in e} |bl_i|, \ i = 1, 2$$

zu l_1 und l_2; der Wert von d_e gibt an, wie weit die engste Stelle zwischen l_1 und l_2 ist. Die Kante e wird entfernt, wenn d_e kleiner als der Roboterradius r ist, weil er hier sowieso nicht hindurchpaßt.

Nun führen wir vom Knoten s aus einen *Breitendurchlauf*[23] im verkleinerten Graphen durch und stellen dabei fest, ob t von s aus noch erreichbar ist. Ist dies der Fall, haben wir einen kollisionsfreien Weg von s nach t gefunden.

Breitendurchlauf

Im allgemeinen brauchen Start- und Zielpunkt nicht auf dem Voronoi-Diagramm der Hindernisse zu liegen. Dann gehen wir folgendermaßen vor:

Start und Ziel außerhalb von $V(S)$

Wir bestimmen das Hindernis l, dessen Voronoi-Region den Zielpunkt t enthält, und darauf den zu t nächsten Punkt y_t; vergleiche Abbildung 5.33. Nun verfolgen wir den Strahl von y_t durch t, er muß das Voronoi-Diagramm in einem Punkt t' treffen.

[23] Breitendurchlauf und Tiefendurchlauf in Graphen werden in nahezu jedem Buch über Algorithmen erklärt. Sie können beide in Zeit proportional zur Anzahl der Kanten – hier also $O(n)$ – ausgeführt werden.

Wenn wir es schaffen, den Roboter ohne Kollision zum Punkt t' zu bewegen, kann uns auch das gerade Stück von t' nach t keine Schwierigkeiten bereiten, vorausgesetzt, es gilt

$$|ty_t| \geq r.$$

Sollte diese Bedingung verletzt sein, ist am Zielpunkt t nicht genug Platz für den Roboter, und die gestellte Aufgabe ist unlösbar. Ist die Bedingung erfüllt, gilt für jeden Punkt $x \in t't$

$$|xl| = |xy_t| \geq |ty_t| \geq r,$$

so daß auch hier keine Kollision entsteht.

Mit dem Startpunkt s verfahren wir ebenso und konstruieren einen Punkt $s' \in V(S)$. Nun gilt folgender Satz:

Theorem 5.29 *Genau dann kann sich der Roboter kollisionsfrei von s nach t bewegen, wenn sein Radius r die Abstände $|sy_s|$ und $|ty_t|$ nicht übersteigt und wenn es eine kollisionsfreie Bewegung von s' nach t' längs der Kanten des Voronoi-Diagramms $V(S)$ gibt.*

Beweis. Daß die genannten Bedingungen hinreichend für die Existenz eines kollisionsfreien Weges sind, ist nach den vorausgegangenen Überlegungen klar. Sie sind aber auch notwendig, d. h. wenn wir auf diese Weise keinen Weg finden, dann gibt es keinen!

Sei nämlich π ein kollisionsfreier Weg von s nach t; er muß in dem von den äußeren vier Liniensegmenten umschlossenen Bereich verlaufen, braucht aber durchaus nicht dem Voronoi-Diagramm zu folgen. Die oben betrachtete Zuordnung

$$\rho : t \longmapsto t'$$

läßt sich nicht nur auf den Startpunkt s verallgemeinern, sondern auf *jeden* Punkt x der Szene, der nicht auf einem der Liniensegmente liegt: Wir bestimmen den zu x nächsten Punkt y_x auf dem nächstgelegenen Liniensegment und definieren als $x' = \rho(x)$ den ersten Punkt auf $V(S)$, der von dem Strahl getroffen wird, welcher von y_x aus durch x läuft. Offenbar gilt

$$\rho(x) = x \Longleftrightarrow x \in V(S);$$

für einen Punkt x aus $V(S)$ spielt es dabei gar keine Rolle, *welches* nächstgelegene Liniensegment wir bei unserer Definition von ρ verwenden.

Entscheidend ist nun, daß die Abbildung

$$\rho : I \setminus \bigcup_{i=1}^{n} l_i \longrightarrow V(S)$$

stetig ist; hierbei bezeichnet I das von den äußeren Segmenten umschlossene Gebiet. Wir wollen keinen formalen Beweis führen, sondern auf den wesentlichen Grund hinweisen: Wenn man auf einem Liniensegment l steht und den Rand der Voronoi-Region $VR(l, S)$ betrachtet, kann eine kleine Änderung der Blickrichtung nicht zu großen Sprüngen des angepeilten Randpunktes führen. Eine Situation, wie sie in Abbildung 5.34 dargestellt ist, widerspräche Lemma 5.25 auf Seite 252!

$x \longmapsto x'$ ist stetig

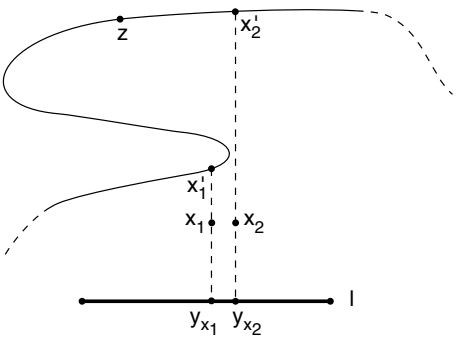


Abb. 5.34 Hätte der Rand von $VR(l, S)$ diese Gestalt, könnte das Liniensegment von z zum nächsten Punkt auf l nicht ganz in der Region von l enthalten sein.

Weil ρ stetig ist, stellt $\rho(\pi)$ einen *zusammenhängenden Weg* von $s' = \rho(s)$ nach $t' = \rho(t)$ dar, der ganz in $V(S)$ verläuft. Mit π ist auch dieser Weg kollisionsfrei: Ist nämlich der Abstand eines Punktes $x \in \pi$ zu seinem nächsten Hindernis l schon $\geq r$, gilt das erst recht für $x' = \rho(x)$, denn x' kann von l höchstens weiter entfernt sein als x, ist aber noch keinem anderen Hindernis näher als $|x'l|$. □

Theorem 5.29 liefert nicht nur eine Existenzaussage, sondern auch ein Lösungsverfahren:

Theorem 5.30 *Ist das Voronoi-Diagramm der n Liniensegmente vorhanden, läßt sich für einen beliebigen Roboterradius r und beliebige Punkte s und t in Zeit $O(n)$ ein kollisionsfreier Weg von s nach t bestimmen oder aber feststellen, daß es keinen gibt.*

Beweis. Zuerst lokalisieren wir in linearer Zeit die Punkte s und t in $V(S)$, berechnen s' und t' und fügen sie als Knoten ein. Dann entfernen wir in Zeit $O(n)$ alle Kanten aus $V(S)$, deren zugehörige Liniensegmente für den gegebenen Roboterradius zu eng beieinander stehen. Schließlich suchen wir mit Breitensuche in Zeit $O(n)$ einen Weg von s' nach t' in dem verbliebenen Graphen. □

Dieses Ergebnis geht auf Ó'Dúnlaing und Yap [111] zurück.

nicht kreisförmiger Roboter

Wie sollen wir vorgehen, wenn der Roboter nicht kreisförmig ist? Solange wir auf Drehbewegungen um einen Referenzpunkt im Innern des Roboters verzichten und nur Translationen betrachten, können wir die Form des Roboters durch eine beliebige konvexe Menge C beschreiben. Statt der euklidischen Metrik müssen wir dann die konvexe Distanzfunktion d_D für das Voronoi-Diagramm der Liniensegmente zugrunde legen, wobei D die Spiegelung von C am Referenzpunkt ist; vergleiche Übungsaufgabe 5.9 auf Seite 239.

Alle Resultate lassen sich auf diesen allgemeineren Fall übertragen! Weitere Ansätze zur Bahnplanung findet man zum Beispiel bei Alt und Yap [5, 6] oder bei Schwartz und Yap [128].

Will man nicht nur kollisionsfreie, sondern auch kostenoptimale Bahnen planen, steht man vor sehr schwierigen Problemen. Schon die Aufgabe, eine Leiter so zu transportieren, daß die Träger an beiden Enden zusammen einen möglichst geringen Weg zurücklegen, erweist sich als überraschend komplex; siehe Icking et al. [81].

weitere Verallgemeinerungen

Wir konnten in diesem Abschnitt nur einen kleinen Teil der möglichen Verallgemeinerungen betrachten, die das Voronoi-Diagramm in den letzten Jahren erfahren hat. Nicht erwähnt haben wir zum Beispiel das *Voronoi-Diagramm der entferntesten Nachbarn*, bei dem diejenigen Punkte der Ebene zu einer Region zusammengefaßt werden, die denselben *entferntesten* Nachbarn in der Punktmenge S besitzen. Mit seiner Hilfe kann man in Analogie zu Lemma 5.13 den kleinsten Kreis bestimmen, der die Punktmenge S umschließt. Bei einer anderen interessanten Variante werden den Punkten additive Gewichte zugeordnet, siehe das Applet

kleinster Kreis um S

`http://www.geometrylab.de/VoroAdd/`

Wer mehr über Eigenschaften und Anwendungen von Voronoi-Diagrammen wissen möchte, sei auf Aurenhammer [9] und Aurenhammer und Klein [10] verwiesen. Dort werden auch viele weitere Abstandsbegriffe erörtert, wie etwa der Hausdorff-Abstand; siehe z. B. Alt et al. [3].

Wenn wir die verschiedenen Arten von Voronoi-Diagrammen, die in den letzten Abschnitten betrachtet wurden, noch einmal Revue passieren lassen, stellt sich die Frage, worin ihre Gemeinsamkeiten bestehen und ob diese gemeinsamen strukturellen Merkmale für eine einheitliche Behandlung ausreichen. Diese Frage

hat zur Einführung der *abstrakten Voronoi-Diagramme* geführt. Sie beruhen nicht auf Objekten und einem Abstandsbegriff, sondern auf Bisektorkurven, die gewisse Eigenschaften haben müssen. Näheres hierzu findet sich in Klein [86] und in Klein et al. [87]. Andere Verallgemeinerungen betreffen Voronoi-Diagramme auf Graphen; siehe [76].

abstrakte Voronoi-Diagramme

Die Theorie der Voronoi-Diagramme ist in höheren Dimensionen noch nicht so weit entwickelt wie in der Ebene. So weiß man zwar, daß das Diagramm von n Punkten im $\mathbb{R}^3$ die Komplexität $\Theta(n^2)$ besitzt. Aber die strukturelle Komplexität des Diagramms von n Geraden in der euklidischen Metrik ist noch nicht bekannt. Erste Resultate gibt es für einfache konvexe Distanzfunktionen; siehe Chew et al. [31].

höherdimensionale Voronoi-Diagramme

Ebenso wie das Voronoi-Diagramm läßt sich auch die Delaunay-Triangulation auf verschiedene Arten verallgemeinern; siehe z. B. Gudmundsson et al. [66].

Lösungen der Übungsaufgaben

Übungsaufgabe 5.1 Nein. Der Punkt p ist innerer Punkt von jeder offenen Halbebene $D(p,q)$, $q \in S \setminus \{p\}$, und daher auch innerer Punkt des Durchschnitts $VR(p,S)$ von $n-1$ Halbebenen. Also ist eine ganze ε-Umgebung von p in der Voronoi-Region von p enthalten.

Übungsaufgabe 5.2 Ist ein Punkt x im Abschluß der Voronoi-Regionen von p und q enthalten, dann liegt er auch im Bisektor $B(p,q)$. Also kann x nicht innerhalb von $VR(p,S)$ liegen, sondern nur auf dem Rand.

Umgekehrt: Gilt $x \in \partial VR(p,S)$, liegen in jeder Umgebung $U_{1/n}(x)$ auch Punkte, die nicht zur Region von p gehören, sondern zu anderen Voronoi-Regionen. Weil es in S außer p nur endlich viele andere Punkte gibt, muß für $n \to \infty$ ein $q \in S$ unendlich oft dabei vorkommen. Also liegt x auch im Abschluß der Voronoi-Region von q.

Die beiden in der Aufgabenstellung angegebenen Charakterisierungen von $V(S)$ sind also gleichwertig. Es bleibt zu zeigen, daß sie mit unserer Definition übereinstimmen. Sei also

$$x \in \overline{VR(p,S)} \cap \overline{VR(q,S)}.$$

Dann liegt x auch im Bisektor $B(p,q)$. Andererseits gilt für jedes $r \neq p,q$ die Aussage

$$x \in \overline{VR(p,S)} \subset \overline{D(p,r)}.$$

Demnach kann x nicht näher an r liegen als an p, hat also mindestens die Punkte p und q als nächste Nachbarn und gehört deshalb zu $V(S)$.

Umgekehrt: Jedes x in $V(S)$ hat mindestens zwei nächste Nachbarn in S. Lemma 5.1 zeigt, daß x im Abschluß der Voronoi-Region von jedem nächsten Nachbarn enthalten ist, also insbesondere im Durchschnitt der Abschlüsse von zwei Regionen.

Übungsaufgabe 5.3 Ist $V(S)$ nicht zusammenhängend, gibt es eine Region $VR(p,S)$, die Teile von $V(S)$ voneinander trennt. Der Rand von $VR(p,S)$ enthält daher zwei unbeschränkte, disjunkte polygonale Ketten. Weil sie sich nicht treffen, die Region $VR(p,S)$ aber andererseits konvex ist, kann es sich nur um zwei parallele Geraden handeln.

Seien q und r die Punkte aus S, deren Regionen zu beiden Seiten an den Streifen $VR(p,S)$ angrenzen; siehe Abbildung 5.35. Dann liegen p, q, r auf derselben Geraden l. Gäbe es irgendein

$s \in S$, das nicht auf l liegt, würde der Bisektor $B(s,p)$ den Streifen kreuzen und die Region von p somit abschneiden.

Für n Punkte auf einer gemeinsamen Geraden besteht das Voronoi-Diagramm tatsächlich aus $n-1$ parallelen Bisektoren.

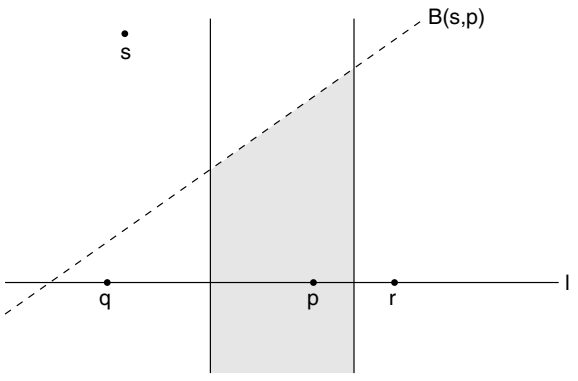


Abb. 5.35 Die Region von p kann sich nicht über den Bisektor $B(s,p)$ hinaus erstrecken.

Übungsaufgabe 5.4 Jede Kante kann schlimmstenfalls zu jedem der $O(n)$ vielen Streifen ein Segment beitragen; also gibt es höchstens $O(n^2)$ viele Segmente. Abbildung 5.36 gibt ein Beispiel dafür, daß für manche Punktmengen tatsächlich quadratisch viele Segmente auftreten können.

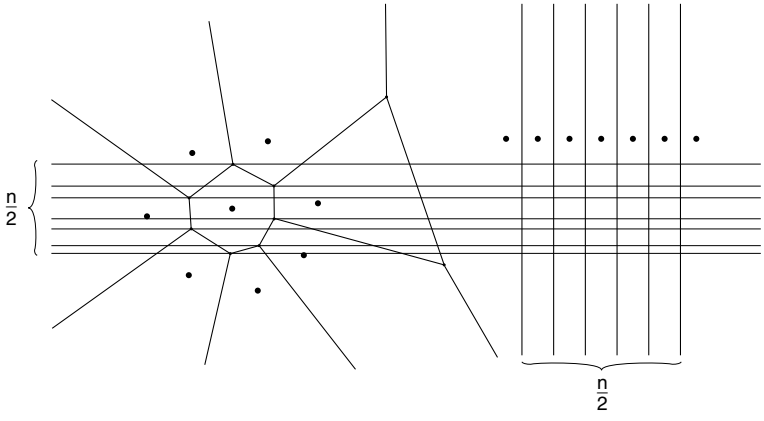


Abb. 5.36 Die $n/2$ vielen Voronoi-Kanten rechts zerfallen in jeweils $n/2$ viele Segmente.

Übungsaufgabe 5.5 Wir entfernen den Punkt p aus der Menge S und betrachten das Voronoi-Diagramm der Punkte aus $S' = S\backslash\{p\}$. Liegt p in der Region von q in $V(S')$, so ist q nächster Nachbar von p. Weil das Segment qp – eventuell ohne Endpunkt p – in $V(S')$ ganz zur Region von q gehört, kann in $V(S)$ nur die Region von p ein Stück davon abbekommen. Folglich haben die Regionen von q und p in $V(S)$ eine gemeinsame Voronoi-Kante.

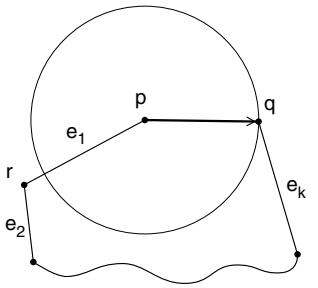


Abb. 5.37 Wenn q nächster Nachbar von p ist, muß die Kante e_1 länger sein als pq.

Übungsaufgabe 5.6

(i) Sei q der nächste Nachbar von p in S und T ein minimaler Spannbaum von S. Wenn die Kante pq nicht in T enthalten wäre, müßte es in T eine Kantenfolge $e_1, e_2, \ldots, e_k$ mit $k > 1$ von p nach q geben; siehe Abbildung 5.37. Weil nach Voraussetzung $|rp| \neq |qp|$ ist, kann r nur weiter von p entfernt sein als q. Wenn wir also aus T die Kante e_1 entfernten und statt dessen die kürzere Kante pq einfügten, ergäbe sich ein kürzerer Spannbaum[24] – Widerspruch!

(ii) Allgemein gilt folgendes Lemma, das häufig beim Korrektheitsbeweis für den Algorithmus von Kruskal angewendet wird:

Ist $S = P \cup Q$ eine disjunkte Zerlegung in nicht-leere Teilmengen und (p_0, q_0) das dichteste Punktepaar in $P \times Q$, d. h. gilt

$$|p_0q_0| = \min_{p\in P,\ q\in Q} |pq|,$$

dann gibt es einen minimalen Spannbaum von S, der die Kante p_0q_0 enthält. Hier gilt sogar: *Jeder* minimale Spannbaum T muß diese Kante enthalten! Sonst müßte nämlich die Kantenfolge von p_0 nach q_0 in T mindestens eine andere Kante e enthalten, die –

[24] Wenn man in einem Baum zwei Knoten mit einer zusätzlichen Kante verbindet, entsteht ein Zyklus. Entfernt man irgendeine Kante aus dem Zyklus, entsteht wieder ein Baum über derselben Knotenmenge.

wie p_0q_0 – einen Punkt in P mit einem Punkt in Q verbindet und deshalb echt länger ist als p_0q_0. Durch Austausch von e mit p_0q_0 entstünde ein kürzerer Spannbaum, was unmöglich ist.

Sei nun $|S| = n$ und $p_0 \in S$. Wir betrachten[25] für $i = 1, 2, \ldots, n-1$ die dichtesten Punktepaare

$$\begin{array}{rcl} (q_1, p_1) & \text{in} & \{p_0\} \times S \setminus \{p_0\} \\ (q_2, p_2) & \text{in} & \{p_0, p_1\} \times S \setminus \{p_0, p_1\} \\ \vdots & & \\ (q_{n-1}, p_{n-1}) & \text{in} & \{p_0, p_1, \ldots, p_{n-2}\} \times S \setminus \{p_0, p_1, \ldots, p_{n-2}\}. \end{array}$$

Weil die rechten Endpunkte p_i paarweise verschieden sind, handelt es sich um $n-1$ paarweise verschiedene Kanten, die in jedem minimalen Spannbaum von S enthalten sein müssen, wie wir oben gezeigt haben.

Ein Baum mit n Knoten besitzt aber nur $n-1$ Kanten![26] Also ist der minimale Spannbaum eindeutig bestimmt.

Übungsaufgabe 5.7

(i) Nach Definition ist genau dann pq eine Delaunay-Kante, wenn es im Voronoi-Diagramm $V(S)$ eine Kante e gibt, die zum Rand der Region $VR(p, S)$ und $VR(q, S)$ gehört. Für jeden Punkt x im Innern von e sind p und q die beiden einzigen nächsten Nachbarn in S; folglich enthält der Kreis durch p und q mit Mittelpunkt x keinen Punkt aus $S \setminus \{p, q\}$ im Innern oder auf dem Rand. Umgekehrt hat nach Lemma 5.1 jeder solche Kreis einen inneren Punkt einer Voronoi-Kante zum Mittelpunkt, an die $VR(p, S)$ und $VR(q, S)$ angrenzen.

(ii) Ja, wie Abbildung 5.38 zeigt. In diesem Fall kann allerdings weder p ein nächster Nachbar von q sein noch umgekehrt, wie der Beweis von Lemma 5.7 zeigt.

Übungsaufgabe 5.8 Wenn man über alle d Dreiecke einer Triangulation von S die Anzahl der Kanten (also jeweils drei) aufsummiert, hat man alle Kanten doppelt gezählt außer den r äußeren Dreieckskanten auf $ch(S)$. Also ist $2e = 3d+r$. Andererseits besagt die Eulersche Formel nach Theorem 1.1 auf Seite 15

$$n - e + (d + 1) = 2.$$

Die Behauptung folgt unmittelbar durch Einsetzen.

Übungsaufgabe 5.9 Das geht aus Abbildung 5.39 hervor.

[25] Dieses Vorgehen ist auch als *Algorithmus von Prim* bekannt.

[26] Das kann man direkt der Eulerschen Formel entnehmen; siehe Theorem 1.1.

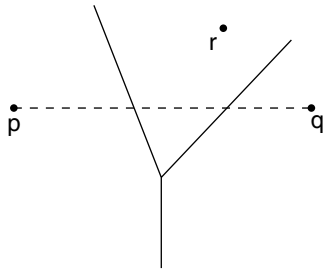


Abb. 5.38 Die Delaunay-Kante pq kreuzt die Region von r.

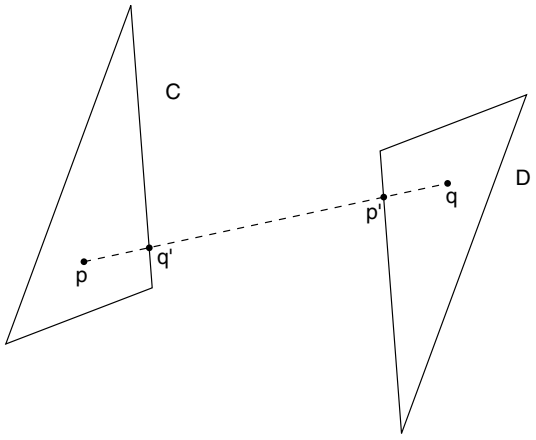


Abb. 5.39 Für das Spiegelbild D von C gilt $d_C(p,q) = \frac{|pq|}{|pq'|} = \frac{|qp|}{|qp'|} = d_D(q,p)$.

Übungsaufgabe 5.10

(i) Siehe Abbildung 5.40.

(ii) Sei $q \neq 0$ ein beliebiger Punkt im $\mathbb{R}^2$ und $a = L_1(0,q)$ der L_1-Abstand von q zum Nullpunkt. Dann liegt der Punkt

$$q' = \frac{1}{a}\ q$$

einerseits auf der Geraden durch 0 und q, wegen

$$L_1(0,q') = L_1(0, \frac{1}{a}\ q) = \frac{1}{a}\ L_1(0,q) = \frac{1}{a}\ a = 1$$

andererseits aber auch auf dem Rand des Einheitskreises R von L_1. Deswegen ist nach Definition von d_R

$$d_R(0,q) = \frac{|q|}{|q'|} = \frac{|q|}{|\frac{1}{a}\ q|} = a = L_1(0,q);$$

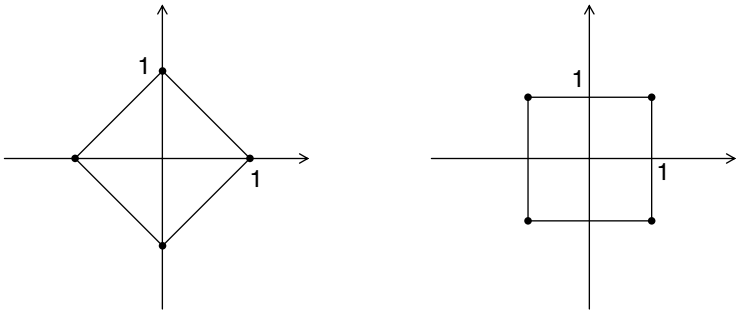


Abb. 5.40 Die Einheitskreise von L_1 und L_∞.

vergleiche Abbildung 5.20. Dieses Argument gilt nicht nur für L_1, sondern für *jede* Norm in der Ebene; wir haben beim Beweis lediglich verwendet, daß man Skalarfaktoren wie $\frac{1}{a}$ vorziehen darf.

Übungsaufgabe 5.11 Abbildung 5.41 zeigt drei Punkte p, q, r und ihr Voronoi-Diagramm in der L_1-Metrik. Die Punkte p und q liegen auf den diagonalen Ecken eines Quadrats; ihr Bisektor $B^*_C(p,q)$ wird definitionsgemäß aus den senkrechten Rändern der Viertelebenen und dem diagonalen Segment gebildet. Das Voronoi-Diagramm besteht aus $B^*_C(p,q)$ und $B^*_C(q,r) = B_C(q,r)$ und enthält keinen Voronoi-Knoten, obwohl der Punkt x alle drei Punkte p, q, r als nächste L_1-Nachbarn hat.[27]

Übungsaufgabe 5.12 Zunächst sei g die X-Achse und $p = (0,a)$ mit $a \neq 0$, siehe Abbildung 5.42. Genau dann gehört ein Punkt (x,y) zum Bisektor von p und g, wenn $y = \sqrt{x^2 + (y-a)^2}$ gilt, also

$$y = \frac{x^2}{2a} + \frac{a}{2}.$$

Durch diese Gleichung wird eine *Parabel* beschrieben. Auch für beliebige p, g mit $p \notin g$ ist $B(p,g)$ eine Parabel; denn wir können das Koordinatensystem so drehen und verschieben, daß g mit der X-Achse zusammenfällt und p auf der Y-Achse liegt. Dabei dreht $B(p,g)$ sich mit.

Liegt der Punkt p *auf* der Geraden g, ist nach Definition $B(p,g)$ diejenige Gerade, welche g am Punkt p im rechten Winkel kreuzt.

[27] Dasselbe gilt für jeden Punkt auf der Halbgeraden $B_C(p,q) \cap B_C(q,r)$.

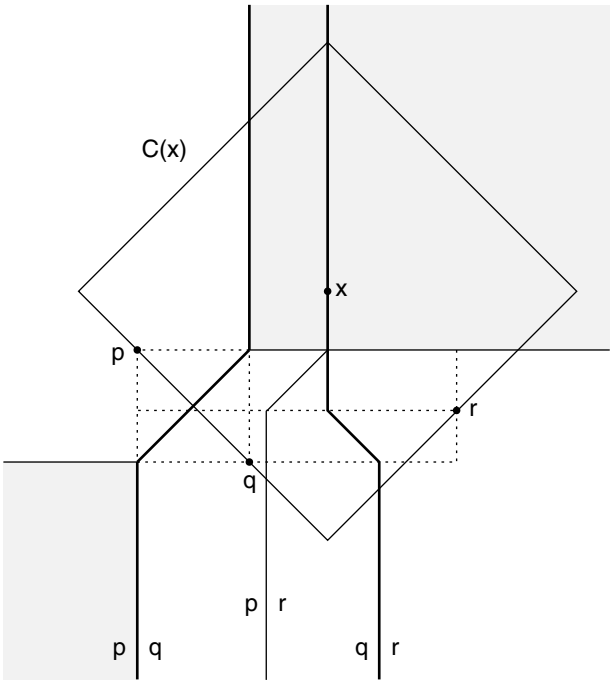


Abb. 5.41 Das Voronoi-Diagramm von $\{p, q, r\}$ in der L_1-Metrik.

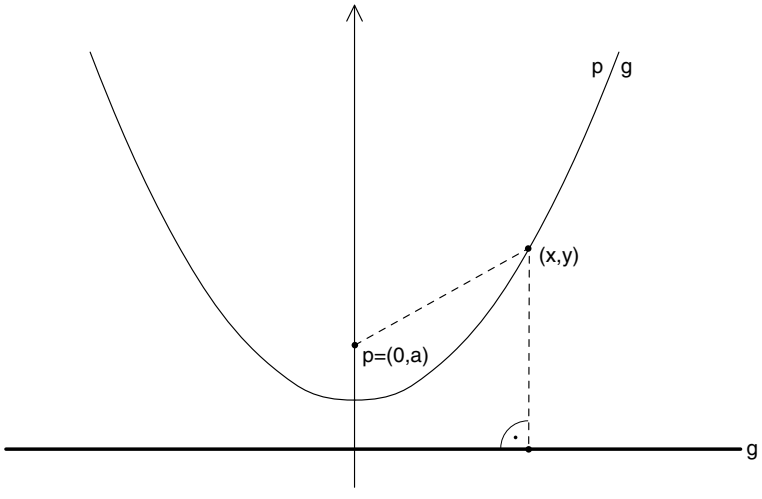


Abb. 5.42 Der Bisektor $B(p, g)$ von $p = (0, a)$ und $g = \{Y = 0\}$ ist die Parabel $\{Y = \frac{X^2}{2a} + \frac{a}{2}\}$.

Übungsaufgabe 5.13 Das kommt überraschenderweise tatsächlich vor; siehe Abbildung 5.43.

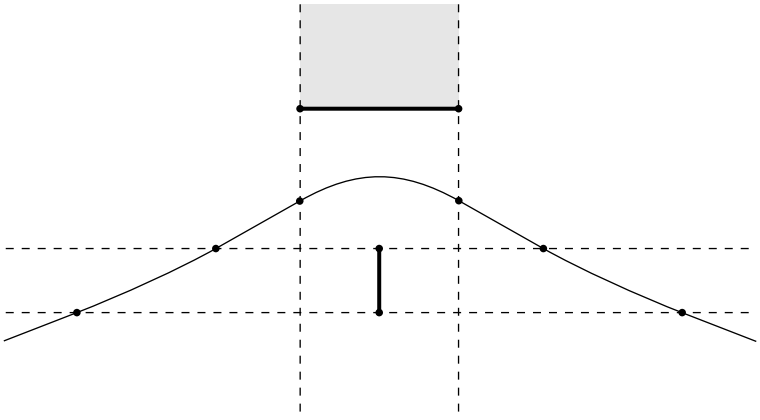


Abb. 5.43 Ein Bisektor mit sieben Stücken, von denen keines im Durchschnitt der beiden Streifen liegt.

6

Berechnung des Voronoi-Diagramms

In Kapitel 5 haben wir gesehen, wie nützlich das *Voronoi-Diagramm* $V(S)$ und die dazu duale *Delaunay-Triangulation* $DT(S)$ bei der Lösung von Distanzproblemen sind. Jetzt geht es uns darum, diese Strukturen für eine gegebene Menge S von n Punkten in der Ebene effizient zu berechnen.

Dabei genügt es, wenn wir $V(S)$ *oder* $DT(S)$ schnell konstruieren können; die jeweils duale Struktur läßt sich dann in Zeit $O(n)$ daraus ableiten.

Wir beschränken uns auf den Fall, daß S eine Menge von n Punkten ist, und legen die euklidische Metrik als Abstandsbegriff zugrunde. Zur Vereinfachung der Darstellung nehmen wir an, daß keine drei Punkte aus S auf einer gemeinsamen Geraden und keine vier auf einem gemeinsamen Kreisrand liegen. Als Folge davon ist das Voronoi-Diagramm zusammenhängend und enthält nur Voronoi-Knoten vom Grad drei; entsprechend ist $DT(S)$ wirklich eine Triangulation von S.

einschränkende Annahmen

In den folgenden Abschnitten stellen wir vier optimale Algorithmen vor, mit denen sich das Voronoi-Diagramm $V(S)$ in Zeit $O(n \log n)$ und Speicherplatz $O(n)$ berechnen läßt. Dabei kommen alle wichtigen algorithmischen Paradigmen zur Anwendung, die uns schon von anderen Beispielen bekannt sind: die randomisierte inkrementelle Konstruktion, das *sweep*-Verfahren, *divide and conquer* und die Technik der geometrischen Transformation.

viele Paradigmen anwendbar

Hinter jedem der vier Verfahren stecken originelle Ideen. Keines von ihnen kann als „naheliegend“ bezeichnet werden. In der Tat ist die Konstruktion des Voronoi-Diagramms eine schwierigere Aufgabe als etwa die Berechnung einer zweidimensionalen konvexen Hülle. Diese Tatsache wird im folgenden Abschnitt präzisiert.

6.1 Die untere Schranke

In Korollar 5.5 auf Seite 219 hatten wir schon festgestellt, daß die Berechnung des Voronoi-Diagramms einer n-elementigen Punktmenge S mindestens die Zeit $\Omega(n \log n)$ erfordert. Denn mit seiner Hilfe läßt sich ja die konvexe Hülle von S in linearer Zeit bestimmen. Die Konstruktion der konvexen Hülle benötigt aber selbst schon $\Omega(n \log n)$ viel Zeit, wie aus Lemma 4.2 folgt.

In Kapitel 5 haben wir gesehen, daß das Voronoi-Diagramm $V(S)$ viel mehr Information über die Punktmenge S enthält als nur ihre konvexe Hülle. Es ist deshalb nicht verwunderlich, daß seine Berechnung komplizierter ist. Der Unterschied zeigt sich folgendermaßen:

Wenn man die n gegebenen Punkte zum Beispiel nach einer Koordinate sortiert, läßt sich die konvexe Hülle anschließend in linearer Zeit konstruieren; siehe Theorem 4.7. Eine zweidimensionale konvexe Hülle zu berechnen ist also nicht schwieriger als Sortieren. Das trifft auf das Voronoi-Diagramm nicht zu!

auch nach Sortieren: $\Omega(n \log n)$

Theorem 6.1 *Angenommen, die n Punkte in S sind bereits nach aufsteigenden X-Koordinaten sortiert. Dann erfordert die Konstruktion des Voronoi-Diagramms immer noch $\Omega(n \log n)$ viel Zeit.*

Beweis. Wir zeigen, daß jeder Algorithmus, der das Voronoi-Diagramm von n sortierten Punkten berechnet, mit zusätzlichem linearem Zeitaufwand eine Lösung des Problems *ε-closeness* liefern kann; hierfür hatten wir in Korollar 1.6 auf Seite 40 die untere Schranke $\Omega(n \log n)$ bewiesen.

Seien also n positive[1] reelle Zahlen $y_1, \ldots, y_n$ und ein $\varepsilon > 0$ gegeben; wir müssen feststellen, ob es zwei Indizes $i \neq j$ mit $|y_i - y_j| < \varepsilon$ gibt.

Dazu bilden wir die nach X-Koordinaten sortierte Punktfolge

$$p_i = \left(\frac{i\varepsilon}{n}, y_i \right), \quad 1 \leq i \leq n$$

und konstruieren das Voronoi-Diagramm $V(S)$ von $S = \{p_1, p_2, \ldots, p_n\}$; siehe Abbildung 6.1.

Die Y-Achse wird durch das Voronoi-Diagramm $V(S)$ in zwei Halbgeraden und maximal $n-2$ Segmente zerlegt. Wir können diese Zerlegung in Zeit $O(n)$ bestimmen, indem wir die Y-Achse mit der in Kapitel 5 auf Seite 229 vorgestellten Technik durch $V(S)$ verfolgen.

Nun laufen wir auf der Y-Achse entlang; für jeden Abschnitt, der zu einer Voronoi-Region $VR(p_i, S)$ gehört, testen wir, ob er

[1] Diese Annahme stellt keine Einschränkung dar.

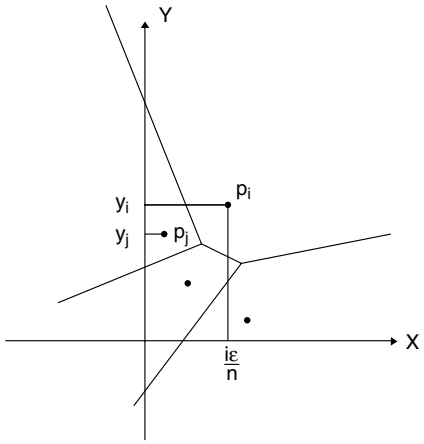


Abb. 6.1 Die Projektion von p_i auf die Y-Achse liegt nicht in der Voronoi-Region p_i.

die Projektion $(0, y_i)$ des Punktes p_i auf die Y-Achse im Innern oder als Endpunkt enthält.

Zwei Fälle sind möglich. Im ersten Fall besucht die Y-Achse *jede* der n Voronoi-Regionen, und für *jedes* i gilt $(0, y_i) \in \overline{VR(p_i, S)}$. Dann kennen wir nun die sortierte Reihenfolge der y_i auf der Y-Achse! Wir können deshalb in Zeit $O(n)$ ein dichtestes Paar bestimmen und nachsehen, ob sein Abstand kleiner als ε ist.

Im zweiten Fall gibt es mindestens einen Punkt p_i, dessen Projektion $(0, y_i)$ auf die Y-Achse *nicht* zum Abschluß von $VR(p_i, S)$ gehört, sondern in einer anderen Region $VR(p_j, S)$ liegt. Dieser Fall ist in Abbildung 6.1 dargestellt. Dann gilt

$$|y_i - y_j| \leq |(0, y_i)\, p_j| < |(0, y_i)\, p_i| = \frac{i\varepsilon}{n} \leq \varepsilon,$$

d. h. wir wissen jetzt, daß es tatsächlich zwei Eingabezahlen gibt, deren Abstand kleiner als ε ist. □

Dieser schöne Beweis stammt von Djidjev und Lingas [48].

Bevor wir nun mit der Diskussion zeitoptimaler Konstruktionsverfahren für das Voronoi-Diagramm beginnen, überlegen wir kurz, was sich mit naiven Verfahren erreichen ließe. naiv: $O(n^2 \log n)$

Übungsaufgabe 6.1 Man zeige, daß sich das Voronoi-Diagramm von n Punkten in Zeit $O(n^3)$ bzw. in Zeit $O(n^2 \log n)$ konstruieren läßt.

6.2 Inkrementelle Konstruktion

Das Voronoi-Diagramm oder die Delaunay-Triangulation durch sukzessive Hinzunahme der Punkte $p_1, \ldots, p_n$ *inkrementell* zu berechnen, ist eine sehr naheliegende Idee, auf der die ältesten Konstruktionsverfahren beruhen. Mit diesem Ansatz beschäftigt sich dieser Abschnitt.

Wir werden die inkrementelle Konstruktion für die Delaunay-Triangulation durchführen und uns zuerst überlegen, was eigentlich zu tun ist, um aus der Delaunay-Triangulation

$$DT_{i-1} = DT(S_{i-1})$$

der Punktmenge

$$S_{i-1} = \{p_1, p_2, \ldots, p_{i-1}\}$$

durch Hinzunahme von p_i die Triangulation DT_i von $S_i = \{p_1, \ldots, p_{i-1}, p_i\}$ zu gewinnen.

6.2.1 Aktualisierung der Delaunay-Triangulation

Erinnern wir uns an Lemma 5.18 auf Seite 235: Drei Punkte $q, r, t \in S_{i-1}$ bilden genau dann ein Delaunay-Dreieck in DT_{i-1}, wenn der Umkreis von $tria(q, r, t)$ keinen anderen Punkt aus S_{i-1} enthält.

Konflikt zwischen Dreieck und Punkt

Wenn jetzt der Punkt p_i zu S_{i-1} hinzukommt, kann er im Umkreis von mehreren Dreiecken von DT_{i-1} enthalten sein; wir sagen für ein Dreieck T in DT_{i-1}:

$$T \text{ ist mit } p_i \text{ in Konflikt } \iff \text{der Umkreis von } T \text{ enthält } p_i.$$

Beim Einfügen von p_i sind also zwei Aufgaben zu erledigen: Einerseits müssen neue Delaunay-Kanten eingefügt werden, die p_i mit anderen Punkten verbinden, andererseits müssen alle Dreiecke, die mit p_i in Konflikt stehen, umgebaut werden.

1. Fall: $p_i \in ch(S_{i-1})$

Der neue Punkt p_i kann entweder innerhalb der konvexen Hülle von S_{i-1} liegen oder außerhalb. Wir nehmen zunächst an, daß p_i *innerhalb* von $ch(S_{i-1})$ liegt. Dann gibt es ein Delaunay-Dreieck $tria(q, r, s)$, das den Punkt p_i enthält; siehe Abbildung 6.2 (i). Natürlich steht dieses Dreieck mit p_i in Konflikt.

einzufügende Kanten

Das folgende Lemma 6.2 erlaubt uns sofort, p_iq, p_ir und p_is als neue Delaunay-Kanten einzufügen.

Lemma 6.2 *Sei p_i im Umkreis eines Delaunay-Dreiecks $tria(q, s, r)$ von DT_{i-1} enthalten. Dann ist jedes der Liniensegmente p_iq, p_ir und p_is eine Delaunay-Kante in DT_i.*

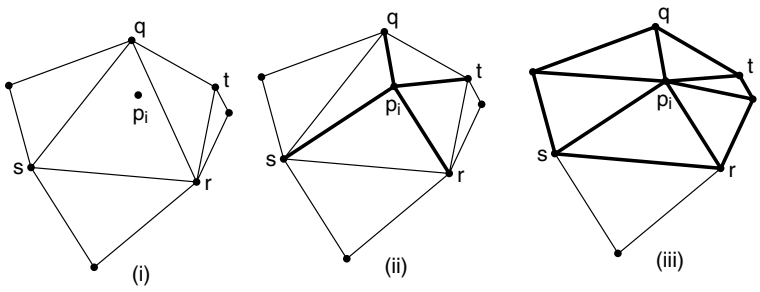


Abb. 6.2 Aktualisierung von DT_{i-1} nach Einfügen von p_i. In (ii) wurden neue Delaunay-Kanten zwischen p_i und q, r, s eingefügt, und die Kante qr wurde durch $p_i t$ ersetzt. Zwei weitere solche *edge flips* sind erforderlich, bevor DT_i in (iii) fertig ist.

Beweis. Sei C der Umkreis von q, s, r; siehe Abbildung 6.3. Außer p_i liegt kein Punkt von S_i in seinem Inneren. Wir bewegen den Mittelpunkt von C geradlinig auf q zu und erhalten dabei den Kontakt des Kreisrands mit q aufrecht. Sobald der Kreismittelpunkt den Bisektor $B(p_i, q)$ erreicht, haben wir einen Kreis C' gefunden, der nur p_i und q enthält. Nach Übungsaufgabe 5.7 ist $p_i q$ eine Delaunay-Kante von DT_i. Dasselbe Argument gilt für r und s. □

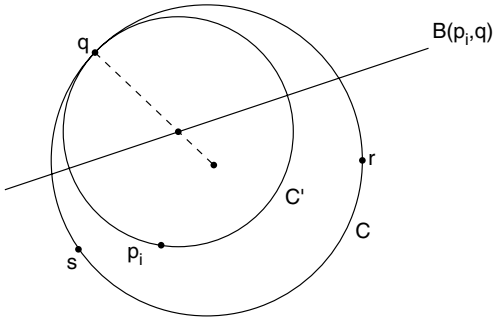


Abb. 6.3 Nur im Punkt q berührt der Kreis C' den Rand von C.

Damit haben wir bereits eine Triangulation von S_i erhalten. Ob es sich hierbei schon um die neue Delaunay-Triangulation DT_i handelt, hängt davon ab, ob es außer $tria(q, s, r)$ noch andere Dreiecke gibt, die mit p_i in Konflikt stehen. Wir inspizieren zunächst die unmittelbar angrenzenden Dreiecke.

In Abbildung 6.2 hat $tria(q, r, t)$ einen Konflikt mit p_i. Nach Lemma 6.2 muß die Kante $p_i t$ als neue Delaunay-Kante eingefügt

werden. Weil zwei Delaunay-Kanten sich nicht kreuzen können, muß die alte Kante qr entfernt werden. Diese Ersetzung ist nichts anderes als ein *edge flip*, wie wir ihn schon auf Seite 236 betrachtet haben.

Abbildung 6.2 (ii) zeigt den Zustand nach der Ersetzung von qr durch p_it. Als nächstes werden die Kanten rt und qs „geflippt“, weil auch hier die Umkreise ihrer von p_i abgewandten Dreiecke den Punkt p_i enthalten. Die fertige Delaunay-Triangulation DT_i ist in (iii) zu sehen.

Allgemein haben wir während dieses Ersetzungsvorgangs mit einer Menge von Dreiecken in Form eines Sterns zu tun, die p_i als gemeinsamen Eckpunkt haben; siehe Abbildung 6.4. Wir nennen sie den *Stern von* p_i.

Stern von p_i

Von jeder Kante zwischen p_i und einer Ecke des Sterns wissen wir, daß sie zu DT_i gehört. Von den äußeren Kanten des Sterns wissen wir es zunächst nicht.

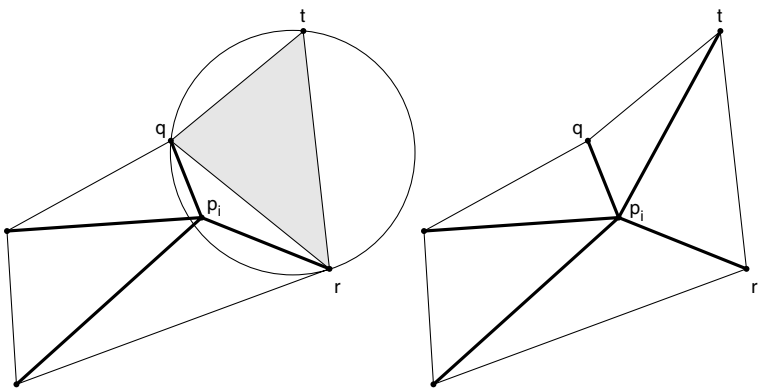


Abb. 6.4 Solange der Umkreis eines angrenzenden Dreiecks den Punkt p_i enthält, werden *edge flips* ausgeführt.

Deshalb testen wir jedes der angrenzenden Dreiecke auf Konflikt mit p_i; liegt ein Konflikt vor, wie beim Dreieck $tria(q, r, t)$ in Abbildung 6.4, wird ein *edge flip* ausgeführt. Dabei vergrößert sich der Stern, und p_i bekommt eine neue Kante, die zu DT_i gehören muß. Auch das neue angrenzende Dreieck ist zu testen, und so fort.

Übungsaufgabe 6.2 Man zeige, daß der Stern eines Punktes p_i stets von p_i aus sternförmig ist.

Wenn keine *edge flips* mehr ausführbar sind, stellt sich die Frage, ob es in der nun entstandenen Triangulation von S_i noch irgendwelche anderen Konflikte zwischen Punkten und Dreiecken gibt.

Zum Glück nicht, wie folgende Übungsaufgabe zeigt!

Übungsaufgabe 6.3 Wenn keines der an den Stern von p_i angrenzenden Dreiecke mit p_i in Konflikt steht, gibt es auch keine anderen Konflikte mehr.

Mit dem letzten *edge flip* haben wir also DT_i korrekt konstruiert. Dieses Verfahren wird im Applet

`http://www.geometrylab.de/VoroGlide/`

illustriert.

Im zweiten Fall liegt der neue Punkt p_i *außerhalb* der konvexen Hülle von S_{i-1}; siehe Abbildung 6.5. Man sieht leicht, daß jeder von p_i sichtbare Eckpunkt q von $ch(S_{i-1})$ eine Delaunay-Kante qp_i von DT_i definiert. Die Kanten von $ch(S_{i-1})$ zwischen diesen Eckpunkten brauchen nicht zu DT_i zu gehören; aber auch hier läßt sich mit *edge flips* die neue Delaunay-Triangulation DT_i konstruieren.

2. Fall: $p_i \notin ch(S_{i-1})$

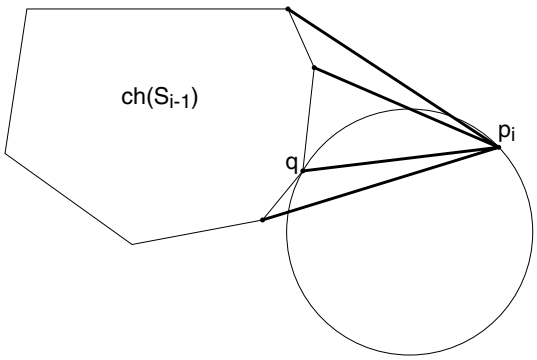


Abb. 6.5 Für jeden Eckpunkt q von $ch(S_{i-1})$, der von p_i aus sichtbar ist, bildet qp_i eine Delaunay-Kante in DT_i.

Man kann die beiden Fälle $p_i \in ch(S_{i-1})$ und $p_i \notin ch(S_{i-1})$ elegant zusammenfassen, wenn man *unendliche Delaunay-Dreiecke* einführt. Zu je zwei benachbarten Eckpunkten q_1, q_2 von $ch(S_{i-1})$ definiert man dazu

unendliche Delaunay-Dreiecke

$$tria(q_1, q_2, \infty) = H(q_1, q_2)$$

als die äußere Halbebene der Geraden durch q_1 und q_2. Den Umkreis von $tria(q_1, q_2, \infty)$ definieren wir ebenfalls als $H(q_1, q_2)$!

Diese Festsetzung ist nicht so abwegig, denn die äußere Halbebene ist in der Tat der Grenzwert aller Kreise durch q_1 und q_2, deren Mittelpunkt gegen ∞ wandert. Offenbar kann p_i in mehreren unbeschränkten Dreiecken enthalten sein.

Als wichtiges Zwischenergebnis halten wir folgendes fest:

Anzahl der Konfliktdreiecke $k_i \approx d_i$

Lemma 6.3 *Sei k_i die Anzahl der Dreiecke von DT_{i-1}, die mit dem Punkt p_i in Konflikt stehen, und sei d_i der Grad von p_i in der Delaunay-Triangulation DT_i. Dann gilt*

$$k_i + 1 \leq d_i \leq k_i + 2$$

Beweis. Wenn p_i in einem beschränkten Dreieck von DT_{i-1} liegt, so steht dieses mit p_i in Konflikt, und p_i erhält zunächst drei Kanten. Bei jedem nachfolgenden *edge flip* wird ein weiteres Konfliktdreieck beseitigt, und der Grad von p erhöht sich um eins, bis keine Konfliktdreiecke mehr übrig sind. In diesem Fall ist also $d_i = k_i + 2$.

Wenn der neue Punkt p_i außerhalb von $ch(S_{i-1})$ liegt und $c+1$ Ecken der konvexen Hülle sehen kann, so erhält er zunächst $c+1$ Kanten und ist in c unbeschränkten Konfliktdreiecken von DT_{i-1} enthalten. Danach erhöht sich der Grad für jedes weitere Konfliktdreieck um eins. Also ist $d_i = k_i + 1$. □

Hieraus folgt sofort eine obere Schranke für den Aktualisierungsaufwand.

Lemma 6.4 *Wenn das Dreieck von DT_{i-1} bekannt ist, das den neuen Punkt p_i enthält, so läßt sich DT_i in Zeit $O(d_i)$ aus DT_{i-1} konstruieren. Hierbei ist d_i der Grad von p_i in DT_i.*

Damit läßt sich die Aussage von Übungsaufgabe 6.1 schon einmal um den Faktor $\log n$ verbessern.

besser: $O(n^2)$

Korollar 6.5 *Die Delaunay-Triangulation von n Punkten läßt sich in Zeit $O(n^2)$ und linearem Speicherplatz konstruieren.*

Beweis. Für jedes $i \geq 4$ reicht $O(i)$ Zeit aus, um das Dreieck in DT_{i-1} zu bestimmen, welches den Punkt p_i enthält; man könnte etwa jedes einzelne Dreieck testen. Dann kann wegen Lemma 6.4 in Zeit $d_i \in O(i)$ die Delaunay-Triangulation DT_i aus DT_{i-1} konstruiert werden. Insgesamt ergibt das einen Zeitaufwand in der Größenordnung von

$$\sum_{i=4}^{n} i < \frac{n(n+1)}{2} \in O(n^2).$$ □

Aber auch diese obere Schranke läßt noch eine Lücke zur unteren Schranke von $\Omega(n \log n)$. Wo kann man Rechenzeit einsparen?

6.2.2 Lokalisierung mit dem Delaunay-DAG

Die im Beweis von Korollar 6.5 angewandte Methode, zum Lokalisieren des neuen Punktes p_i jedes Dreieck von DT_{i-1} zu inspizieren, ist recht zeitaufwendig. Wir wollen jetzt einen effizienteren Ansatz vorstellen.

Er beruht auf einer interessanten Hilfsstruktur, dem sogenannten *Delaunay-DAG*; hierbei steht *DAG* für *directed acyclic graph*, bezeichnet also einen gerichteten Graphen, in dem es keinen geschlossenen Weg mit identisch orientierten Kanten gibt.

Delaunay-DAG

Diese Struktur wurde zuerst von Boissonnat und Teillaud [18] eingeführt. Sie speichert in geeigneter Form alle Dreiecke, die bei der sukzessiven Konstruktion der Triangulationen $DT_3, DT_4, \ldots$ auftreten, und ihre Entwicklungsgeschichte.

Zur Motivation betrachten wir Abbildung 6.6. In DT_i rechts kommt ein neues Dreieck T mit Eckpunkt p_i vor. Nur eine seiner Kanten – nämlich sr – ist bereits in DT_{i-1} enthalten; dort berandet sie die beiden Dreiecke V und SV.

Das Dreieck V liegt auf derselben Seite von sr wie T und kommt in DT_i nicht mehr vor; man nennt V den Vater von T. Dagegen liegt SV auf der entgegengesetzten Seite von sr wie T und bleibt in DT_i erhalten; SV wird der *Stiefvater* von T genannt.

Vater verschwindet, Stiefvater bleibt

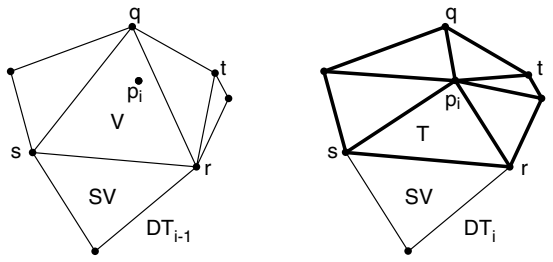


Abb. 6.6 In DT_i rechts wurde das Vater-Dreieck V aus DT_{i-1} durch seinen Sohn T ersetzt, während der Stiefvater SV noch existiert.

Für die Dreiecke V, SV und T gilt eine interessante Beziehung.

Lemma 6.6 *Jeder Punkt $p \notin S_i$, der mit dem Dreieck $T = tria(p_i, r, s)$ von DT_i in Konflikt steht, hat in DT_{i-1} einen Konflikt mit dem Vater V oder dem Stiefvater SV von T.*

Beweis. Siehe Abbildung 6.7. □

Die Definition von Vater und Stiefvater läßt sich auf unbeschränkte Dreiecke ausdehnen; siehe Abbildung 6.8.

Nun können wir den Delaunay-DAG der Folge $(p_1, p_2, \ldots, p_{i-1})$ definieren: Für jedes Dreieck, das in der Folge

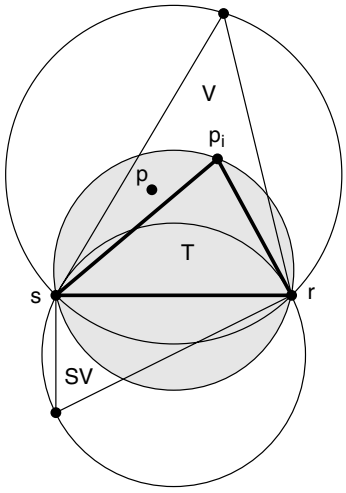


Abb. 6.7 Der Umkreis von T ist in der Vereinigung der Umkreise von V und SV enthalten.

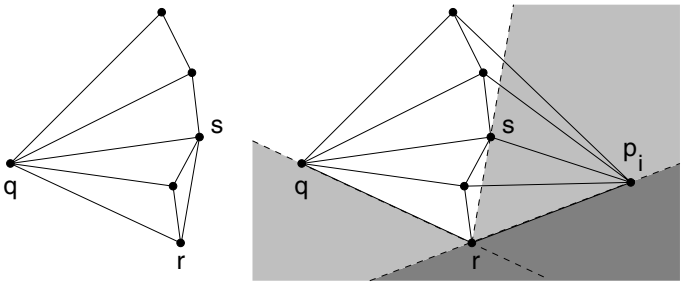


Abb. 6.8 Die Halbebene $H(r, p_i)$ ist in der Vereinigung von $H(q, r)$ und $H(r, s)$ enthalten; $H(r, s)$ ist der Vater, $H(q, r)$ der Stiefvater von $H(r, p_i)$.

DAG speichert die Geschichte

von Triangulationen $DT_3, DT_4, \ldots, DT_{i-1}$ vorkommt, existiert in DAG_{i-1} *ein* Knoten.[2] Gerichtete Kanten verlaufen von jedem Vater und Stiefvater zu seinen Söhnen bzw. Stiefsöhnen. Die vier Dreiecke in DT_3 sind Kinder eines gemeinsamen Wurzelknotens. Hierdurch wird die Entwicklungsgeschichte von DT_{i-1} dargestellt.

Abbildung 6.9 zeigt ein Beispiel für sechs Punkte. Zur Platzersparnis haben wir die Punkte nur durch ihre Indizes bezeichnet und statt $tria(p_h, p_j, p_k)$ bzw. $H(p_h, p_j)$ nur (h, j, k) bzw. (h, j) geschrieben.

[2] Auch für solche Dreiecke, die in mehreren Triangulationen vorkommen, wird nur ein Knoten angelegt.

Wir stellen uns vor, daß die Knoten von DAG_{i-1} in horizontale *Schichten* aufgeteilt sind. In der $(j-2)$-ten Schicht unterhalb der Wurzel stehen genau diejenigen Dreiecke, die in DT_j vorkommen, aber noch nicht in DT_{j-1}; sie alle haben p_j zum Eckpunkt, falls $j \geq 4$ ist. Die Dreiecke der aktuellen Triangulation DT_{i-1} brauchen nicht alle auf der untersten Schicht von DAG_{i-1} zu stehen, denn einige von ihnen können ja schon älter sein.

Übungsaufgabe 6.4

(i) Wenn man nur diejenigen Kanten betrachtet, die Väter mit ihren Söhnen verbinden, so ist der Delaunay-DAG ein Baum.

(ii) Wie viele Söhne kann ein Vater haben?

(iii) Kann es Dreiecke geben, die keine Söhne besitzen und trotzdem nicht mehr in der aktuellen Triangulation vorkommen?

Suche nach einem p_i enthaltenden Dreieck

Das Lokalisieren des neuen Punktes p_i wird nun durch den Delaunay-DAG wesentlich erleichtert. Die entscheidende Aussage folgt direkt aus Lemma 6.6:

Korollar 6.7 *Sei T ein Delaunay-Dreieck in $DT_3, \ldots, DT_{i-1}$, das mit p_i in Konflikt steht. Dann gibt es im Graphen DAG_{i-1} einen gerichteten Weg von der Wurzel nach T, der nur solche Dreiecke besucht, die selbst in Konflikt mit p_i sind.*

Wir starten deshalb in DAG_{i-1} von der Wurzel aus eine Tiefenerkundung und kehren jedesmal sofort um, wenn wir auf ein Dreieck stoßen, das mit p_i *keinen* Konflikt hat. Korollar 6.7 stellt sicher, daß bei dieser Erkundung *alle* Konfliktdreiecke gefunden werden, darunter eines aus der aktuellen Triangulation DT_{i-1}, das p_i enthält.

Besuch aller Konfliktdreiecke in Zeit $O(m_i)$

Sei m_i die Anzahl der Dreiecke in DAG_{i-1}, deren Vater oder Stiefvater mit p_i in Konflikt steht. Dann läßt sich die oben beschriebene Erkundung in Zeit $O(m_i)$ ausführen, denn außer der Wurzel und ihren vier Kindern werden ja nur solche Dreiecke besucht.

Aktualisierung von DT_{i-1} in Zeit $O(m_i)$

Die Aktualisierung von DT_{i-1} erfolgt durch *edge flips*, wie bisher. Nach Lemma 6.3 und Lemma 6.4 werden dazu $O(k_i)$ viele Schritte benötigt, wobei k_i die Anzahl der Konfliktdreiecke in DT_{i-1} bedeutet. Nach Lemma 6.6 ist $k_i \leq m_i$.

Nun muß noch die Hilfsstruktur DAG_{i-1} aktualisiert werden. Für jedes Dreieck in DT_i, das noch nicht in DT_{i-1} vorkommt, wird ein neuer Knoten erzeugt; diese Knoten werden als unterste Schicht an DAG_{i-1} angehängt.

Jetzt fehlen den neuen Knoten noch die Verbindungskanten von den Vätern und den Stiefvätern weiter oben in DAG_{i-1}.

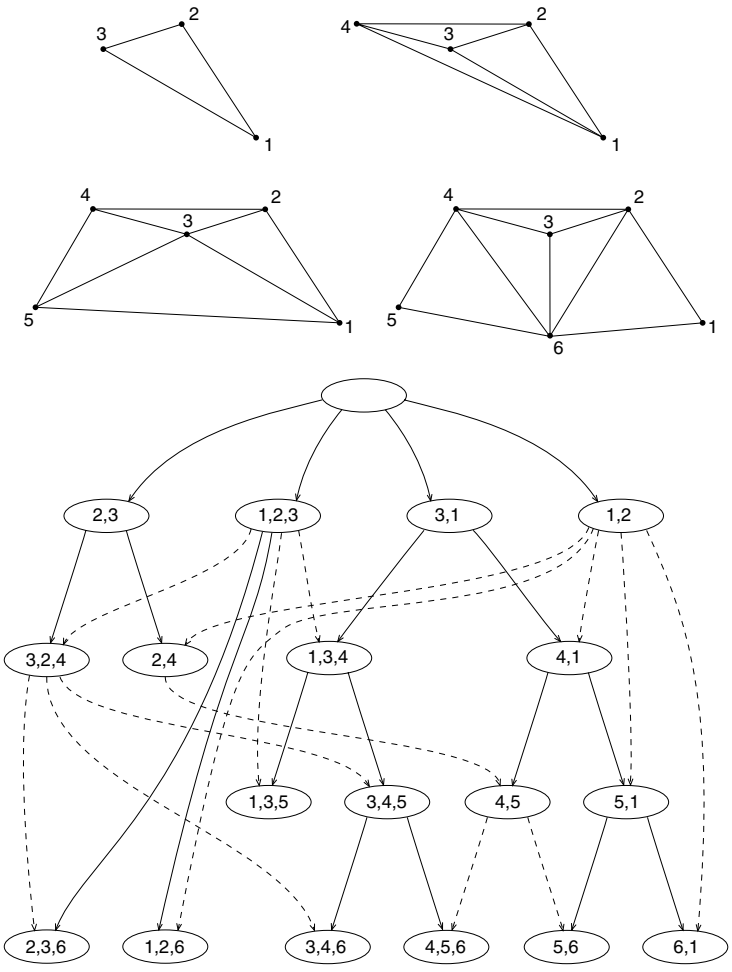


Abb. 6.9 Die Entstehung einer Delaunay-Triangulation durch sukzessives Einfügen und der zugehörige Delaunay-DAG. Die durchgezogenen Kanten verweisen jeweils vom Vater auf den Sohn, die gestrichelten vom Stiefvater auf den Sohn.

Aktualisierung von DAG_{i-1} in $O(m_i)$

Wann immer ein neues Dreieck durch *edge flip* erzeugt wird, steht fest, wer sein Vater und Stiefvater ist. Wenn wir also die Dreiecke von DT_{i-1} mit ihren Knoten in DAG_{i-1} verzeigern, können wir Vater und Stiefvater in Zeit $O(1)$ in DAG_{i-1} lokalisieren und die fehlenden Kanten zu den neuen Knoten in der untersten Schicht einfügen. Damit ist der Aufbau von DAG_i beendet. Weil es $d_i \leq k_i + 2 \leq m_i + 2$ neue Dreiecke in DT_i gibt, genügt

hierfür ebenfalls $O(m_i)$ viel Zeit.

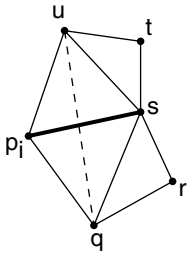
Zusammengefaßt ergibt sich damit folgender Algorithmus für das Einfügen eines neuen Punktes p_i: Algorithmus

besuche alle Konfliktdreiecke in DAG_{i-1},
    bis Dreieck T in DT_{i-1} mit $p_i \in T$ gefunden;

(* Delaunay-Triangulation aktualisieren *)

füge Delaunay-Kanten zwischen p_i und den Ecken von T ein;

while es gibt $tria(q, u, p_i)$, dessen Nachbardreieck $tria(q, s, u)$
        mit p_i in Konflikt steht **do**
    ersetze qu durch $p_i s$; (* *edge flip* *)
    Vater von $tria(s, u, p_i) := tria(s, u, q)$;
    Vater von $tria(q, s, p_i) := tria(s, u, q)$;
    Stiefvater von $tria(s, u, p_i) := tria(s, t, u)$;
    Stiefvater von $tria(q, s, p_i) := tria(q, r, s)$;



(* Delaunay-DAG aktualisieren *)

for jedes Dreieck T in DT_i mit Eckpunkt p_i **do**
    erzeuge neuen Knoten K; (* für neue Dreiecke *)
    verzeigere K mit T;
    erzeuge Kanten von den Knoten des Vaters und
        Stiefvaters von T nach K

Man beachte, daß die Zuweisungen von Vater- und Stiefvaterdreiecken nur vorläufig sind, solange noch *edge flips* ausgeführt werden. Dreiecke, die durch *edge flips* verschwinden, werden sofort gelöscht. War ein Dreieck zu irgendeinem Zeitpunkt Vater, kann es nicht mehr zu DT_i gehören. Für unbeschränkte Dreiecke verfährt man entsprechend.

Damit haben wir folgendes Zwischenergebnis:

Lemma 6.8 *Die Verwendung eines Delaunay-DAG macht es möglich, einen neuen Punkt p_i in Zeit $O(m_i)$ in die Delaunay-Triangulation DT_{i-1} einzufügen. Dabei bezeichnet m_i die Anzahl der Dreiecke, die in $DT_4, \ldots, DT_{i-1}$ vorkommen und deren Vater oder Stiefvater mit p_i in Konflikt steht.*

Obwohl bei diesem Einfügeverfahren immerhin ein Graph als Konfliktstruktur unterhalten wird, ist es konzeptuell doch recht einfach. Wie effizient es arbeitet, werden wir im nächsten Abschnitt untersuchen.

6.2.3 Randomisierung

mittlerer Zeitaufwand

Wir wollen jetzt den *mittleren Zeitaufwand* abschätzen, der bei der inkrementellen Konstruktion von DT_n nach diesem Verfahren entsteht, wenn jede der $n!$ möglichen Einfügereihenfolgen der Punkte $p_1, \ldots, p_n$ gleich wahrscheinlich ist. Diese Art der Mittelbildung hatten wir bereits in Abschnitt 4.1.2 kennengelernt.

Wir können nicht nur den Gesamtaufwand abschätzen, sondern sogar die mittleren Kosten jeder einzelnen Einfügeoperation:

Theorem 6.9 *Bei Verwendung eines Delaunay-DAG kann man in mittlerer Zeit $O(\log i)$ den Punkt p_i in die Delaunay-Triangulation von $i-1$ Punkten einfügen, falls jede Reihenfolge von $p_1, \ldots, p_i$ gleich wahrscheinlich ist. Der Speicherbedarf ist im Mittel linear.*

Speicherplatzbedarf

Beweis. Wir beginnen mit dem zu erwartenden *Speicherplatzbedarf*. Er wird durch die Größe von DAG_i bestimmt, weil während des sukzessiven Einfügens von $p_4, p_5, \ldots, p_i$ niemals Knoten aus dem aktuellen DAG entfernt werden. Der Graph DAG_i enthält außer der Wurzel genau einen Knoten für jedes Dreieck in $DT_3, \ldots, DT_i$ und für jeden Knoten höchstens zwei eingehende Kanten; für den Platzbedarf ist daher die Knotenzahl maßgeblich.

Wir ordnen jedes Dreieck T der ersten Triangulation DT_j zu, in der es auftritt, und setzen

$$\begin{aligned} b_j &= \text{Anzahl der Dreiecke in } DT_j \setminus DT_{j-1} \\ &= \text{Anzahl der Dreiecke in } DT_j \text{ mit Eckpunkt } p_j \\ &\leq \operatorname{grad}(p_j) + 1. \end{aligned}$$

$\operatorname{grad}(p_j)$

Hierbei ist $\operatorname{grad}(p_j)$ der Knotengrad von p_j; die Schranke $\operatorname{grad}(p_j)+1$ wird durch die beiden unendlichen Dreiecke erreicht, wenn p_j auf der konvexen Hülle liegt. Wenn wir $DT_2 = \emptyset$ definieren, gilt damit für die Knotenzahl

$$|DAG_i| = \sum_{j=3}^{i} b_j.$$

Rückwärtsanalyse

Um die zu erwartenden Werte der Zahlen b_j bequem abschätzen zu können, stellen wir uns vor, die Zeit liefe *rückwärts* ab: Zuerst wird p_i zufällig in der Menge S_i ausgewählt, dann p_{i-1} in der Restmenge und so fort. Stets erhält jeder Punkt dieselbe Chance.

Folglich ist p_j ein zufällig gewählter Knoten in der Triangulation DT_j der zu diesem Zeitpunkt noch vorhandenen j-elementigen Restmenge. Nach Theorem 5.3 auf Seite 219 hat daher p_j im Mittel einen Grad kleiner als sechs, d. h. für den Erwartungswert gilt:

$$E(b_j) \leq E(\text{grad}(p_j)) + 1 < 7.$$

Durch Summation folgt sofort, daß der zu erwartende Speicherplatzbedarf in $O(i)$ liegt.

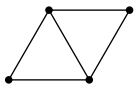
Zeitaufwand

Kommen wir nun zum *Zeitaufwand*. Die mittlere Anzahl von *edge flips* pro Einfügung ist nach Lemma 6.4 konstant, weil der mittlere Knotengrad kleiner als sechs ist. Aber wir müssen den DAG benutzen, um p_i zu lokalisieren. Wegen Lemma 6.8 müssen wir zeigen:

$$E(m_i) \in O(\log i),$$

wobei m_i die Anzahl aller bisher konstruierten Dreiecke bezeichnet, deren Vater oder Stiefvater mit p_i in Konflikt steht.

Sei T ein solches Dreieck; es trete zum ersten Mal in DT_j auf. Dann gehören Vater und Stiefvater von T zu DT_{j-1} und teilen sich – wenn sie beide endlich sind – eine Kante; vergleiche Abbildung 6.6. Wegen der Ähnlichkeit mit einem Fahrradrahmen nennt man die Kombination von Vater und Stiefvater auch ein *Fahrrad*. Es steht mit p_i in Konflikt, weil eines seiner beiden Dreiecke V, SV einen Konflikt mit p_i hat.



Fahrrad

Wir haben also für jedes der m_i Dreiecke ein Fahrrad gefunden, das mit p_i in Konflikt steht. Bei dieser Zuordnung kommt jedes Fahrrad höchstens dreimal an die Reihe, denn nur eines seiner beiden Dreiecke kann ein Vater sein,[3] und ein Vater hat nach Übungsaufgabe 6.4 höchstens drei Söhne.

Bezeichnet also f_i die Anzahl aller in $DT_3, \ldots, DT_{i-1}$ vorkommenden Fahrräder, die mit p_i in Konflikt stehen, so folgt

$$m_i \leq 3f_i.$$

Setzen wir weiterhin

$$f_{j,i} \quad = \quad \begin{array}{l}\text{Anzahl der Fahrräder in } DT_j \setminus DT_{j-1},\\ \text{die mit } p_i \text{ in Konflikt stehen,}\end{array}$$

so gilt

$$m_i \leq 3f_i = 3\sum_{j=3}^{i-1} f_{j,i}.$$

[3] Wird der Sohn eingefügt, stirbt der Vater, während der Stiefvater überlebt.

Jetzt beweisen wir, daß jede Zahl $f_{j,i}$ im Mittel einen Wert $< \frac{72}{j}$ besitzt. Durch Summation ergibt sich dann die logarithmische Schranke für den Mittelwert von m_i; vergleiche den Schluß des Beweises von Theorem 4.6 auf Seite 165.

Der Erwartungswert von $f_{j,i}$ hängt nicht davon ab, welchen speziellen Index $i > j$ wir betrachten.[4] Der Einfachheit halber diskutieren wir den Fall $i = j + 1$.

Rückwärtsanalyse

Jeder Punkt p in S_{j+1} hat dieselbe Chance, als p_{j+1} gewählt zu werden.

Fällt die Wahl von p_{j+1} auf p, so gibt es nach Lemma 6.3 in DT_j weniger als $\text{grad}(p)$ viele Dreiecke, die mit p in Konflikt stehen. Weil ein Dreieck zu höchstens drei Fahrrädern gehören kann, gibt es demnach weniger als $3 \cdot \text{grad}(p)$ Konfliktfahrräder in DT_j.

Von ihnen tragen nur diejenigen zu $f_{j,j+1}$ bei, die den jetzt zu wählenden Punkt p_j zum Eckpunkt haben und deshalb noch nicht in DT_{j-1} vorgekommen sind.

Für jeden Kandidaten q in $S_j = S_{j+1} \setminus \{p\}$ bezeichne $f_{q,p}$ die Anzahl der Fahrräder in DT_j mit Eckpunkt q und Konflikt mit p. Der gesuchte Mittelwert ergibt sich, indem wir alle Möglichkeiten für die Wahlen von p_{j+1} und p_j in Betracht ziehen:

$$\begin{aligned} E(f_{j,j+1}) &= \frac{1}{j+1} \sum_{p \in S_{j+1}} \frac{1}{j} \cdot \sum_{q \in S_{j+1} \setminus \{p\}} f_{q,p} \\ &< \frac{1}{j+1} \sum_{p \in S_{j+1}} \frac{1}{j} \cdot 4 \cdot 3 \cdot \text{grad}(p) \\ &= \frac{12}{j} \cdot \frac{1}{j+1} \sum_{p \in S_{j+1}} \text{grad}(p) \\ &< \frac{72}{j}. \end{aligned}$$

Bei der ersten Ungleichung ist zu beachten, daß die innere Summe über die Zahlen $f_{q,p}$ jedes Konfliktfahrrad von DT_j so oft aufzählt, wie es Eckpunkte besitzt, also höchstens viermal. Bei der letzten Abschätzung haben wir wieder Theorem 5.3 verwendet, wonach der mittlere Grad eines Delaunay-Knotens kleiner als sechs ist.

Damit ist Theorem 6.9 bewiesen. □

randomized incremental construction

Ein randomisiertes inkrementelles Verfahren zur Konstruktion des Voronoi-Diagramms in erwarteter Zeit $O(n \log n)$ wurde zuerst von Clarkson und Shor [34] angegeben; es hat aber einige Zeit gedauert, bis man eine einfache Darstellung fand. Ein wesentlicher

[4] Der Übergang von i zu einem anderen Index $i' > j$ stellt nur eine Umbenennung dar, die nichts an der Fairneß einer Zufallspermutation ändert.

Schritt zur Vereinfachung besteht in der von Seidel [134] eingeführten Rückwärtsanalyse. Auf dieser Technik beruhen auch unsere Mittelwertabschätzungen.

Devillers et al. [44] haben gezeigt, wie der Delaunay-DAG auch für das *Entfernen von Punkten* verwendet werden kann. Es ist übrigens gar nicht nötig, die Strukturen DAG_i und DT_i beide zu verwalten; die Information über die Nachbarschaftsbeziehungen der aktuellen Delaunay-Dreiecke kann man zusätzlich im DAG speichern und beim Einfügen eines Punktes aktualisieren.

Der *worst case* eines jeden randomisierten Verfahrens zur Konstruktion der Delaunay-Triangulation läßt sich auf $O(n^2)$ beschränken: Man braucht es ja nur mit dem in Korollar 6.5 beschriebenen $O(n^2)$-Verfahren um die Wette laufen zu lassen; sobald der erste Algorithmus terminiert, wird auch der andere angehalten.

Daß jedes inkrementelle Konstruktionsverfahren im *worst case* mindestens $\Omega(n^2)$ viele Schritte benötigt, zeigt das folgende von Meiser [105] stammende Beispiel in Abbildung 6.10. Es ist in der Sprache der Voronoi-Diagramme formuliert.

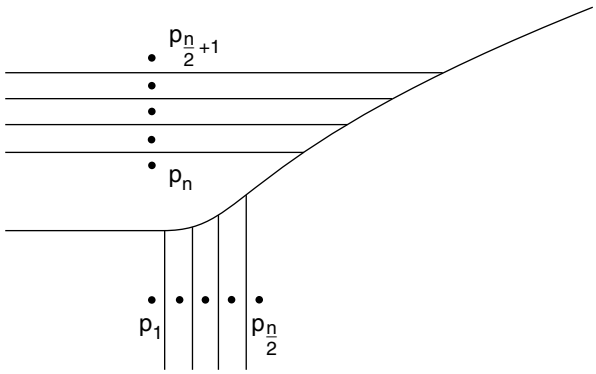


Abb. 6.10 Beim sukzessiven Einfügen von $p_1, \ldots, p_n$ sind $\Omega(n^2)$ viele Schritte nötig.

worst case $\Omega(n^2)$ unvermeidbar

Auf der X- und Y-Achse liegen je $n/2$ Punkte; die Punkte auf der X-Achse seien bereits eingefügt. Wenn nun die Punkte auf der Y-Achse von oben nach unten eingefügt werden, hat die Voronoi-Region des jeweils letzten Punktes p_{i-1} mit jeder Voronoi-Region $VR(p_1, S), \ldots, VR(p_{\frac{n}{2}}, S)$ eine gemeinsame Kante.[5] Alle diese $n/2$ vielen Voronoi-Kanten müssen beim Einfügen von p_i wieder entfernt und durch neue Kanten ersetzt werden. Das macht insgesamt $\Omega(n^2)$ viel Arbeit!

[5] In Abbildung 6.10 ist das für den Punkt p_n gut zu erkennen.

Übungsaufgabe 6.5 Man zeige, daß bei einer günstigeren Einfügereihenfolge das Voronoi-Diagramm in Abbildung 6.10 mit Aktualisierungsaufwand $O(n)$ konstruiert werden kann.

In diesem Abschnitt haben wir ein recht einfaches Verfahren zur Konstruktion der Delaunay-Triangulation vorgestellt, das im Mittel eine Laufzeit von $O(n \log n)$ erzielt. In den folgenden Abschnitten werden Algorithmen betrachtet, die diese Schranke auch im *worst case* einhalten.

6.3 Sweep

sweep beim Voronoi-Diagramm?

In Kapitel 2 hatten wir uns mit dem *sweep*-Verfahren beschäftigt und gesehen, daß sich damit eine ganze Reihe von geometrischen Problemen in der Ebene effizient lösen lassen. Die Frage drängt sich förmlich auf, ob man mit dieser Methode auch das Voronoi-Diagramm schnell konstruieren kann!

Auf den ersten Blick hat unser Problem große Ähnlichkeit mit der Aufgabe aus Abschnitt 2.3.2, alle Schnittpunkte von n Liniensegmenten zu bestimmen: Man möchte auch hier eine senkrechte *sweep line* von links nach rechts über die Ebene laufen lassen und die jeweils geschnittenen Voronoi-Kanten nach ihren Y-Koordinaten sortiert speichern, wie Abbildung 6.11 zeigt. Jeder Schnittpunkt zwischen zwei benachbarten Voronoi-Kanten würde ein *Schnittereignis* definieren; bei seinem Eintreten müßte die Sweep-Status-Struktur *SSS* entsprechend aktualisiert werden.

Übungsaufgabe 6.6 Man begründe, warum dieser Ansatz nicht funktioniert.

Diese Hinderungsgründe wirkten so abschreckend, daß nach Veröffentlichung des *sweep*-Verfahrens für Liniensegmente [16] acht Jahre vergingen, bevor Fortune [59] zeigte, daß man *doch* Voronoi-Diagramme mittels *sweep* berechnen kann.

6.3.1 Die Wellenfront

die Idee

Wir beschreiben hier eine vereinfachte Variante, die auf Seidel [131] zurückgeht. Die Idee ist ganz natürlich: Man speichert beim Sweepen nur diejenigen Teile des Voronoi-Diagramms links von der *sweep line*, die sich nicht mehr ändern können, unabhängig davon, welche Punkte aus S die *sweep line* künftig noch antreffen wird.

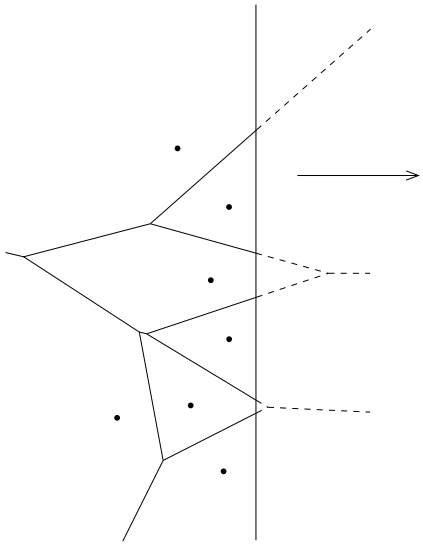


Abb. 6.11 Läßt sich so das Voronoi-Diagramm konstruieren?

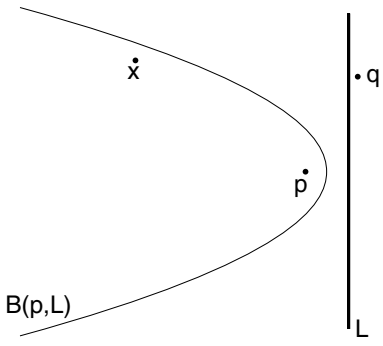


Abb. 6.12 Kein Punkt rechts von L kann das Gebiet zur Linken von $B(p, L)$ beeinflussen.

Abbildung 6.12 zeigt einen Punkt p links von der *sweep line* L und seinen Bisektor $B(p, L)$ – eine Parabel, wie wir in Übungsaufgabe 5.12 auf Seite 249 gesehen haben.

Sei x ein Punkt zur Linken von $B(p, L)$; dann liegt x näher an p als an irgendeinem Punkt q *auf* und erst recht *rechts von* der Geraden L. Also kann x nicht zur Voronoi-Region von q gehören! Mit anderen Worten: Das Gebiet links von $B(p, L)$ ist vor allen Punkten sicher, die rechts von der *sweep line* liegen; deren Regionen und Bisektoren können es nicht betreten.

Diese Tatsache nutzen wir für unseren Algorithmus aus.

Sei $S = \{p_1, p_2, \ldots, p_n\}$ in der Reihenfolge aufsteigender X-Koordinaten. Angenommen, die *sweep line* L befindet sich gerade zwischen p_i und p_{i+1}. Wir betrachten das Voronoi-Diagramm der Punkte $p_1, \ldots, p_i$ links von L *und der sweep-line* L *selbst*; siehe Abbildung 6.13.

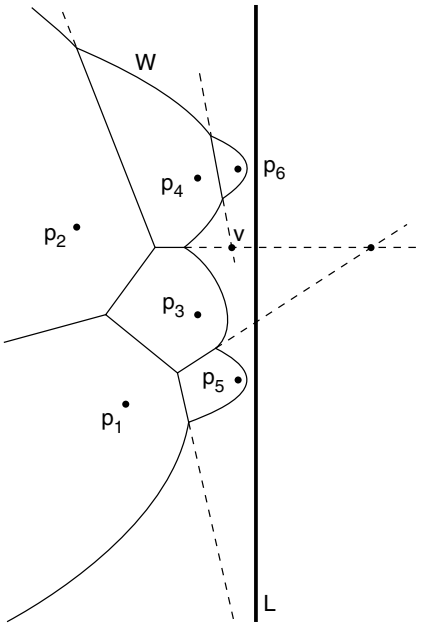


Abb. 6.13 Der Rand der Voronoi-Region von L bildet die *Wellenfront* W.

Der Rand der Voronoi-Region der *sweep line* besteht aus Parabelbögen, die Teile von Bisektoren $B(p_j, L)$ mit $j \leq i$ sind. Man nennt ihn die *Wellenfront* W. Die Parabeln selbst heißen *Wellen*, ihre in W vorkommenden Stücke nennen wir *Wellenstücke*.

Wellenfront

Links von der Wellenfront liegt ein Teil des Voronoi-Diagramms der Punkte $p_1, \ldots, p_i$, der zugleich ein Teil von $V(S)$ ist; denn die schon vorhandenen Voronoi-Kanten zwischen den Regionen von zwei Punkten aus S_i können sich nicht mehr ändern, wenn die *sweep line* auf neue Punkte stößt.

Während die *sweep line* L weiter nach rechts wandert, folgen ihr mit halber Geschwindigkeit alle Parabeln $B(p_j, L)$ mit $j \leq i$ und damit die gesamte Wellenfront W nach; daß die Wellen sich vorwärtsbewegen, kann man in Abbildung 6.14 gut erkennen.

Wo zwei in W benachbarte Wellenstücke, z. B. Teile von $B(r, L)$ und von $B(q, L)$ sich schneiden, rückt ihr Schnittpunkt längs des geraden Bisektors $B(r, q)$ vor. Die Verlängerungen dieser

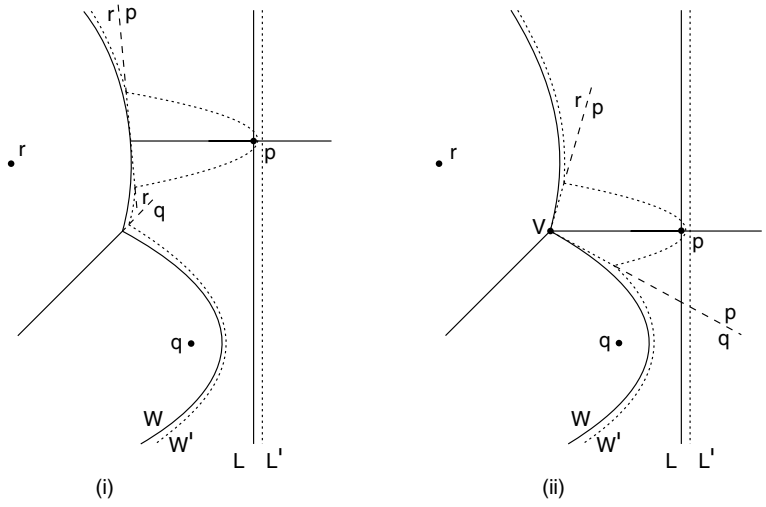


Abb. 6.14 Wenn die *sweep line* auf einen neuen Punkt trifft, entsteht eine neue Welle.

Bisektoren nach rechts über die Wellenfront hinaus werden *Spikes* genannt; sie sind in Abbildung 6.13 gestrichelt eingezeichnet. Spikes

Dadurch, daß die Wellenstücke längs der Spikes vorrücken, vergrößert sich das Teilstück von $V(S)$ links von W.

Als Sweep-Status-Struktur SSS speichern wir nun nicht die von der *sweep line* geschnittenen Voronoi-Kanten, sondern die aktuelle Wellenfront W, also den Rand von $VR(L, S_i \cup \{L\})$. Ihre Gestalt ist recht einfach.

Lemma 6.10 *Die Wellenfront W ist zusammenhängend und Y-monoton.* Struktur der Wellenfront

Beweis. Weil je zwei solche Bisektorparabeln sich schneiden, ist W zusammenhängend. Die Monotonie kann direkt aus der Form der Parabeln geschlossen werden oder aus der Verallgemeinerung von Lemma 5.25 auf Geraden: Danach liegt zu jedem Punkt x aus der Voronoi-Region von L auch das Liniensegment xy_x zum nächsten Punkt y_x auf L ganz in der Region von L. □

Die Wellenstücke in W sind also nach Y-Koordinaten geordnet. In Abbildung 6.13 tragen von unten nach oben die Punkte p_1, p_5, p_3, p_4, p_6, p_4, p_2 ein Stück zur Wellenfront bei. Mit den Bezeichnungen auf Seite 64 sind dies die *aktiven* Punkte von S. Die Voronoi-Regionen der *toten* Punkte berühren die Wellenfront nicht mehr, die *schlafenden* Punkte müssen von der *sweep line*

erst noch entdeckt werden. Offenbar kann *eine* Welle wie $B(p_4, L)$ *mehrere* Stücke zur Wellenfront beitragen! Sollte hier ein Problem lauern?

Übungsaufgabe 6.7 Sei S eine Menge von n Punkten links von der Senkrechten L.

Komplexität der Wellenfront

(i) Wieviele Stücke kann ein einzelner Bisektor $B(p, L)$, $p \in S$, zum Rand von $VR(S \cup \{L\})$ beitragen?

(ii) Man zeige, daß $VR(L, S \cup \{L\})$ insgesamt die Komplexität $O(n)$ hat.

6.3.2 Entwicklung der Wellenfront

Welche Ereignisse können eine strukturelle Änderung der Wellenfront auslösen?

Erstens kommt es vor, daß ein Wellenstück aus W *verschwindet*, weil es den Schnittpunkt seiner beiden Spikes erreicht. In Abbildung 6.13 zum Beispiel sind Wellenstücke von p_3, p_4, p_6 in W enthalten, und die zu $B(p_3, p_4)$ und $B(p_4, p_6)$ gehörigen Spikes schneiden sich im Punkt v. Beim weiteren Vorrücken von L wird das Wellenstück von p_4 immer kleiner, bis es schließlich ganz verschwindet. Der Punkt v liegt jetzt auf dem Rand der Regionen von p_3, p_4, p_6 und L, ist also insbesondere ein *Voronoi-Knoten* im Diagramm der Punkte links von L. In v vereinigen sich die benachbarten Wellenstücke von p_3 und p_6. Die neuen Nachbarn erzeugen einen neuen Spike, der Teil von $B(p_3, p_6)$ ist.

Wellenstück verschwindet

neues Wellenstück erscheint

Zweitens können neue Wellenstücke in W *erscheinen*. Das passiert immer dann, wenn die *sweep line* L auf einen neuen Punkt p trifft. In diesem Moment besteht der Bisektor $B(p, L)$ aus der waagerechten Geraden durch p; siehe Abbildung 6.14 (i). Sobald L sich weiterbewegt, verschwindet die rechte Halbgerade von $B(p, L)$, und die linke öffnet sich zu einer Parabel.

Wenn die linke Halbgerade von $B(p, L)$ ins Innere eines Wellenstücks von r in W trifft, wird dieses durch die neue Welle von p in zwei Teile gespalten.

Zwischen den Schnittpunkten der sich öffnenden Parabel von p mit dem Wellenstück von r entsteht eine Voronoi-Kante, die Teil von $B(p, r)$ ist. Ihre Verlängerungen über die Schnittpunkte hinaus bilden zwei neue Spikes; vergleiche die Spikes von p_6 in Abbildung 6.13.

Es ist ebenso möglich, daß die linke Halbgerade von $B(p, L)$ auf einen Punkt v in W stößt, bei dem sich zwei Wellenstücke, etwa von q und r, treffen; siehe Abbildung 6.14 (ii). In diesem Fall ist v ein Voronoi-Knoten, und die Parabel von p läuft zwei Spikes entlang, die zu $B(r, p)$ und $B(p, q)$ gehören.

Damit haben wir zwei Ereignistypen identifiziert, die zu einer Veränderung der *SSS* führen:

zwei Ereignistypen

Spike-Ereignis

- *Spike-Ereignis*: ein Wellenstück trifft auf den Schnittpunkt seiner beiden Spikes und verschwindet;

Punkt-Ereignis

- *Punkt-Ereignis*: die *sweep line* trifft auf einen neuen Punkt, und ein neues Wellenstück erscheint in der Wellenfront.

Andere Ereignisse können nicht eintreten: Solange die Spikes sich nicht schneiden und keine neuen Punkte entdeckt werden, laufen die Wellenstücke ungestört weiter.

Jetzt können wir den Sweep-Algorithmus in Angriff nehmen.

6.3.3 Der Sweep-Algorithmus für $V(S)$

Unsere Wellenstücke sind vom Typ

```
type tWellenStück = record
    Pkt: tPunkt;
    VorPkt, NachPkt: tPunkt;
```

Dabei bezeichnet *Pkt* den Punkt, aus dessen Welle das Wellenstück stammt. Weil die Wellenfront mehrere solche Stücke enthalten kann, geben wir zusätzlich die Punkte *VorPkt* und *NachPkt* an, zu denen der untere und der obere Nachbar dieses Wellenstücks in W gehören.

Damit ist das Wellenstück eindeutig beschrieben, weil zwei Wellen, etwa die von q und p, in W nicht in der Reihenfolge $\ldots q \ldots p \ldots q \ldots p$ vorkommen können; vgl. die Lösung von Übungsaufgabe 6.7. Die Punkte *VorPkt* und *NachPkt* nehmen bei den beiden äußersten Stücken der Wellenfront den Wert ∞ an.

Ähnlich wie in Abschnitt 2.3.2 verwenden wir neben der Sweep-Status-Struktur *SSS* eine *Ereignisstruktur ES*. Die in *ES* gespeicherten Ereignisse sind vom Typ

Ereignisstruktur

```
type tEreignis = record
    Zeit: real;
    case Typ: (SpikeEreig, PunktEreig) of
        SpikeEreig:
            AltWStück: tWellenStück;
            SchnittPkt: tPunkt;
        PunktEreig:
            NeuPkt: tPunkt;
```

Bei einem Spike-Ereignis verschwindet *AltWStück*, weil seine Spikes sich im *SchnittPkt* treffen. Ein Punkt-Ereignis tritt ein, wenn die *sweep line* auf *NeuPkt* stößt.

Die Zeit messen wir, wie üblich, durch die X-Koordinate der *sweep line*. Zeit

Übungsaufgabe 6.8 Welche Werte muß das Feld *Zeit* bei den verschiedenen Ereignistypen erhalten?

Die Sweep-Status-Struktur *SSS* muß (außer ihrer Initialisierung) folgende Operationen erlauben:

$UWStück(SSS, y, x)$:	bestimmt das Wellenstück unterhalb von $\{Y = y\}$ zum Zeitpunkt x.
$OWStück(SSS, y, x)$:	bestimmt das Wellenstück oberhalb von $\{Y = y\}$ zum Zeitpunkt x.

Meistens wird die Gerade $\{Y = y\}$ das Innere eines Wellenstücks von W treffen, so daß beide Funktionen dasselbe Ergebnis liefern. Außerdem brauchen wir

$FügeEin(SSS, WStück, x)$:	fügt Wellenstück *WStück* in die Ordnung zur Zeit x ein.
$Entferne(SSS, WStück, x)$:	entfernt Wellenstück *WStück* zur Zeit x.

Damit läßt sich der Sweep-Algorithmus zur Berechnung des Voronoi-Diagramms folgendermaßen beschreiben; man beachte die Ähnlichkeit zur Lösung des Liniensegment-Schnittproblems auf Seite 71. Zur Illustration der Bezeichnungen siehe Abbildung 6.15.

```
(* Initialisierung *)
initialisiere die Strukturen SSS und ES;
sortiere die n Punkte nach aufsteigenden X-Koordinaten;
erzeuge daraus Punkt-Ereignisse;
füge diese Ereignisse in die ES ein;

(* sweep *)
while ES ≠ ∅ do
    Ereignis := NächstesEreignis(ES);
    with Ereignis do
        case Typ of
        SpikeEreig:
            Entferne(SSS, AltWStück, Zeit);
            U := UWStück(SSS, SchnittPkt.Y, Zeit);
            O := OWStück(SSS, SchnittPkt.Y, Zeit);
            TesteSchnittErzeugeEreignis(U.VorPkt, U.Pkt,
                                        O.Pkt, Zeit);
            TesteSchnittErzeugeEreignis(U.Pkt, O.Pkt,
                                        O.NachPkt, Zeit);
```

PunktEreig:
$U := UWStück(SSS,\ NeuPkt.Y,\ Zeit)$;
$O := OWStück(SSS,\ NeuPkt.Y,\ Zeit)$;
with *NeuWStück* **do**
$Pkt := NeuPkt$;
$VorPkt = U.Pkt$;
$NachPkt = O.Pkt$;
$FügeEin(SSS,\ NeuWStück,\ Zeit)$;
$TesteSchnittErzeugeEreignis(U.VorPkt,\ U.Pkt,$
$NeuPkt,\ Zeit)$;
$TesteSchnittErzeugeEreignis(NeuPkt,\ O.Pkt,$
$O.NachPkt,\ Zeit)$

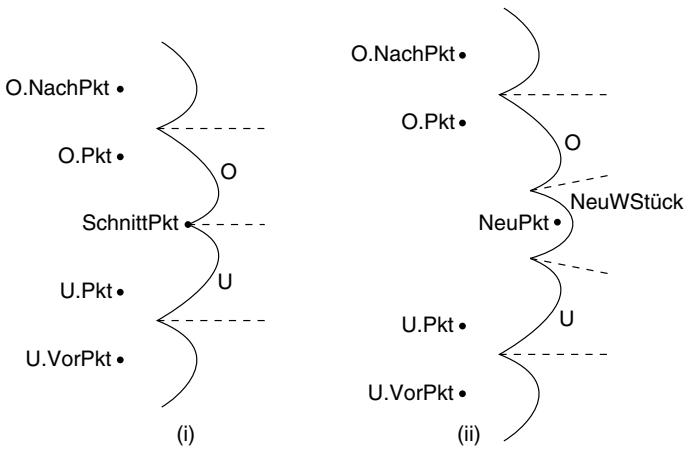


Abb. 6.15 Die Bearbeitung von Spike- und Punkt-Ereignissen.

Bei einem Aufruf $TesteSchnittErzeugeEreignis(p, q, r, x)$ zur Zeit x muß die Wellenfront konsekutive Stücke der Wellen von p, q, r enthalten. Es wird getestet, ob die zu $B(p, q)$ und $B(q, r)$ gehörenden Spikes sich rechts von der Wellenfront schneiden. Falls ja, wird ein entsprechendes Spike-Ereignis erzeugt und in *ES* eingefügt.

In Abbildung 6.15 (i) wird ein Spike-Ereignis bearbeitet. Das alte Wellenstück *AltWStück* ist bereits entfernt. Nun muß noch der neue Spike von *U.Pkt* und *O.Pkt* mit seinen beiden Nachbarn auf Schnitt getestet werden. In (ii) wurde bei Bearbeitung eines Punktereignisses soeben das Wellenstück *NeuWStück* eingefügt; in der Abbildung ist es symbolisch vergrößert. Jeder seiner beiden Spikes muß mit seinem anderen Nachbarn auf Schnitt getestet werden.

wo bleibt $V(S)$?

Unser Algorithmus berechnet korrekt alle strukturellen Veränderungen, die die Wellenfront während des *sweep* erfährt. Aber wo bleibt das Voronoi-Diagramm?

Nun, die Wellenfront zeichnet es sozusagen in den Sand,[6] und wir können den schon konstruierten Teil von $V(S)$ leicht von Ereignis zu Ereignis erweitern. Wenn am Ende kein Ereignis mehr zu bearbeiten ist, entfernen wir die *sweep line* und nehmen alle dann noch vorhandenen Spikes als unbeschränkte Voronoi-Kanten hinzu.

Effizienz

Kommen wir nun zur Effizienz dieses Verfahrens. Die Strukturen *SSS* und *ES* können so implementiert werden, daß sich jeder Zugriff in logarithmischer Zeit ausführen läßt. Die Größe der *SSS* bleibt in $O(n)$, wie Übungsaufgabe 6.7 (ii) gezeigt hat.

Um die Größe von *ES* ebenfalls in $O(n)$ zu halten, läßt sich die Idee anwenden, die wir auf Seite 73 in Kapitel 2 für die Schnittpunktbestimmung von Liniensegmenten beschrieben haben: Nur die Schnittereignisse zwischen direkt benachbarten Spikes werden in der Ereignisstruktur gespeichert.

Und wie oft wird insgesamt auf *SSS* und *ES* zugegriffen? Nur $O(1)$-mal für jeden Aufruf der Funktion *NächstesEreignis*, wie ein Blick auf den Algorithmus zeigt. Dabei werden genau n viele Punkt-Ereignisse bearbeitet und $O(n)$ viele Spike-Ereignisse, denn jedes von diesen liefert genau einen Voronoi-Knoten von $V(S)$!

Damit haben wir folgendes Ergebnis:

$O(n \log n)$ Zeit, $O(n)$ Platz

Theorem 6.11 *Das Voronoi-Diagramm von n Punkten in der Ebene läßt sich mit dem Sweep-Verfahren im* worst case *in Zeit $O(n \log n)$ und linearem Speicherplatz berechnen, und das ist optimal.*

Die Optimalität ergibt sich aus Korollar 5.5 auf Seite 219.

6.4 Divide and Conquer

Im letzten Abschnitt haben wir gesehen, wie sich das Voronoi-Diagramm optimal mit dem Sweep-Verfahren konstruieren läßt. Historisch älter und ebenfalls optimal ist der *divide-and-conquer*-Algorithmus von Shamos und Hoey [135], der nun vorgestellt werden soll. Seine Entdeckung hat die Entwicklung des Faches Algorithmische Geometrie ausgelöst.

Wie bei jeder Anwendung des *divide-and-conquer*-Prinzips wird auch hier das Problem in zwei möglichst gleich große Teile

[6] Die Punkte, an denen die Wellenstücke sich berühren, beschreiben die Voronoi-Kanten von $V(S)$.

zerlegt, die dann rekursiv gelöst werden. Die Hauptaufgabe besteht darin, die Lösungen der Teilprobleme zu einer Lösung des Gesamtproblems zusammenzusetzen.

Zur Zerlegung der n-elementigen Punktmenge S verwenden wir eine senkrechte oder waagerechte *Splitgerade*. Nach Übungsaufgabe 3.8 (i) auf Seite 134 kann man damit immer erreichen, daß der kleinere Teil von S zumindest etwa ein Viertel aller Punkte enthält.

Splitgerade

Um eine geeignete Splitgerade für S und später für die rekursiv entstehenden Teilmengen schnell bestimmen zu können, werden am Anfang einmal alle Punkte nach ihren X- und Y-Koordinaten sortiert und in zwei Verzeichnissen gespeichert. Anhand dieser beiden Verzeichnisse kann in linearer Zeit eine günstige Splitgerade bestimmt werden. Nach Aufteilung der Punktmenge längs dieser Splitgeraden können auch die Verzeichnisse in linearer Zeit auf die beiden Teilmengen aufgeteilt werden, so daß sie dort für den nächsten Rekursionsschritt zur Verfügung stehen. Halten wir fest:

Lemma 6.12 *Nach $O(n \log n)$ Vorbereitungszeit läßt sich die Punktmenge S rekursiv durch achsenparallele Splitgeraden so zerlegen, daß jeder Zerlegungsschritt einer Teilmenge T in Zeit $O(|T|)$ ausgeführt werden kann und zwei Teilmengen mit Mindestgröße $\frac{1}{4}(|T|-1)$ liefert.*

Aufwand für Initialisierung und *divide*

Entscheidend ist nun die Frage, wie schnell sich zwei Voronoi-Diagramme $V(L)$ und $V(R)$ zum Diagramm $V(L \cup R)$ zusammensetzen lassen, wenn L und R durch eine Splitgerade voneinander getrennt sind. Dieser Frage gehen wir im folgenden nach. Danach werden wir den Zeitbedarf dieses Verfahrens besprechen.

6.4.1 Mischen von zwei Voronoi-Diagrammen

Seien also L und R zwei Teilmengen von S, die von einer senkrechten Splitgeraden separiert werden. Angenommen, die Voronoi-Diagramme $V(L)$ und $V(R)$ sind schon vorhanden. Wie läßt sich daraus $V(S)$ berechnen?

Abbildung 6.16 zeigt, was dieser sogenannte *merge-Schritt* leisten soll; oben sind $V(L)$ und $V(R)$ dargestellt, zusammen mit den Punkten auf der jeweils anderen Seite der Splitgeraden, und unten das Diagramm $V(S)$.

merge-Schritt

In $V(S)$ gibt es zwei Arten von Voronoi-Kanten: Solche, bei denen die Punkte der beiden angrenzenden Regionen zu derselben Teilmenge L oder R von S gehören, und solche, die zwischen einer L-Region und einer R-Region verlaufen.

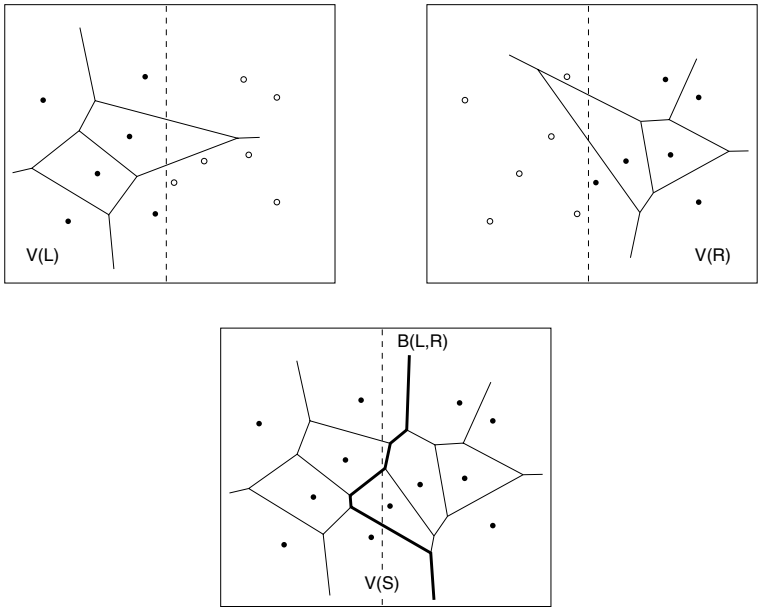


Abb. 6.16 Die Voronoi-Diagramme $V(L)$ und $V(R)$ werden zu $V(S)$ zusammengesetzt.

Die folgende Übungsaufgabe zeigt, daß alle Kanten der ersten Art schon in $V(L)$ oder $V(R)$ vorkommen, dort aber möglicherweise länger sind als in $V(S)$.

Übungsaufgabe 6.9 Sei e eine Voronoi-Kante in $V(S)$, die zwischen zwei Regionen von Punkten aus $L \subset S$ verläuft. Man zeige, daß es in $V(L)$ eine Kante e' gibt, die e enthält.

Neu zu berechnen sind aber alle Kanten der zweiten Art, die zwischen den Regionen von zwei Punkten $p \in L$ und $q \in R$ verlaufen. Zusammen bilden sie gerade die Menge

$$B(L,R) = \{x \in \mathbb{R}^2; \min_{p \in L} |px| = \min_{q \in R} |qx|\}$$

Bisektor von L und R

aller Punkte, die einen nächsten Nachbarn in L und in R besitzen. Man nennt $B(L,R)$ auch den *Bisektor von L und R*.

In Abbildung 6.16 bildet $B(L,R)$ eine polygonale Kette. Das ist kein Zufall.

Lemma 6.13 *Sind L und R durch eine senkrechte Gerade separiert, so ist ihr Bisektor $B(L,R)$ eine einzelne, Y-monotone polygonale Kette.*

Beweis. Angenommen, wir färben in $V(S)$ alle Regionen von Punkten aus L weiß und alle R-Regionen schwarz. Dann besteht $B(L, R)$ aus den Grenzen zwischen weißen und schwarzen Gebieten. Deshalb kann $B(L, R)$ im Prinzip nur aus unbeschränkten polygonalen Ketten und einfachen geschlossenen Ketten bestehen.[7]

Sei e eine Kante aus $B(L, R)$, die zwischen den Regionen von $p \in L$ und $q \in R$ verläuft. Weil p eine kleinere X-Koordinate als q hat, kann e – als Teil von $B(p, q)$ – nicht waagerecht sein; außerdem muß die Region von p zur Linken von e liegen. Weil dies für *alle* Kanten in $B(L, R)$ gilt, kann es keine geschlossenen Ketten geben, und die unbeschränkten Ketten sind monoton in Y.

Es bleibt also zu zeigen, daß nur eine solche Kette existiert. Angenommen, es gäbe mehrere. Wegen der Monotonie müßte jede Waagerechte alle Ketten schneiden, und zwar stets in derselben Reihenfolge, weil die Ketten sich nicht kreuzen. Andererseits muß auf der linken Seite jeder Kette weißes Gebiet liegen und schwarzes auf der rechten Seite, ein Widerspruch. □

Wir hatten oben festgestellt, daß alle Kanten von $V(S)$ zwischen den Regionen von Punkten aus *derselben* Teilmenge schon in $V(L)$ und $V(R)$ vertreten sind, aber möglicherweise für $V(S)$ zu lang sind. Wenn eine Kante e', die in $V(L)$ von v nach w' läuft, in $V(S)$ bereits an einem Punkt w endet, dann nur deshalb, weil sie in w auf eine Region $VR(r, S)$ trifft mit $r \in R$. Dann liegt der Punkt w aber auf $B(L, R)$! Das bedeutet: Wir können $V(S)$ aus $V(L)$ und $V(R)$ zusammensetzen, indem wir

- den Bisektor $B(L, R)$ von L und R konstruieren,
- die Diagramme $V(L)$ und $V(R)$ längs $B(L, R)$ aufschneiden[8] und
- den linken Teil von $V(L)$ mit $B(L, R)$ und mit dem rechten Teil von $V(R)$ zu $V(S)$ zusammensetzen.

Anders gesagt: $V(L)$ und $V(R)$ werden mit dem Faden $B(L, R)$ zusammengenäht, und die überstehenden Stücke werden abgeschnitten.

Die schwierigste Teilaufgabe besteht dabei in der Konstruktion des Bisektors von L und R; hiermit beschäftigt sich der folgende Abschnitt. Ist $B(L, R)$ erst vorhanden, lassen sich die beiden übri-

Hauptaufgabe: $B(L, R)$ konstruieren

[7] Das gilt sogar für alle disjunkten Zerlegungen $S = L \cup R$, auch wenn die Teilmengen nicht duch eine Gerade separiert werden.

Weil wir vorausgesetzt haben, daß sich an jedem Voronoi-Knoten nur drei Regionen treffen, sind je zwei Ketten disjunkt; andernfalls könnten sie sich in einzelnen Voronoi-Knoten berühren.

[8] Und nicht etwa längs der Splitgeraden!

gen Schritte leicht in Zeit $O(n)$ bewerkstelligen. Man kann sie sogar schon während der Berechnung von $B(L, R)$ ausführen.

6.4.2 Konstruktion von $B(L, R)$

Endstück von $B(L, R)$ suchen

Die Berechnung des Bisektors $B(L, R)$ erfolgt in zwei Phasen: Zuerst sucht man sich eines der beiden unbeschränkten Endstücke von $B(L, R)$; dann verfolgt man den Bisektor durch die Diagramme $V(L)$ und $V(R)$, bis man beim anderen Endstück ankommt.

Die Endstücke von $B(L, R)$ sind Halbgeraden, die ins Unendliche laufen. Dort werden wir nach ihnen suchen, einer Idee von Chew und Drysdale [29] folgend. Wir denken uns dazu $V(L)$ und $V(R)$ wie in Abbildung 6.17 übereinandergelegt. Der Durchschnitt von zwei unbeschränkten Regionen $VR(p, L)$ und $VR(q, R)$ kann leer, beschränkt oder unbeschränkt sein. Die unbeschränkten Durchschnitte bilden einen zusammenhängenden Ring im Unendlichen; wir haben sie in Abbildung 6.17 durchnumeriert.

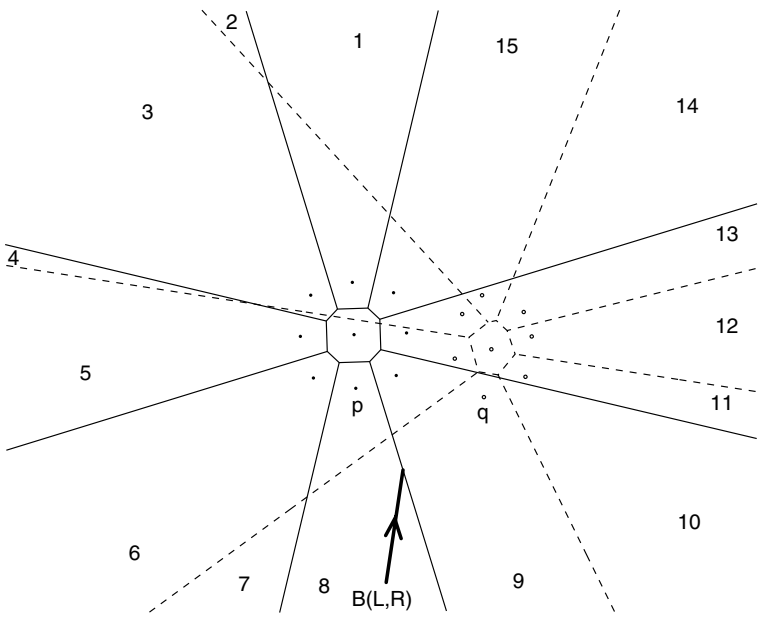


Abb. 6.17 Ein unbeschränktes Stück von $B(p, q)$ ist Endstück des Bisektors $B(L, R)$ von L und R.

Wir besuchen die unbeschränkten Durchschnitte der Reihe nach. Wenn wir in $VR(p, L) \cap VR(q, R)$ sind, testen wir, ob ein

unbeschränktes Stück von $B(p,q)$ darin enthalten ist; solch ein Test kann in $O(1)$ Zeit ausgeführt werden, weil es dabei nur auf die unbeschränkten Voronoi-Kanten ankommt.

Sobald wir fündig werden – und das müssen wir, denn es *gibt* ja zwei unbeschränkte Endstücke von $B(L,R)$ – haben wir ein Endstück von $B(L,R)$ gefunden, wie folgende Übungsaufgabe zeigt:

Übungsaufgabe 6.10 Wenn ein unbeschränktes Stück von $B(p,q)$ in $VR(p,L) \cap VR(q,R)$ enthalten ist, so gehört es zu $B(L,R)$.

In Abbildung 6.17 enthält Durchschnitt Nr. 8 ein unbeschränktes Stück von $B(L,R)$.

Wir wollen kurz darauf eingehen, *wie* man die unbeschränkten Durchschnitte der Regionen von $V(L)$ und $V(R)$ gegen den Uhrzeiger besucht. Die Voronoi-Diagramme werden in einer *QEDS* gespeichert, die es ermöglicht, die unbeschränkten Regionen *eines jeden Diagramms für sich* bequem zu durchlaufen; siehe Seite 19.

Man startet in $V(L)$ mit irgendeiner unbeschränkten Region $VR(p_0,L)$. Dann läuft man gegen den Uhrzeiger um $V(R)$ herum und testet für jede unbeschränkte Voronoi-Kante, ob sie mit $VR(p_0,L)$ einen unbeschränkten Durchschnitt hat. Auf diese Weise findet man eine Region $VR(q_0,R)$, die mit $VR(p_0,L)$ einen unbeschränkten Durchschnitt hat. Nach dieser Synchronisierung kann man sich in beiden Diagrammen simultan durch die anderen unbeschränkten Durchschnitte vorarbeiten.

Besuch der unbeschränkten $VR(p,L) \cap VR(q,R)$

Damit ist die erste Phase erfolgreich beendet; wir halten fest:

Lemma 6.14 *Sind $V(L)$ und $V(R)$ vorhanden, so läßt sich ein Endstück von $B(L,R)$ in Zeit $O(n)$ finden.*

In der zweiten Phase geht es darum, den Bisektor $B(L,R)$ durch die Diagramme $V(L)$, $V(R)$ weiterzuverfolgen. Wir nehmen an, daß wir in der ersten Phase das untere Endstück von $B(L,R)$ bestimmt haben und uns nun von dort hocharbeiten müssen. Dabei liegen die Regionen von L stets auf der linken Seite, die von R zur Rechten.

Orientierung

Angenommen, der Bisektor $B(L,R)$ betritt in einem Punkt v_1 die Region von $VR(p_1,L)$, während er gleichzeitig in $VR(q_1,R)$ verläuft; siehe Abbildung 6.18. Dann wird $B(L,R)$ in der Region von p_1 zunächst durch ein Stück von $B(p_1,q_1)$ fortgesetzt.

Dieses Stück endet, sobald es auf den Rand von $VR(p_1,L)$ oder auf den Rand von $VR(q_1,R)$ trifft. Wir müssen also feststellen, welcher der beiden Ränder zuerst erreicht wird. Dazu berechnen wir für das Stück von $B(p_1,q_1)$ ab v_1 die (wegen der Konvexität der Voronoi-Region eindeutig bestimmten) Schnittpunkte v_2 und

$B(L,R)$ verfolgen

w_2 mit $\partial VR(p_1, L)$ und $\partial VR(q_1, R)$ und testen, welcher von beiden näher an v_1 liegt. Dort endet das zu $B(L, R)$ gehörende Stück von $B(p_1, q_1)$.

In Abbildung 6.18 wird zuerst der Punkt w_2 erreicht; hier wechselt $B(L, R)$ in die Region von q_2 über und wird deshalb[9] durch ein Teilstück von $B(p_1, q_2)$ fortgesetzt. Nun gilt es zu prüfen, ob das neue Teilstück zuerst den Schnittpunkt v_3 mit dem Rand der Region von p_1 erreicht oder den Schnittpunkt w_3 mit $\partial VR(q_2, R)$.

Zu diesem Zweck müssen die beiden Schnittpunkte w_3 und v_3 auf ihren Rändern aufgesucht werden. Hierbei wenden wir die in Abschnitt 5.3.4 ab Seite 229 eingeführte Verfolgungstechnik gleichzeitig in $V(L)$ und $V(R)$ an. Wir beschränken uns im folgenden auf die Beschreibung der in $V(L)$ auszuführenden Schritte.

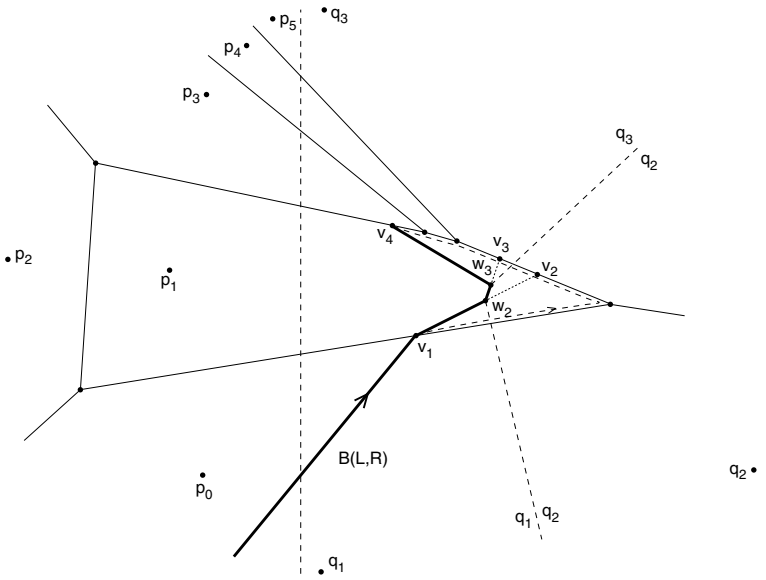


Abb. 6.18 Die Verfolgung des Bisektors $B(L, R)$.

Nach dem Betreten der Region $VR(p_1, L)$ im Punkt v_1 haben wir ihren Rand gegen den Uhrzeiger zunächst nach dem Schnittpunkt v_2 mit der Verlängerung von $B(p_1, q_1)$ abgesucht. Wenn wir nun anschließend nach v_3 suchen, dem Schnittpunkt des Randes mit $B(p_1, q_2)$, brauchen wir nur von v_2 aus weiterzulaufen: Auf

[9] Die formale Begründung dafür, daß ein Stück von $B(p_1, q_2)$, das im Durchschnitt der Regionen von p_1 und q_2 liegt, automatisch zu $B(L, R)$ gehört, ist dieselbe wie in Übungsaufgabe 6.10.

dem Stück von v_1 nach[10] v_2 kann v_3 ja schon der Konvexität der Voronoi-Regionen wegen nicht liegen! Ebenso können wir von v_3 aus gegen den Uhrzeiger weiterlaufen, bis v_4 erreicht ist.

Wie groß ist der Aufwand, den wir in $V(L)$ treiben müssen, um den Bisektor $B(L,R)$ während solch eines Besuchs einer Region $VR(p_1,L)$ zu verfolgen? Er ist offenbar linear in der Anzahl der Kanten von $\partial VR(p_1,L)$, die vom Eintrittspunkt zum Austrittspunkt führen, plus der Anzahl der „Knickstellen“ von $B(L,R)$ innerhalb und auf dem Rand der Region. Jede solche Knickstelle ist ein Voronoi-Knoten in $V(S)$, und davon gibt es insgesamt nur $O(n)$ viele. Auch die Ränder aller Regionen in $V(L)$ haben insgesamt nur $O(n)$ viele Kanten; aber Vorsicht! Könnte es nicht sein, daß der Bisektor $B(L,R)$ eine Region $VR(p,L)$ mehrfach besucht und unser Verfahren dabei dasselbe Stück des Randes mehrfach durchläuft?

Abschätzung des Aufwands

mehrfache Besuche

Wie die folgende Übungsaufgabe zeigt, sind mehrfache Besuche durchaus möglich.

Übungsaufgabe 6.11 Man gebe ein Beispiel an, bei dem der Bisektor $B(L,R)$ eine Region $VR(p,L)$ von $V(L)$ mehrfach besucht.

Aber auch bei mehrfachen Besuchen derselben Region wird kein Stück des Randes mehrfach durchlaufen! Sei nämlich beim k-ten Besuch in $VR(p,L)$ der Eintrittspunkt $B(L,R)$ mit e_k und der Austrittspunkt mit a_k bezeichnet. Wir betrachten zwei Besuche, ohne Einschränkung den ersten und den zweiten, und schauen uns die zyklische Reihenfolge der vier Punkte e_1, a_1, e_2, a_2 auf dem Rand von $VR(p,L)$ an; siehe Abbildung 6.19.

Besuchsreihenfolge

Die „geschachtelte“ Reihenfolge (e_1, e_2, a_2, a_1) in (i) ist nicht möglich, denn zu welcher Region sollen die Punkte zur linken Seite des Stücks von $B(L,R)$ von e_2 nach a_2 gehören? Sie müssen zur Region eines Punktes von L gehören, und ihr nächster Nachbar in L ist p_1. Aber von p_1 sind sie durch das andere Stück von $B(L,R)$ abgeschnitten!

Die in (ii) gezeigte „umgekehrt geschachtelte“ Reihenfolge (e_1, a_2, e_2, a_1) scheidet aus demselben Grunde aus. Ferner sind (e_1, e_2, a_1, a_2) und (e_1, a_2, a_1, e_2) unmöglich, weil die beiden Stücke von $B(L,R)$ sich nicht kreuzen können.[11] Für die vier Punkte bleibt damit nur die in (iii) gezeigte Reihenfolge (e_1, a_1, e_2, a_2) übrig. Hier sind aber die von e_k nach a_k führenden Randstücke disjunkt!

[10] Der Rand von $VR(p_1,L)$ sei gegen den Uhrzeiger orientiert.

[11] Wir wissen ja aus Lemma 6.13, daß $B(L,R)$ aus einer einzelnen polygonalen Kette besteht.

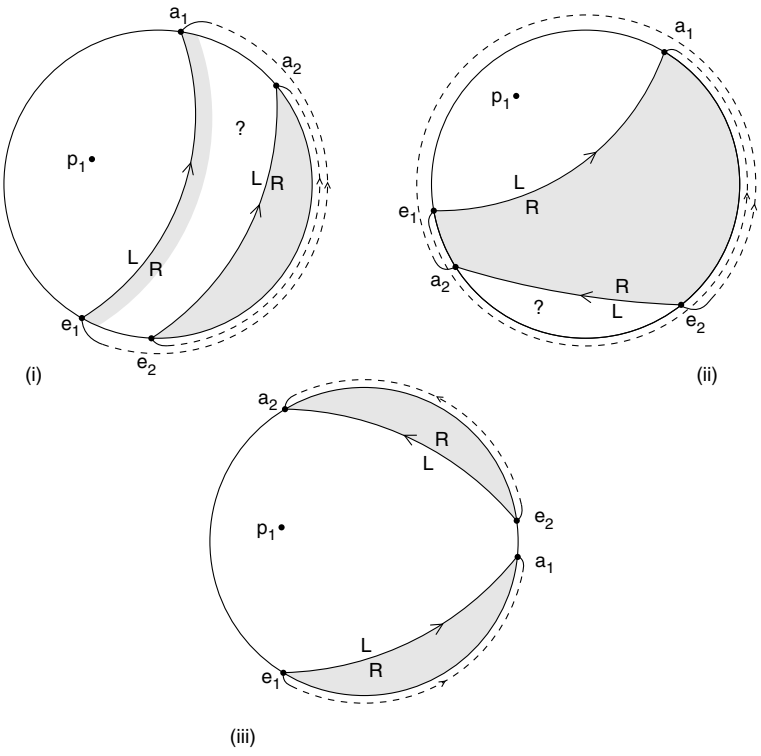


Abb. 6.19 Wenn $B(L,R)$ die Region $VR(p_1,L)$ mehrfach besucht, kommt nur Fall (iii) in Frage. Hier werden bei der Verfolgung von $B(L,R)$ disjunkte Stücke des Randes durchlaufen, die gestrichelt eingezeichnet sind.

Weil dieses Argument für je zwei *beliebige* Besuche von $B(L,R)$ in einer Region $VR(p,L)$ gilt, dürfen wir die Gesamtkosten für das Absuchen der Regionenränder von $V(L)$ nach den Austrittsstellen von $B(L,R)$ durch die Kantenzahl von $V(L)$, also durch $O(n)$, abschätzen.

Damit haben wir auch die zweite Phase erledigt und können folgendes Zwischenergebnis notieren:

Lemma 6.15 *Ist ein Endstück des Bisektors $B(L,R)$ bekannt, so läßt sich der Rest von $B(L,R)$ durch simultane Verfolgung in $V(L)$ und $V(R)$ in Zeit $O(n)$ bestimmen.*

6.4.3 Das Verfahren divide and conquer für $V(S)$

Zusammenfassend können wir unseren Algorithmus folgendermaßen skizzieren:

```
sortiere S nach X- und Y-Koordinaten;
bilde sortierte Verzeichnisse XListe, YListe;
VS := BaueVoro(XListeS, YListeS);
```

Dabei wird die Hauptarbeit durch folgende rekursive Funktion erledigt:

```
function BaueVoro(XListeS, YListeS: tListe): tQEDS;
    if |S| ≤ 2
    then
        konstruiere direkt die QEDS von V(S)
    else (* Zerlegen des Problems *)
        bestimme senkrechte oder waagerechte
                                  Splitgerade G für S;
        if G ist senkrecht
        then
            zerlege S in Teilmengen L, R links und
                                        rechts von G;
            zerlege XListeS in XListeL und XListeR;
            zerlege YListeS in YListeL und YListeR;

            (* Rekursion *)
            VL := BaueVoro(XListeL, YListeL);
            VR := BaueVoro(XListeR, YListeR);

            (* Zusammensetzen der Lösungen *)
            E := UnteresBisektorEndstück(VL, VR);
            B := Bisektor(E, VL, VR);

            VLB := der Teil von VL links von B;
            VRB := der Teil von VR rechts von B;

            BaueVoro := QEDS von VLB ∪ B ∪ VRB;
        if G ist waagerecht (* entsprechend *)
```

Aufgrund unserer Vorarbeiten können wir Laufzeit und Speicherplatzbedarf dieses Algorithmus schnell abschätzen:

Theorem 6.16 *Mit dem Verfahren divide and conquer läßt sich das Voronoi-Diagramm von n Punkten in der Ebene in Zeit $O(n \log n)$ und linearem Speicherplatz konstruieren.*

Beweis. Lemma 6.12 besagt, daß beim rekursiven Aufruf von *BaueVoro* für die Punkte einer Menge $T \subseteq S$ der Teilungsschritt

in Zeit $O(|T|)$ ausgeführt wird und zwei Teilmengen liefert, von denen jede mindestens $\frac{1}{5}|T|$ viele Elemente enthält.

Beim Zusammensetzen brauchen wir nach Lemma 6.14 für die Bestimmung des unteren Endstücks des Bisektors $O(|T|)$ viel Zeit. Der Rest von $B(L, R)$ kann dann nach Lemma 6.15 ebenfalls in Zeit $O(|T|)$ konstruiert werden. Auch das anschließende „Vernähen“ von VL und VR kann in linearer Zeit geschehen.

Im Binärbaum der Rekursionsaufrufe ist für alle Knoten mit demselben festen Abstand zur Wurzel insgesamt $O(n)$ viel Zeit für die anfallenden Arbeiten beim Zerlegen und Zusammensetzen nötig. Der Gesamtaufwand hängt also von der *Höhe* des Rekursionsbaums ab.

Die maximale Höhe würde erreicht, wenn jeweils nur ein Fünftel der vorhandenen Punkte abgespalten wird, so daß vier Fünftel im größeren Teil verbleiben. Weil für $j \geq \log_{\frac{5}{4}} n$

$$\left(\frac{4}{5}\right)^j n \leq 1$$

wird, ist die Höhe durch $\log_{\frac{5}{4}} n \in O(\log n)$ beschränkt, so daß der Zeitaufwand insgesamt in $O(n \log n)$ liegt. □

6.5 Geometrische Transformation

Reduktion von $DT(S)$ auf $ch(S')$ im $\mathbb{R}^3$

Als letzte Variante wollen wir zeigen, wie sich die Konstruktion der Delaunay-Triangulation $DT(S)$ durch eine geschickte geometrische Transformation auf die Berechnung der konvexen Hülle einer Punktmenge im $\mathbb{R}^3$ zurückführen läßt. Die Idee geht zurück auf Edelsbrunner und Seidel [52].

Konvexe Hüllen hatten wir in Kapitel 1 auf Seite 22 für Punktmengen in beliebigen Räumen $\mathbb{R}^d$ definiert. Im $\mathbb{R}^3$ ist die konvexe Hülle $ch(S')$ einer Punktmenge S' ein konvexes Polyeder. Wenn keine vier Punkte aus S' auf einer Ebene liegen, besteht sein Rand aus Dreiecken.

In Übungsaufgabe 4.5 hatten wir gezeigt, daß sich die konvexe Hülle von n Punkten im $\mathbb{R}^3$ in Zeit $O(n \log n)$ berechnen läßt, indem man das randomisierte inkrementelle Verfahren zur Konstruktion zweidimensionaler konvexer Hüllen auf den $\mathbb{R}^3$ überträgt.

Wir betrachten nun das Paraboloid

$$P = \{(x, y, z) \in \mathbb{R}^3; x^2 + y^2 = z\}$$

im $\mathbb{R}^3$, das durch Rotation der Parabel $\{Y^2 = Z\}$ um die Z-Achse entsteht.

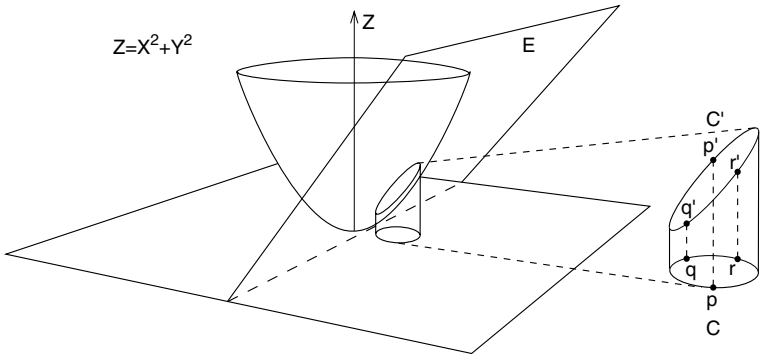


Abb. 6.20 Das Bild von Kreisen auf dem Paraboloid P.

Für jeden Punkt $p = (x, y)$ der XY-Ebene sei $p' = (x, y, x^2 + y^2)$ der Punkt auf dem Paraboloid oberhalb von p; vergleiche Abbildung 6.20. Die Transformation $p \longmapsto p'$ läßt sich auf Punktmengen fortsetzen. Sie hat eine bemerkenswerte Eigenschaft:

Lemma 6.17 *Sei K der Rand eines Kreises in der XY-Ebene. Dann ist die geschlossene Kurve K' auf dem Paraboloid P in einer Ebene des $\mathbb{R}^3$ enthalten.*

Beweis. Angenommen, der Kreisrand wird durch

$$K = \{(x, y) \in \mathbb{R}^2; (x - c)^2 + (y - d)^2 = r^2\}$$

beschrieben, mit Konstanten c, d und r. Durch Ausmultiplizieren der Quadrate und Ersetzen von $x^2 + y^2$ durch z erhält man

$$\begin{aligned} K' &= \{(x, y, z) \in \mathbb{R}^3; (x - c)^2 + (y - d)^2 = r^2 \text{ und } x^2 + y^2 = z\} \\ &= \{(x, y, z) \in \mathbb{R}^3; z - 2cx - 2dy + c^2 + d^2 = r^2\} \cap P; \end{aligned}$$

dies ist offenbar der Durchschnitt einer Ebene des $\mathbb{R}^3$ mit dem Paraboloid P. □

Die ebenso überraschende wie nützliche Konsequenz aus Lemma 6.17 lautet:

Theorem 6.18 *Sei S eine endliche Punktmenge in der XY-Ebene. Dann ist die Delaunay-Triangulation von S gleich der Projektion der unteren*[12] *konvexen Hülle von S' auf die XY-Ebene.*

Delaunay-Triangulation und konvexe Hülle

[12] Damit ist der von der XY-Ebene aus sichtbare Teil von $ch(S')$ gemeint.

Beweis. Seien p, q, r drei Punkte aus S; nach Lemma 5.18 auf Seite 235 bilden sie genau dann ein Delaunay-Dreieck, wenn ihr Umkreis keinen anderen Punkt aus S enthält.

Sei K der Rand des Umkreises von $tria(p, q, r)$; nach Lemma 6.17 ist $K' = E \cap P$ für eine Ebene E im $\mathbb{R}^3$. Offenbar entsprechen die Punkte der XY-Ebene innerhalb von K gerade den Punkten auf dem Paraboloid P unterhalb der Ebene E; siehe Abbildung 6.20. Also sind folgende Aussagen äquivalent:

(1) Kein Punkt aus S liegt im inneren Gebiet von K.

(2) Kein Punkt aus S' liegt unterhalb von E.

Die letzte Eigenschaft besagt aber gerade, daß p', q', r' ein Dreieck auf dem Rand der konvexen Hülle von S' bilden. Damit ist die Behauptung bewiesen. □

Theorem 6.18 eröffnet eine weitere Möglichkeit zur Berechnung der Delaunay-Triangulation in Zeit $O(n \log n)$. Dieser Ansatz kann auf beliebige Dimensionen verallgemeinert werden; siehe z. B. Edelsbrunner [51].

6.6 Verallgemeinerungen

Von den in diesem Abschnitt vorgestellten Algorithmen zur Berechnung von Voronoi-Diagramm oder Delaunay-Triangulation kann man das inkrementelle Verfahren aus Abschnitt 6.2 am leichtesten implementieren. Es läßt sich auch bestens auf andere Metriken und nicht-punktförmige Objekte verallgemeinern. Allerdings sollte man dann das Voronoi-Diagramm und nicht die Delaunay-Triangulation konstruieren und statt des Delaunay-DAG eine Hilfsstruktur verwenden, in der Konflikte von Voronoi-Kanten (anstelle von Delaunay-Dreiecken) gespeichert werden; siehe [87].

randomisierte inkrementelle Konstruktion

sweep

Aber auch das Sweep-Verfahren aus Abschnitt 6.3 läßt sich verallgemeinern: Man kann damit Voronoi-Diagramme von Liniensegmenten berechnen, aber auch von Punkten unter *hübschen Metriken* wie der Karlsruhe-Metrik, erst recht also Voronoi-Diagramme unter beliebigen konvexen Distanzfunktionen; vergleiche dazu Abschnitt 5.5 in Kapitel 5.

divide and conquer

Das *divide-and-conquer*-Verfahren kann ebenfalls verallgemeinert werden, aber nicht so problemlos. Die Schwierigkeit liegt in der Gestalt der Bisektoren $B(L, R)$ von disjunkten Teilmengen L, R von S: Wenn sie – anders als bei der euklidischen Metrik – geschlossene Ketten enthalten, ist nicht klar, wie man für solche Komponenten ein Endstück finden soll.

geometrische Transformation

Die Idee der geometrischen Transformation macht starken Gebrauch von den Eigenschaften der euklidischen Metrik und läßt sich nicht leicht verallgemeinern.

Es gibt aber einen verwandten Ansatz, der ebenfalls auf Edelsbrunner und Seidel [52] zurückgeht und nicht auf die euklidische Metrik beschränkt ist.

Angenommen, zu jedem Punkt $p \in S$ ist eine Abstandsfunktion

$$f_p : \mathbb{R}^2 \longrightarrow \mathbb{R}_{\geq 0}$$

definiert, die jedem Punkt $x = (x_1, x_2)$ der Ebene den Wert $f_p(x)$ als „Abstand zu p“ zuordnet. Für zwei Punkte p, q kann man dann

$$D(p, q) = \{x \in \mathbb{R}^2; f_p(x) < f_q(x)\}$$

setzen und, davon ausgehend, ein Voronoi-Diagramm $V(S)$ definieren.

$V(S)$ als untere Kontur

Für stetige Funktionen f_p ist der Graph

$$G_p = \{(x_1, x_2, f_p(x_1, x_2)) ; (x_1, x_2) \in \mathbb{R}^2\}$$

eine Fläche im $\mathbb{R}^3$. Das Voronoi-Diagramm von S bezüglich der Funktion $(f_p)_{p \in S}$ ist dann nichts anderes als *die Projektion der unteren Kontur* der Flächen $(G_p)_{p \in S}$ auf die XY-Ebene!

Wählt man speziell $f_p(x) = |px|$, entsteht das euklidische Voronoi-Diagramm der Punktmenge S. Die Fläche G_p ist in diesem Fall ein Kegel mit Spitze in p und Innenwinkel 90°, der auf der XY-Ebene senkrecht steht. Man kann solch einen Kegel als Schar konzentrischer Kreise ansehen, die sich von p ausbreiten und dabei die Z-Achse emporsteigen.

Dieser Ansatz eignet sich sehr gut zur Darstellung von Voronoi-Diagrammen am Bildschirm. Man identifiziert die XY-Ebene mit der Bildebene und zeichnet nacheinander mit unterschiedlichen Farben die Flächen G_p auf den Schirm. Für jedes Pixel braucht man sich dann nur die Farbe der Fläche G_p mit der kleinsten Z-Koordinate zu merken!

Diese einfache Technik ist in der Computergraphik unter dem Namen *Z-buffering* bekannt.

Lösungen der Übungsaufgaben

Übungsaufgabe 6.1 Nach Definition ist jede der n Voronoi-Regionen $VR(p, S)$ der Durchschnitt von $n - 1$ Halbebenen $D(p, q), q \in S \setminus \{p\}$. Jeder solche Durchschnitt läßt sich naiv in Zeit $O(n^2)$ berechnen, aber auch in Zeit $O(n \log n)$, wie in Abschnitt 4.1.4 gezeigt worden ist.

Übungsaufgabe 6.2 Nachdem p_i frisch eingefügt ist, besteht sein Stern aus drei Dreiecken, von denen zwei unbeschränkt sein können; hierfür stimmt die Behauptung. Bei jedem nachfolgenden *edge flip* bilden die beiden betroffenen Dreiecke zusammen ein konvexes Viereck, weil p_i im Umkreis des Konfliktdreiecks enthalten ist. Der Stern wird also um einen Strahl zu einer Ecke erweitert, der zwischen zwei vorhandenen Strahlen verläuft. Also ist auch die neue Ecke von p_i aus sichtbar; siehe Abbildung 6.4.

Übungsaufgabe 6.3 Gäbe es ein von p_i weiter entfernt liegendes Dreieck $tria(t, v, w)$, dessen Umkreis p_i enthält, so wäre zum Beispiel $p_i t$ nach Lemma 6.2 eine Delaunay-Kante in DT_i. Sie müßte eine Außenkante des Sterns von p_i kreuzen, die selbst zu DT_i gehört, weil ihr angrenzendes Dreieck konfliktfrei ist. Zwei Kanten einer Triangulation können sich aber nicht kreuzen! Ebensowenig kann eines der neu entstandenen Dreiecke T mit Eckpunkt p_i mit einem Punkt p_h, $h < i$, in Konflikt stehen; denn alle drei Kanten von T gehören zu DT_i.

Übungsaufgabe 6.4

(i) Jedes Delaunay-Dreieck hat genau einen Vater; folglich ist der DAG ein Baum, wenn man nur die Kanten zwischen Vätern und Söhnen zuläßt.

(ii) Wenn ein endliches Dreieck Vater wird, teilt es sich mit jedem Sohn eine andere Kante. Folglich kann es maximal drei Söhne haben, die übrigens derselben Schicht im Delaunay-DAG angehören. Abbildung 6.21 zeigt, daß ein endlicher Vater einen, zwei oder drei Söhne haben kann; der neu hinzukommende Punkt ist jeweils mit p bezeichnet. Ein unendliches Dreieck kann höchstens einen endlichen und zwei unendliche Söhne haben, siehe Abbildung 6.22.

(iii) Ja, siehe etwa $tria(p_1, p_3, p_5)$ in Abbildung 6.9.

Übungsaufgabe 6.5 Die Punkte $p_1, \ldots, p_{\frac{n}{2}}$ werden – wie bisher – von links nach rechts eingefügt; dabei ist jeweils nur $O(1)$ viel Umbauarbeit am Voronoi-Diagramm zu leisten. Jetzt fügt man die Punkte auf der Y-Ache *von unten nach oben* ein, also in der Reihenfolge $p_n, p_{n-1}, \ldots, p_{\frac{n}{2}+1}$! Ab p_{n-1} muß dabei jedesmal nur die

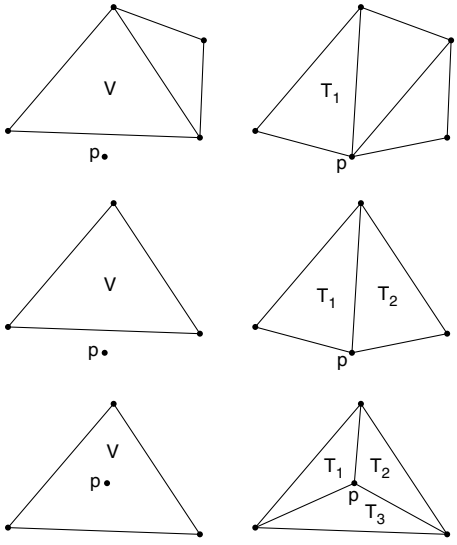


Abb. 6.21 Ein Vaterdreieck V kann einen, zwei oder drei Söhne T_i haben.

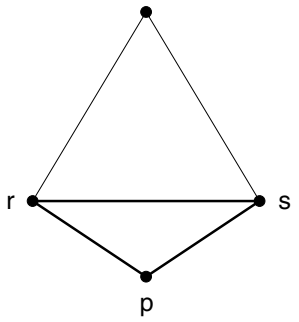


Abb. 6.22 Das unendliches Dreieck $H(r,s)$ hat die Söhne $tria(r,p,s)$, $H(r,p)$ und $H(p,s)$.

Voronoi-Kante des Vorgängers mit $p_{\frac{n}{2}}$ gekürzt werden,[13] was $O(1)$ Zeit erfordert. Insgesamt ist der Aktualisierungsaufwand in $O(n)$.

Übungsaufgabe 6.6 Zunächst einmal ist nicht klar, was geschehen soll, wenn die *sweep line* auf einen neuen Punkt p aus S trifft. Die Voronoi-Region p kann sich ja weit nach links hinter die *sweep line* erstrecken und bereits konstruierte Teile des Voronoi-Diagramms zerstören. Bei näherem Hinsehen zeichnet sich hier ein hartes Problem ab: Wie soll man wissen, daß die *sweep line*

[13] In Abbildung 6.10 ist das die unbeschränkte Kante nach rechts oben.

gerade die Voronoi-Kante zwischen den Regionen von s und r schneidet, wenn die Punkte noch gar nicht entdeckt worden sind? Siehe Abbildung 6.23.

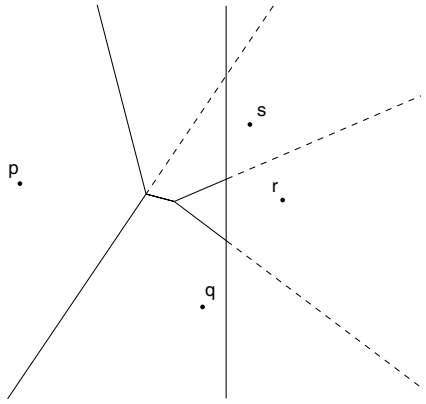


Abb. 6.23 Zu diesem Zeitpunkt „kennt" die *sweep line* nur die Punkte p und q und deren Bisektor. Woher soll sie von der Existenz der übrigen Voronoi-Kanten wissen?

Übungsaufgabe 6.7

(i) Ein einzelner Bisektor $B(p, L)$ kann n Stücke zum Rand der Region von L beitragen, wie Abbildung 6.24 zeigt. Das ist maximal.

Würde nämlich die Welle von p durch andere Bisektoren in mehr als n Stücke zerlegt, müßte es einen anderen Punkt $q \in S$ geben, dessen Welle mindestens zweimal in W vertreten ist, derart daß in W die Reihenfolge $\ldots p \ldots q \ldots p \ldots q \ldots$ vorkommt. Das aber ist wegen (ii) unmöglich.

(ii) Die Wellenfront W ist die *rechte Kontur* der n Parabeln $B(p, L), p \in S$. Je zwei von ihnen können sich nur zweimal schneiden.[14] Die Folge der Beschriftungen der Wellenstücke mit den Namen ihrer zugehörigen Punkte bildet also eine *Davenport-Schinzel-Sequenz* der Ordnung 2 über n Buchstaben und kann nach Übungsaufgabe 2.12 auf Seite 85 höchstens die Länge $2n - 1$ besitzen.

[14] Die Schnittpunkte von $B(p, L)$ und $B(q, L)$ liegen auf der Geraden $B(p, q)$; eine Parabel wird von einer Geraden in höchstens zwei Punkten geschnitten.

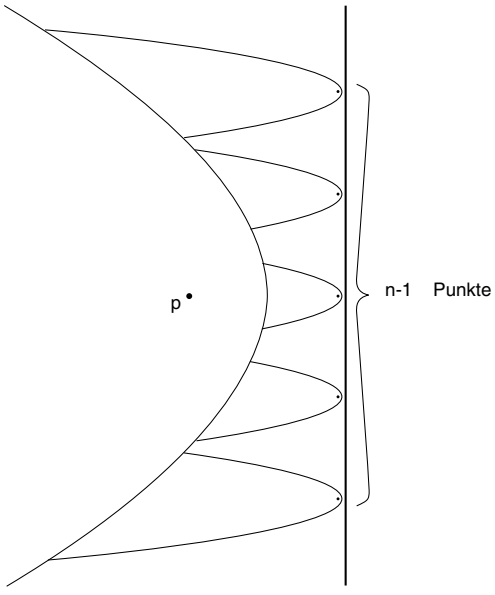


Abb. 6.24 Die Welle von p trägt n Stücke zur Wellenfront bei.

Übungsaufgabe 6.8 Bei einem Punkt-Ereignis ist $Zeit = NeuPkt.X$. Ein Spike-Ereignis für das Wellenstück $AltWStück$ findet zu dem Zeitpunkt statt, in dem die *sweep line* vom $SchnittPkt$ der beiden Spikes genauso weit entfernt ist wie der Punkt, zu dessen Welle das alte Wellenstück gehört. Also muß in diesem Fall gelten

$$Zeit = SchnittPkt.X + |AltWStück.Pkt\ SchnittPkt|.$$

Übungsaufgabe 6.9 Auch hier greift das schon mehrfach verwendete Argument: Wenn man zu einer Teilmenge L von S übergeht, kann für jedes $p \in L$ die Region $VR(p, L)$ höchstens größer werden als $VR(p, S)$, weil sich weniger Konkurrenten um die Ebene streiten. Wenn also in $V(S)$ eine Kante e zwischen den Regionen von $p_1, p_2, \in L$ verläuft, muß es auch in $V(L)$ eine solche Kante in mindestens derselben Länge geben.

Übungsaufgabe 6.10 Jeder Punkt $x \in B(p, q) \cap VR(p, L) \cap VR(q, R)$ hat p als nächsten Nachbarn in L und q als nächsten Nachbarn in R. Außerdem ist er von beiden gleich weit entfernt. Folglich gehört x zu $B(L, R)$.

Übungsaufgabe 6.11 In Abbildung 6.25 besucht $B(L, R)$ die Voronoi-Region von p_2 zweimal.

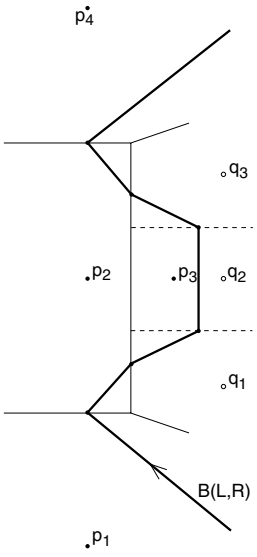


Abb. 6.25 Der Bisektor $B(L, R)$ besucht die Region von p_2 in $V(L)$ zweimal.

7

Bewegungsplanung bei unvollständiger Information

Bis jetzt haben wir uns überwiegend[1] mit Problemen beschäftigt, bei denen die zur Lösung erforderliche Information von Anfang an vollständig vorhanden ist, meist in Form der Eingabedaten.

Im realen Leben ist diese Annahme nicht immer gerechtfertigt. Für jemanden, der sich im Innern eines Labyrinths befindet, bedeutet es einen gewaltigen Unterschied, ob er das Labyrinth kennt oder nicht: Kennt er das Labyrinth, kann er den kürzesten Weg ins Freie *berechnen*, bevor er aufbricht. Ist das Labyrinth jedoch unbekannt, muß er sich aufmachen und nach einem Ausweg *suchen*!

Ausweg aus dem Labyrinth

Die Berechnung des kürzesten Weges ließe sich wie in Abschnitt 5.5.3 auf ein Graphenproblem reduzieren: Man betrachtet den *Sichtbarkeitsgraphen*, der den Startpunkt und alle Hindernisecken als Knoten enthält. Zwei Knoten werden mit einer Kante verbunden, wenn sie sich sehen können. Nun wendet man den Algorithmus von Dijkstra zur Bestimmung aller vom Startpunkt ausgehenden kürzesten Wege in diesem Graphen an; vgl. Güting [69].

Sichtbarkeitsgraph

Offenbar sind hier ganz *unterschiedliche Effizienzmaße* anzulegen, je nachdem, ob der Ausweg berechnet oder gesucht wird: Im ersten Fall kommt es auf die *Rechenzeit* an, die für die Planung des kürzesten Auswegs benötigt wird. Im zweiten Fall ist dagegen die *Länge des Weges* entscheidend, den man insgesamt zurücklegt, um ins Freie zu gelangen. Dieser Weg kommt durch Versuch und Irrtum zustande und wird in der Regel sehr viel länger sein als der kürzeste Ausweg!

unterschiedliche Effizienzmaße

Außerdem ist gar nicht klar, ob überhaupt eine Suchstrategie existiert, mit deren Hilfe man immer einen Ausweg findet, wenn

[1] Mit zwei Ausnahmen: dem Sweep-Verfahren zur Berechnung der maximalen Teilsumme in Übungsaufgabe 2.2 auf Seite 56 und der inkrementellen Konstruktion der Delaunay-Triangulation in Abschnitt 6.2.1.

es einen gibt. Mit dieser Frage beschäftigen wir uns im folgenden Abschnitt.

Anwendung: autonome Roboter

Daß man sich für derartige Probleme interessiert, hat ganz praktische Gründe: Wenn ein mobiler autonomer Roboter in einer ihm zunächst nicht bekannten Umgebung eingesetzt wird, steht er vor ganz ähnlichen Problemen wie ein Mensch in einem Labyrinth. Der Auftrag des Roboters kann z. B. darin bestehen, die Umgebung zu erkunden und zu kartographieren; oder er soll sich von seiner augenblicklichen Position auf möglichst kurzem Weg zu einer Zielposition begeben, um dort eine Arbeit auszuführen. In beiden Fällen macht ihm der Mangel an Information zu schaffen.

Definition unseres Modells

Die Umgebung des Roboters – das Labyrinth – wird durch eine endliche Menge von einfachen, geschlossenen polygonalen Ketten beschrieben. Je zwei von ihnen sollen sich nicht schneiden. Diese Ketten bilden die Ränder der Hindernisse. Sie werden im folgenden auch als *Wände* bezeichnet.

Die Hindernisse brauchen nicht aus massivem Mauerwerk zu bestehen. Sie können vielmehr Innenhöfe enthalten wie in Abbildung 7.1 (i). Dort und in den folgenden Abbildungen ist das Mauerwerk hellgrau dargestellt und der freie Raum weiß. Der Startpunkt des Roboters kann in einem solchen Innenhof liegen, aber nicht im Innern des Mauerwerks. Ein Innenhof kann selbst wieder Hindernisse enthalten, und so fort.

Wo Mauerwerk ist und wo freier Raum, wird folgendermaßen festgelegt: Denkt man sich alle Ketten aus der Ebene ausgeschnitten, bleiben endlich viele offene Gebiete übrig. Ganz außen liegt ein unbeschränktes Gebiet, das von keiner Kette umschlossen wird; es besteht aus freiem Raum und wird weiß gefärbt. Die übrigen Gebiete werden nun abwechselnd weiß und grau gefärbt, so daß niemals zwei Gebiete mit derselben Farbe aneinandergrenzen.

Übungsaufgabe 7.1 Warum läßt sich diese Färbung in der angegebenen Weise durchführen?

Natürlich gibt es für den Roboter nur dann einen Ausweg aus dem Labyrinth, wenn sein Startpunkt von keiner Kette umschlossen wird, also nicht in einem Innenhof liegt, sondern in dem unbeschränkten Gebiet oder seinem Abschluß.

punktförmiger Roboter

Der Roboter ist gegenüber seiner Umgebung so klein, daß wir ihn uns als einen Punkt vorstellen. Er paßt daher durch jede Lücke zwischen den Wänden hindurch.

keine Einschränkung

Liegt in dieser Annahme eine unzulässige Vereinfachung? Nein! Das Problem eines kreisförmigen Roboters mit Radius r läßt sich nämlich auf unser Modell reduzieren – durch Vergrößerung der Hindernisse!

Dazu fährt man mit dem Mittelpunkt eines Kreises vom Radius r an den Rändern der Hindernisse entlang.[2] Wo sich vergrößerte Hindernisse überlappen, faßt man sie zusammen. Ein Beispiel ist in Abbildung 7.1 (ii) gezeigt. Im Originallabyrinth gibt es genau dann einen Ausweg für den kreisförmigen Roboter mit Mittelpunkt in Startposition s, wenn es im Labyrinth der vergrößerten Hindernisse einen Ausweg für einen Punkt mit Startposition s gibt.

Diese schöne Idee geht auf Lozano-Pérez [97] zurück und läßt sich auch auf nicht-kreisförmige Roboter anwenden, solange nur Translationen, aber keine Rotationen erlaubt sind.

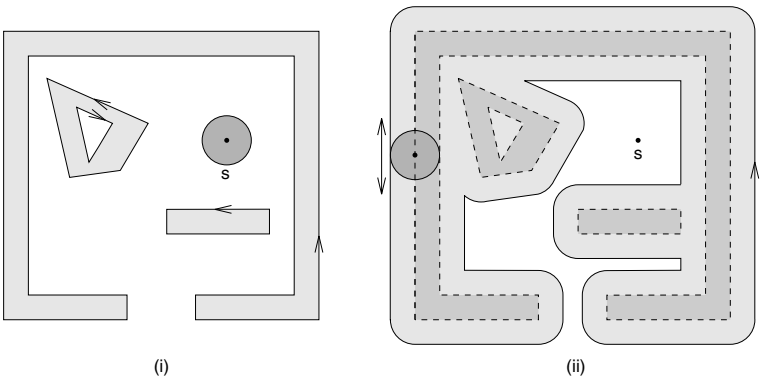


Abb. 7.1 Genau dann läßt sich links der kreisförmige Roboter von Startposition s nach außen bewegen, wenn das rechts für einen Punkt möglich ist.

Sensoren

Wichtig ist, mit welchen *Sensoren* der Roboter seine Umwelt wahrnimmt. Im einfachsten Fall ist nur ein Tastsensor vorhanden, der den Kontakt mit einer Hinderniswand feststellt und es dem Roboter erlaubt, sich an der Wand entlangzubewegen. Wesentlich komfortabler ist eine Rundumsicht, z. B. durch ein rotierendes Lasersystem, das zu jedem Zeitpunkt das Sichtbarkeitspolygon der aktuellen Position liefert, also alle zur Zeit sichtbaren Teile der Wände und ihre jeweiligen Entfernungen.

Obwohl unser Roboter punktförmig ist, stellen wir uns vor, daß er über eine Art Nase verfügt und sich nur „der Nase nach“

[2] Nimmt man an, daß der Mittelpunkt des Roboters R im Nullpunkt liegt, so entstehen allgemein die vergrößerten Hindernisse $\hat{H}_i$ als *Minkowski-Summen* $\hat{H}_i = H_i \ominus R = \{\, h - r \;;\; h \in H_i \text{ und } r \in R \,\}$; siehe das Applet `http://www.geometrylab.de/MinkowskiSum/` . Ihre Ränder sind strenggenommen keine polygonalen Ketten mehr, weil an den äußeren Ecken Kreisbogenstücke entstehen.

Orientierung

vorwärtsbewegen kann. Um die Marschrichtung zu ändern, ist also vorher eine Drehbewegung nötig, die wir uns stetig ausgeführt vorstellen. Dieser Ansatz entspricht dem Modell der *turtle*, das schon in Abschnitt 4.2 auf Seite 178 erwähnt wurde.

Weitere Unterschiede zwischen verschiedenen Robotermodellen liegen in der Fähigkeit, über die ausgeführten Bewegungen Buch zu führen bzw. den eigenen Standort zu bestimmen.[3] Ein einfaches System kann vielleicht nur die Drehbewegungen erfassen, ein komfortableres könnte etwa zu jedem Zeitpunkt seine absoluten Koordinaten kennen und wäre damit imstande, auch die Länge der zurückgelegten Wegstücke zu berechnen.

Welche „Ausstattung" wir bei dem Roboter voraussetzen, wird im folgenden jeweils angegeben. Die Rechenleistung soll aber in keinem Fall eingeschränkt sein: Jeder Roboter verfügt über einen Prozessor und ausreichend viel Speicherplatz.

7.1 Ausweg aus einem Labyrinth

In diesem Abschnitt betrachten wir ein Problem, das schon in der Antike bekannt war. In unserer Terminologie können wir es folgendermaßen formulieren.

Gegeben sei ein punktförmiger Roboter, der nur über einen Tastsensor und einen Winkelzähler verfügt.[4]

Dieser Zähler kann auf 0 zurückgesetzt werden und gibt danach zu jedem Zeitpunkt die Summe der Winkel der insgesamt ausgeführten Drehbewegungen an. Dabei werden Linksdrehungen positiv gezählt und Rechtsdrehungen negativ; vergleiche auch Seite 178. Gesucht ist eine Strategie, mit deren Hilfe solch ein Roboter aus jedem unbekannten Labyrinth herausfindet, aus dem es überhaupt einen Ausweg gibt.

Problemstellung

Dabei verstehen wir unter *herausfinden*, daß der Roboter den Rand der konvexen Hülle der Hindernisse erreicht oder gleich im Startpunkt stehenbleibt, wenn dieser schon außerhalb der konvexen Hülle liegt. Wir nehmen an, daß zu diesem Zeitpunkt eine Statusvariable *Entkommen* gesetzt wird und die Schleifen in den folgenden Programmen verlassen werden.[5]

Abbruchbedingung

[3] Auch reale Roboter haben technische Schwierigkeiten, diese Probleme *exakt* zu lösen.

[4] Das bedeutet für den antiken Helden: Er wacht im Innern des Labyrinths auf und verfügt über keinerlei Hilfsmittel wie Ariadnefäden oder ein Werkzeug zum Anbringen von Markierungen. Außerdem ist es so dunkel, daß er sich allein auf seinen Tastsinn verlassen muß.

[5] Ohne ein Signal von außen kann der Roboter nicht wissen, wann er aus dem Labyrinth entkommen ist.

Wenn der Roboter auf ein Hindernis trifft, muß er sich entscheiden, in welcher Richtung er der Wand folgen will. Wir legen fest, daß der Roboter sich rechtsherum dreht und dann so an der Wand entlangläuft, daß das Mauerwerk sich links von ihm befindet. In Abbildung 7.1 (i) sind die Umlaufrichtungen der Wände eingezeichnet, die sich durch diese Festlegung ergeben.

linke Hand an der Wand

Ein erster Ansatz für die gesuchte Strategie könnte nun folgendermaßen aussehen:

1. Versuch: an der Wand entlang

```
wähle Richtung beliebig;
repeat
    folge Richtung
until Wandkontakt;
repeat
    folge der Wand
until Entkommen
```

Hierbei bezeichnet die Variable *Richtung* die momentane Orientierung des Roboters, also die Richtung seiner Nase.

In Abbildung 7.2 (i) ist ein höhlenartiges Labyrinth zu sehen, aus dem diese Strategie den eingezeichneten Ausweg findet. Wenn aber das Höhleninnere nicht einfach-zusammenhängend ist, kann es zu Endlosschleifen kommen, wie das Beispiel (ii) zeigt.

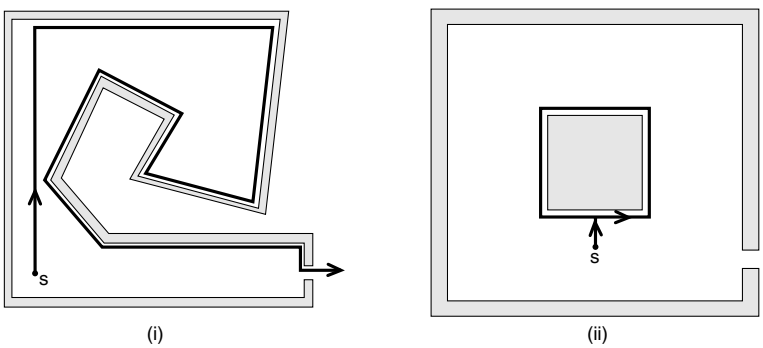


Abb. 7.2 Durch Verfolgen der Wand findet der Roboter aus dem linken Labyrinth heraus, während er rechts in eine Endlosschleife gerät.

Es ist also offenbar wichtig, sich von unterwegs angetroffenen Hindernissen auch wieder zu lösen! Dies versucht unser zweiter Ansatz. Der Roboter wählt eine Anfangsrichtung aus und folgt ihr. Trifft er auf ein Hindernis, folgt er der Wand nur so lange, bis seine Nase wieder in die Anfangsrichtung zeigt. Dort löst er sich vom Hindernis und iteriert das Verfahren.

2. Versuch: möglichst in Anfangsrichtung

wähle *Richtung* beliebig;
Winkelzähler := 0;
repeat
    repeat
        folge *Richtung*
    until *Wandkontakt*;
    repeat
        folge der Wand
    until *Winkelzähler* **mod** $2\pi = 0$
until *Entkommen*

Abbildung 7.3 zeigt, wie sich diese Strategie in den beiden Beispiellabyrinthen verhält: In (ii) löst sich der Roboter schön von dem freistehenden Hindernis und findet den Ausgang, aber nun gerät er in (i) in eine Endlosschleife.

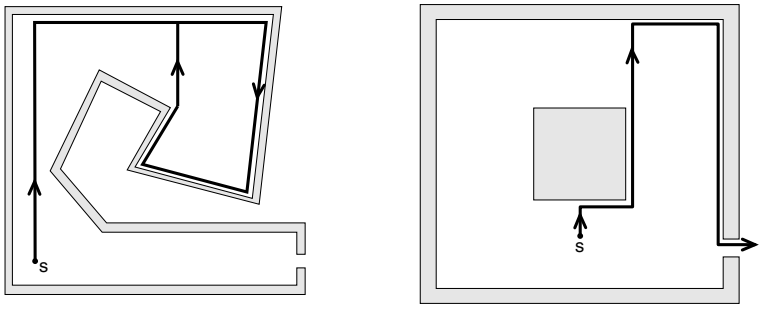


Abb. 7.3 Läuft man in Anfangsrichtung, wann immer die Nase dorthin zeigt, kann man ebenfalls in eine Endlosschleife geraten.

Wie lassen sich Endlosschleifen erkennen und unterbrechen? Intuitiv dreht sich der Roboter bei jedem Umlauf einmal um seine Achse. Das läßt sich mit Hilfe des Winkelzählers feststellen! Dann reicht es aber nicht aus, nur die momentane Richtung zu beachten, also den Wert des Winkelzählers modulo 2π. Wir benötigen nun tatsächlich den Gesamtdrehwinkel.

Unser dritter Versuch unterscheidet sich nur in diesem Detail vom zweiten.

3. Versuch: Pledge-Algorithmus

```
wähle Richtung beliebig;
Winkelzähler := 0;
repeat
    repeat
        folge Richtung
    until Wandkontakt;
    repeat
        folge der Wand
    until Winkelzähler = 0
until Entkommen
```

Dieses Verfahren wird *Pledge-Algorithmus* genannt, nach einem Jungen namens John Pledge, der es [1] zufolge im Alter von zwölf Jahren erfunden hat.

Man beachte, daß auch hier der eingeschlagene Weg aus zwei Sorten von Segmenten besteht: Während der Winkelzähler einen von null verschiedenen Wert hat, führt der Weg an einer Hinderniswand entlang. Ist der Wert des Winkelzählers gleich null, bewegt sich der Roboter frei[6] in die anfangs gewählte Richtung.

freie Wegstücke

Man sieht sofort, daß diese Strategie in beiden Beispielen das Richtige tut: sie läuft die Wege in Abbildung 7.2 (i) und 7.3 (ii). Sie findet also in beiden Fällen ins Freie. Das ist kein Zufall! Es gilt nämlich folgender Satz:

Theorem 7.1 *Der Pledge-Algorithmus findet in jedem Labyrinth von jeder Startposition aus einen Weg ins Freie, von der überhaupt ein Ausweg existiert.*

Beweis. Sei L ein Labyrinth und s eine Startposition in L. Zunächst beweisen wir folgende Aussage über die möglichen Werte des Winkelzählers:

Lemma 7.2 *Der Winkelzähler nimmt niemals einen positiven Wert an.*

Beweis. Am Anfang wird der Zähler auf null gesetzt, und sein Wert ist auch später immer gleich null, wenn der Roboter „freie" Wegstücke in Anfangsrichtung zurücklegt.

Wenn der Roboter auf eine Hinderniswand trifft, führt er zunächst eine Rechtsdrehung aus, um der Wand folgen zu können,

[6] Wir nennen auch solche Wegstücke „frei", die einer in der Anfangsrichtung orientierten Hinderniskante folgen, vorausgesetzt, der Winkelzähler hat den Wert Null.

siehe Abbildung 7.4. Dadurch erhält der Zähler einen negativen Wert. Wenn der Wert des Winkelzählers jemals null wird, löst sich der Roboter wieder vom Hindernis. Aus Stetigkeitsgründen können deshalb keine positiven Werte auftreten. □

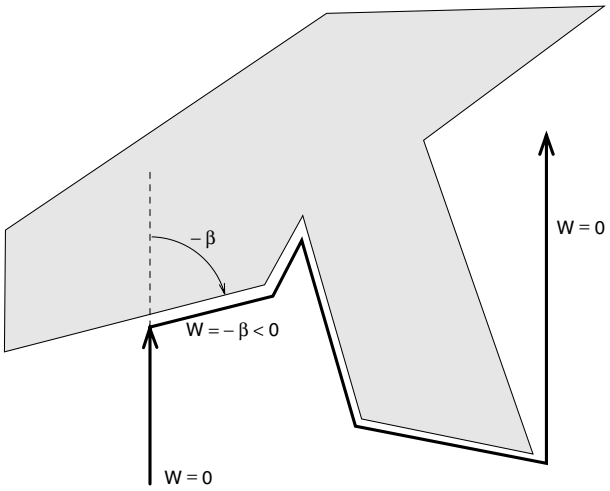


Abb. 7.4 Bis der Roboter das Hindernis wieder verläßt, hat sein Winkelzähler stets einen negativen Wert.

Angenommen, der Roboter findet vom Startpunkt s aus nicht ins Freie. Wir müssen zeigen, daß dann auch kein Entkommen möglich ist, weil s in einem Innenhof liegt. Zunächst betrachten wir die Struktur des unendlichen Weges, den der Roboter in diesem Fall zurücklegt.

Endlosschleife

Lemma 7.3 *Angenommen, der Roboter findet nicht aus dem Labyrinth heraus. Dann besteht sein Weg bis auf ein endliches Anfangsstück aus einem geschlossenen Weg, der immer wieder durchlaufen wird.*

Beweis. Der Weg des Roboters ist eine polygonale Kette, deren Eckpunkte aus einer endlichen Menge stammen: Es kommen nur Ecken von Wänden im Labyrinth in Frage und von jeder Wandecke aus der in Ausgangsrichtung erste Punkt auf dem nächsten Hindernis.

Falls der Roboter einen solchen Eckpunkt zweimal mit demselben Zählerstand besucht, wiederholt er den Weg dazwischen zyklisch immer wieder, weil sein Verhalten determiniert ist, und es folgt die Behauptung des Lemmas.

Wenn aber jeder Eckpunkt höchstens einmal mit demselben Zählerstand besucht wird, erreicht der Roboter nur endlich oft

Eckpunkte mit Zählerstand null. Sobald diese Besuche erfolgt sind, kann der Roboter nie wieder eine freie Bewegung ausführen; er folgt also danach nur noch einer einzigen Wand, und sein Weg wird auch in diesem Fall zyklisch. □

Sei P der geschlossene Weg, den der Roboter bei seinem vergeblichen Versuch, aus dem Labyrinth zu entkommen, laut Lemma 7.3 immer wieder durchläuft. Wir müssen zeigen, daß kein Ausweg existiert.

endloser Weg ist einfach

Lemma 7.4 *Der Weg P kann sich nicht selbst kreuzen.*

Beweis. Andernfalls müßte es ein Segment B von P geben, das in einem Punkt z auf ein anderes Segment A auftrifft. Eines der beiden Segmente – sagen wir B – muß frei sein, weil die Wände sich nicht schneiden; siehe Abbildung 7.5.

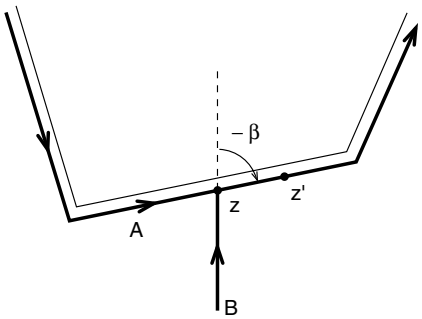


Abb. 7.5 Im Punkt z trifft Segment B auf Segment A.

Seien $W_A(z')$ und $W_B(z')$ die Zählerstände an einem Punkt z' kurz hinter dem Punkt z bei Anreise über A bzw. über B. Dann ist

$$\begin{aligned} W_B(z') &= -\beta && \text{mit} \quad 0 \le \beta < \pi \\ W_A(z') &= -\beta + k2\pi && \text{mit} \quad k \in \mathbb{Z}, \end{aligned}$$

denn hinter z zeigt die Nase des Roboters stets in dieselbe Richtung.[7] Wäre $k \ge 1$, so ergäbe sich

$$W_A(z') = -\beta + k2\pi > -\pi + 2\pi = \pi$$

im Widerspruch zu Lemma 7.2. Also ist $k \le 0$.

Aus $k = 0$ würde $W_A(z') = W_B(z')$ folgen. In diesem Fall würden sich die Wegstücke über A und B hinter z niemals wieder trennen. Wenn also nach z' als nächstes etwa das Segment B besucht würde, käme das Segment A niemals wieder an die Reihe.

[7] Der Fall $\beta = 0$ tritt nur ein, wenn z ein Eckpunkt ist, ab dem der Weg in Anfangsrichtung weiterläuft.

Das steht im Widerspruch zu der Tatsache, daß sowohl A als auch B Teile von P sind und P unendlich oft durchlaufen wird!

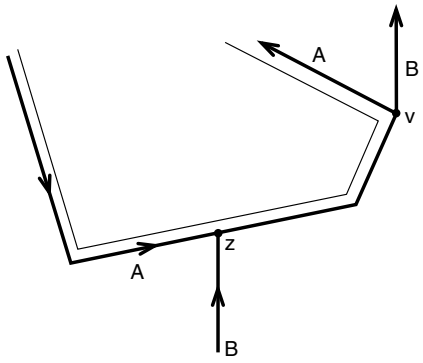


Abb. 7.6 Die Wegstücke A und B kreuzen sich hier nicht.

Also bleibt nur der Fall $k \leq -1$ übrig. Dann ist $W_A(t) < W_B(t)$ für alle Punkte t ab z' bis zu dem Punkt v, an dem sich die Wege wieder trennen; dort muß dann $W_B(v) = 0$ sein. Also sieht die Fortsetzung der Wegstücke im Prinzip so aus wie Abbildung 7.6 zeigt. Es liegt daher keine echte Kreuzung vor, sondern nur eine Berührung. □

Um den Beweis von Theorem 7.1 zu Ende zu führen, betrachten wir zwei Fälle: Angenommen, der Roboter durchliefe den Weg P *gegen* den Uhrzeiger. Dann würde sich nach Lemma 4.17 auf Seite 178 der Winkelzähler bei jedem Durchlauf um 2π *erhöhen*, denn P ist zwar kein einfacher Weg, läßt sich aber leicht zu einem einfachen Weg deformieren, ohne die Winkel zu ändern. Während eines jeden Durchlaufs führt der Zählerstand dieselben Auf- und Abwärtsschwankungen aus; wenn der Wert aber insgesamt bei jeder Runde um 2π zunähme, müßte er irgendwann positiv werden, was wegen Lemma 7.2 unmöglich ist.

Also durchläuft der Roboter den Weg P *im* Uhrzeigersinn, und der Wert des Winkelzählers nimmt dabei jedesmal um 2π *ab*. Dann können irgendwann nur noch echt negative Werte angenommen werden. Deshalb kann der Weg P keine freien Segmente enthalten, er folgt also einer Hinderniswand. Wegen der Umlaufrichtung im Uhrzeigersinn kann es sich dabei nur um die Wand eines Innenhofs handeln, in dem der Roboter gefangen ist; siehe Abbildung 7.7.

Damit ist Theorem 7.1 bewiesen. □

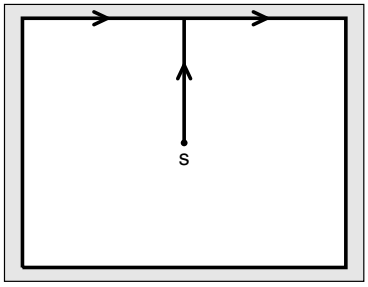


Abb. 7.7 Von dieser Startposition aus gibt es kein Entrinnen!

Dieser Beweis geht auf Abelson und diSessa [1] zurück. Der Pledge-Algorithmus wurde davon unabhängig auch von Asser entdeckt und verifiziert; siehe Hemmerling [72]. Im Applet

```
http://www.geometrylab.de/Polyrobot/
```

ist der Pledge-Algorithmus implementiert.

Werfen wir zum Schluß noch einen Blick auf die *Effizienz* des Pledge-Verfahrens. Wie das Beispiel in Abbildung 7.8 zeigt, kann der von dieser Strategie gefundene Ausweg aus dem Labyrinth sehr viel länger sein als der kürzeste Weg ins Freie. Es kommt sogar vor, daß Segmente mehrfach besucht werden.

Effizienz

7.2 Finden eines Zielpunkts in unbekannter Umgebung

In diesem Abschnitt betrachten wir das Problem, in einer unbekannten Umgebung einen Zielpunkt zu finden. Dazu statten wir unseren Roboter ein wenig besser aus: Wir nehmen an, daß er zu jedem Zeitpunkt seine eigenen Koordinaten[8] kennt. Außerdem sind ihm die Koordinaten des Zielpunkts bekannt. Damit kann der Roboter Entfernungen und die Länge zurückgelegter Wegstücke berechnen. Darüber hinaus kann er sich einen einmal besuchten Punkt merken und feststellen, wenn er ihn wieder erreicht.

Robotermodell

Im übrigen nehmen wir weiterhin an, daß der Roboter über einen Tastsensor verfügt, mit dessen Hilfe er an Hinderniswänden entlanglaufen kann. Diese werden nach wie vor durch geschlossene polygonale Ketten gebildet.

Mit dieser Ausstattung läßt sich ein Zielpunkt in unbekannter Umgebung leicht finden. Lumelsky und Stepanov [98] haben dazu folgende Strategie *Bug* (engl. für Wanze) vorgeschlagen.

Strategie *Bug*

[8] Das können Weltkoordinaten sein, die z. B. über Satellitenpeilung bestimmt werden, oder etwa die lokalen Koordinaten einer Fabrikhalle.

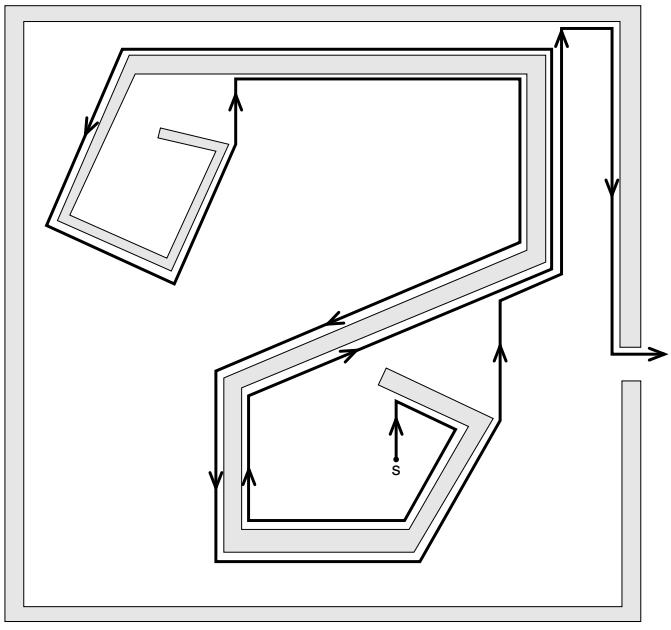


Abb. 7.8 Der Pledge-Algorithmus findet immer einen Ausweg, aber nicht unbedingt den kürzesten.

Der Roboter läuft solange auf den Zielpunkt zu, bis er auf ein Hindernis trifft. Dieses wird einmal vollständig umrundet. Dabei merkt sich der Roboter denjenigen[9] Punkt auf dem Rand des Hindernisses, der dem Zielpunkt am nächsten ist, und kehrt nach der vollständigen Umrundung dorthin zurück.

Von diesem Punkt aus wird der Weg in gleicher Weise fortgesetzt.

repeat
  repeat
    laufe auf *Zielpunkt* zu
  until *Wandkontakt*;
  A := *AktuellePosition*; (* auf Hinderniswand *)
  D := *AktuellePosition*; (* zum Zielpunkt nächster bisher
                besuchter Punkt auf Hinderniswand *)
  repeat
    rücke *AktuellePosition* entlang der Wand vor;
    if *AktuellePosition* näher an *Zielpunkt* als D
      then D := *Aktuelle Position*

[9] Wenn es mehrere zum Zielpunkt nächste Punkte auf dem Hindernisrand gibt, wählt der Roboter einen von ihnen aus.

until *AktuellePosition* = A;

gehe auf kürzestem Wege längs Hinderniswand zu D

until *ZielpunktErreicht*

Hierbei nehmen wir wieder an, daß bei Erreichen des Zielpunkts die inneren Schleifen automatisch verlassen werden.

Das Vorrücken der aktuellen Position in der letzten **repeat**-Schleife haben wir hier als stetigen Prozeß aufgefaßt; in einer polygonalen Umgebung kann der Roboter gleich eine ganze Hinderniskante vorrücken und dann den Punkt D entsprechend aktualisieren. Abbildung 7.9 zeigt ein Beispiel für die Arbeitsweise der Strategie *Bug*. Der Roboter trifft auf seinem Weg zum Zielpunkt t insgesamt auf drei Hindernisse. Die Punkte, in denen er auf Hindernisse trifft, sind mit A_i bezeichnet. In den Punkten D_i löst er sich von den Hindernissen. Wir machen uns zunächst klar, daß dieser Ansatz korrekt funktioniert.

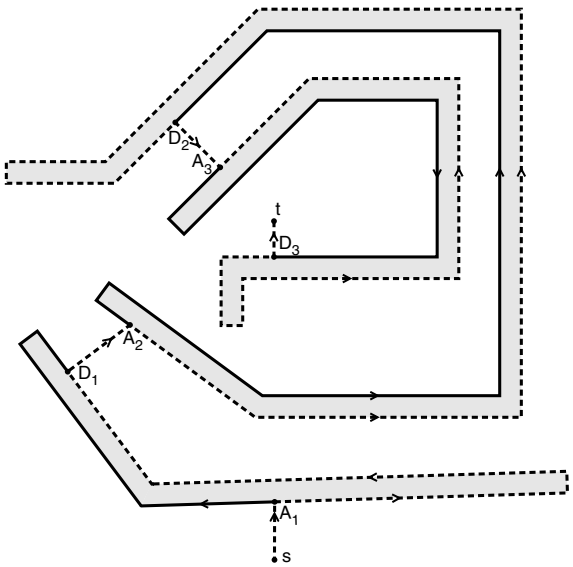


Abb. 7.9 So führt die Strategie *Bug* den Roboter vom Startpunkt s zum Zielpunkt t. Die nur einmal besuchten Wandstücke sind gestrichelt eingezeichnet, die zweimal besuchten mit durchgezogenen Linien. Die Pfeile geben die Laufrichtung des Roboters an.

Theorem 7.5 *Die Strategie Bug findet stets einen Weg vom Startpunkt s zum Zielpunkt t, wenn solch ein Weg überhaupt existiert.* Korrektheit

Beweis. Nach Voraussetzung liegt der Startpunkt s nicht im Mauerwerk, und es gibt einen Weg von s nach t.

Sei $A_1, A_2, \ldots$ die Folge der Punkte, in denen der Roboter jeweils auf eine Hinderniswand trifft. Für jedes A_i sei D_i der zum Zielpunkt t nächstgelegene Punkt auf der jeweiligen Hinderniswand, von dem aus der Roboter seine Reise fortsetzt, bis er in A_{i+1} wieder auf eine Wand trifft. Offenbar gilt

$$|A_i t| \geq |D_i t| \geq |A_{i+1} t| \quad \text{für} \quad i = 1, 2, \ldots$$

Wir wollen uns jetzt überlegen, warum in jedem Punkt D_i der Weg in Richtung t frei ist, so daß der Roboter sich dort tatsächlich von der Wand lösen und wenigstens ein Stück weit auf den Zielpunkt zulaufen kann, bevor er wieder auf ein Hindernis trifft.

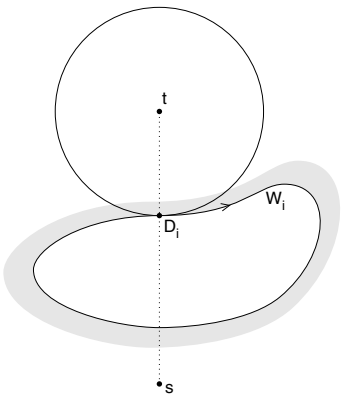


Abb. 7.10 Wie soll der Roboter zum Punkt D_i gelangt sein?

am Punkt D ist der Weg frei

Angenommen, das Mauerwerk des aktuellen Hindernisses läge – von s nach t gesehen – *hinter* dem Punkt D_i; siehe Abbildung 7.10. Weil D_i ein zu t nächster Punkt auf der aktuellen Wand W_i ist, kann W_i den Kreis um t, der durch D_i läuft, nicht betreten. Andererseits dürfen s und t von W_i nicht separiert werden, denn sonst würde ja kein Weg vom Start zum Ziel existieren.

Also kommt nur ein Verlauf von W_i in Betracht, wie er in Abbildung 7.10 dargestellt ist. Dann hätte aber der Roboter irgendwo die Wand W_i durchbrechen müssen, um von s nach D_i zu gelangen – ein Widerspruch!

Damit haben wir gezeigt, daß stets die echte Ungleichung

$$|D_i t| > |A_{i+1} t|$$

gilt, also auch

$$|D_i t| > |D_{i+1} t| \quad \text{für} \quad i = 1, 2, \ldots$$

Dann können aber keine zwei Punkte D_i, D_{i+j} auf demselben Hindernis liegen, d. h. jedes Hindernis wird höchstens einmal besucht.

Weil es nur endlich viele Hindernisse gibt, ist die Folge der Punkte $A_1, D_1, A_2, D_2, \ldots$ demnach endlich. Daher gelangt der Roboter vom letzten Punkt D_m aus entweder direkt ans Ziel, oder er trifft in A_{m+1} auf das $(m+1)$-te Hindernis und findet bei dessen Umrundung den Zielpunkt t. □

Im Unterschied zu dem in Abschnitt 7.1 vorgestellten Pledge-Algorithmus ist für die Strategie *Bug* folgende obere Schranke für die Länge des Weges bekannt, den der Roboter auf der Suche nach dem Zielpunkt zurücklegt:

Effizienz

Theorem 7.6 *Wenn ein nach der Strategie Bug vorgehender Roboter vom Startpunkt s aus den Zielpunkt t erreicht hat, so ist sein zurückgelegter Weg nicht länger als*

$$|st| + \frac{3}{2}\sum_{i=1}^{n} U_i,$$

wobei $U_1, \ldots, U_n$ die Längen derjenigen Wände bezeichnen, auf denen es Punkte gibt, die näher am Zielpunkt t liegen als s.

Beweis. Alle vom Roboter zurückgelegten Strecken von D_i nach A_{i+1} sind insgesamt nicht länger als das Liniensegment st, wie folgende Übungsaufgabe zeigt.

Übungsaufgabe 7.2 Man zeige, daß die Summe der Entfernungen $|sA_1|, |D_1A_2|, |D_2A_3| \ldots$ durch $|st|$ beschränkt ist.

Offenbar trifft der Roboter nur auf Wände mit der im Theorem genannten Eigenschaft, denn für die Auftreffpunkte A_i gilt ja

$$|st| \geq |A_1t| > |A_2t| > \ldots > |A_mt|.$$

Jede getroffene Wand wird einmal vollständig umrundet und anschließend noch einmal höchstens zur Hälfte, wenn der Roboter den Punkt D_i aufsucht. Das ergibt insgesamt den Faktor 3/2. □

So nützlich diese Schranke auch ist: Der vom Roboter zurückgelegte Weg kann beliebig viel länger sein als der kürzeste Weg von s nach t, wie Abbildung 7.11 zeigt. Davon kann man sich auch mit Hilfe des Applets

`http://www.geometrylab.de/Polyrobot/`

überzeugen. In den folgenden Abschnitten werden wir Strategien kennenlernen, bei denen das nicht passieren kann.

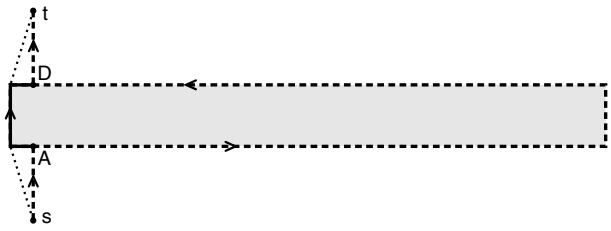


Abb. 7.11 Hier ist der von Strategie *Bug* zurückgelegte Weg beliebig viel länger als der kürzeste Weg von s nach t.

Zuvor sei erwähnt, daß die Kenntnis der Koordinaten keine notwendige Voraussetzung dafür ist, daß ein Roboter seinen Zielpunkt finden kann. Im Prinzip genügt es, wenn er außer dem Tastsensor einen *Zielkompaß* besitzt, der ihm in jeder Position die Richtung zum Zielpunkt weist. Diese theoretisch interessante Tatsache geht auf Hemmerling [73] zurück.

Zielkompaß genügt

Theorem 7.7 *Im Prinzip genügen Zielkompaß und Tastsensor, um in unbekannter Umgebung einen Zielpunkt zu finden.*

Beweis. Wir können die möglichen Bewegungen des Roboters durch folgende drei *Grundbefehle* steuern:

Grundbefehle

T: Laufe von einer Wandecke geradlinig auf den Zielpunkt zu, bis dieser oder eine Wand erreicht ist.

L: Laufe von einem Wandpunkt in gewohnter Richtung an der Wand entlang, bis die nächste Ecke erreicht ist.

R: Wie L, aber in entgegengesetzter Richtung.

Man beachte, daß L und R hier *keine turtle*-Drehungen bezeichnen, sondern sie geben an, ob der Roboter sich „mit der linken oder der rechten Hand“ an der Wand entlangtastet.

Jeder Weg, der nur aus diesen Grundtypen besteht, läßt sich durch ein endliches *Steuerwort* über dem Alphabet $\Sigma = \{T, L, R\}$ darstellen: Abbildung 7.12 zeigt ein Beispiel.

Steuerwort

Wir nehmen an, daß der Zielpunkt t vom Startpunkt s aus erreichbar ist. Wenn wir also die XY-Ebene längs der Wände aufschneiden, liegen s und t im Abschluß A des unbeschränkten Gebiets.

Wenn der Roboter in s startet, kann er nur zu anderen Punkten in A gelangen. Hält er sich dabei an ein Steuerwort über Σ, kommen sogar nur endlich viele Punkte in Frage: Startpunkt s, Hindernisecken und Endpunkte von freien T-Bewegungen, die in s oder in Wandecken starten, sowie der Zielpunkt t.

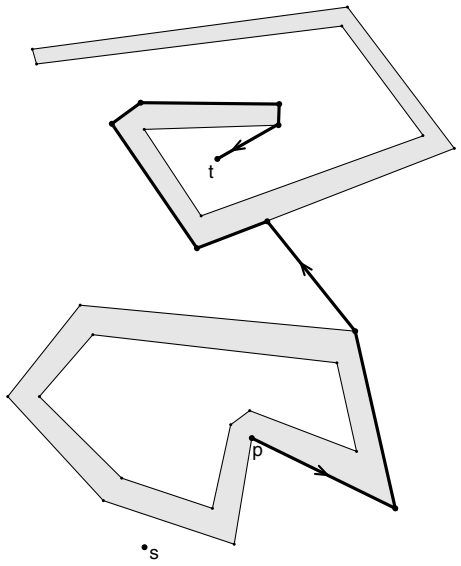


Abb. 7.12 Das Steuerwort $w(p) = L^2TR^5T$ führt den Roboter vom Eckpunkt p ans Ziel.

Sei $\{p_1, p_2, \ldots, p_m\}$ die Menge dieser Punkte. Für jedes p_i gibt es ein Steuerwort $w(p_i)$, das den Roboter von p_i ans Ziel bringt.

Die wesentliche Beweisidee liegt in folgendem überraschenden Lemma:

universelles Steuerwort

Lemma 7.8 *Es gibt ein endliches universelles Steuerwort w über Σ, das den Roboter von* jedem *Punkt p_i aus zum Ziel führt.*

Beweis. Mit $w_1 = w(p_1)$ findet man von p_1 zum Ziel.

Versucht man, dieses Wort auf p_2 als Ausgangspunkt anzuwenden, kann es vorkommen, daß einzelne Grundbefehle sich nicht ausführen lassen; so könnte z. B. als nächster Schritt ein T an der Reihe sein, obwohl der Weg zum Ziel gerade durch ein Hindernis versperrt ist. In diesem Fall wird der Grundbefehl ausgelassen.

Im allgemeinen wird sich der Roboter nach einer solchen Anwendung von w_1 auf p_2 nicht im Zielpunkt befinden, sondern in irgendeinem anderen Punkt q_2; er gelangt aber zum Ziel, wenn er nun seinen Weg mit $w(q_2)$ fortsetzt. Das Wort

$$w_2 = w_1 w(q_2)$$

führt den Roboter also schon von zwei Punkten aus ans Ziel: von p_1 und von p_2. Dabei setzen wir voraus, daß der Roboter keine

weiteren Steuerbefehle mehr ausführt, wenn er das Ziel – etwa nach Abarbeiten von w_1 – erreicht hat.

Dieses Konstruktionsverfahren läßt sich induktiv fortsetzen. Angenommen, wir haben schon ein Wort w_i, das von $p_1, p_2, \ldots, p_i$ aus zum Ziel führt und im Punkt q_{i+1} landet, wenn es auf p_{i+1} angewendet wird. Dann findet

$$w_{i+1} = w_i w(q_{i+1})$$

zusätzlich auch von p_{i+1} aus den Zielpunkt. Mit $w = w_m$ haben wir also das gesuchte universelle Steuerwort gefunden. □

Natürlich kennt der Roboter die Wörter $w(p_i)$ nicht – sonst könnte er ja einfach $w(s)$ anwenden – und schon gar nicht das universelle Wort w. Er kann aber nacheinander alle endlichen Wörter über Σ erzeugen[10] und ihnen nachlaufen. Irgendwann kommt auch das universelle Wort w an die Reihe. Und an welchem Punkt p_i sich der Roboter zu diesem Zeitpunkt auch immer befinden mag, spätestens jetzt wird er nach Lemma 7.8 zum Zielpunkt finden!

Damit ist der Beweis von Theorem 7.7 beendet. □

Für praktische Anwendungen eignet sich dieses Vorgehen nicht besonders gut. Es ist aber für die Theorie interessant, mit welch einfachen Mitteln sich ein Zielpunkt in unbekannter Umgebung finden läßt.

7.3 Kompetitive Strategien

In den vergangenen Abschnitten haben wir Strategien zur Lösung von Problemen bei unvollständiger Information kennengelernt, die zwar ihre Aufgabe erfüllen und das vorgelegte Problem lösen, aber dabei in manchen Fällen hohe Kosten verursachen. So konnte es sowohl beim Pledge-Algorithmus als auch bei der Strategie *Bug* ungünstige Umgebungen geben, in denen die Länge des von der Strategie erzeugten Weges in keinem Verhältnis zur Länge des kürzesten Weges steht, der aus dem Labyrinth ins Freie bzw. zum Zielpunkt führt; siehe Abbildungen 7.8 und 7.11.

Begrenzung der Kosten?

Natürlich läßt sich die optimale Lösung in der Regel nur mit vollständiger Information finden. Aber kann man dem Optimum nicht wenigstens nahekommen?

bin packing

Bei manchen Problemen ist das in der Tat möglich. Betrachten wir als Beispiel einen Klassiker, das Problem *bin packing*. Gegeben ist eine Folge zweidimensionaler Objekte Q_k, $1 \leq k \leq n$. Alle Objekte haben dieselbe Breite, aber individuelle Höhen h_k.

[10] Zum Beispiel in lexikographischer Reihenfolge.

Außerdem stehen n leere Behälter (engl. *bins*) zur Verfügung, die so breit wie die Objekte sind und die Höhe H besitzen, wobei H mindestens so groß ist wie die Höhen h_k der Objekte. Das Problem besteht darin, die Objekte in möglichst wenige Behälter zu verpacken.[11]

Abbildung 7.13 zeigt ein Beispiel, bei dem acht Objekte mit optimaler Platzausnutzung in vier Behälter gepackt wurden. Wenn alle Objekte schon zu Beginn bekannt sind, lassen sich solche optimalen Packungen prinzipiell berechnen – allerdings nicht effizient. Das Problem *bin packing* ist nämlich NP-hart; siehe Garey und Johnson [61].

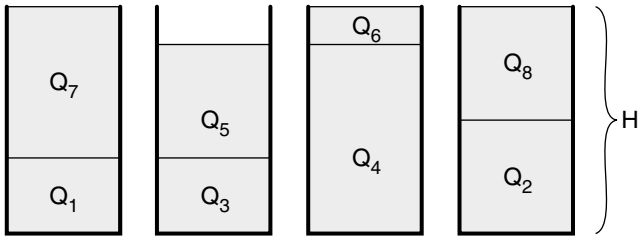


Abb. 7.13 Eine optimale Packung von acht Objekten in vier Behälter identischer Breite und Höhe.

Hier ist ein ganz naives Verfahren zur Lösung dieses Packungsproblems: Man nimmt die Objekte Q_k der Reihe nach in die Hand und legt jedes in den ersten Behälter von links, in dem zu diesem Zeitpunkt noch Platz ist. Dieses Verfahren heißt *first fit*.

first-fit-Strategie

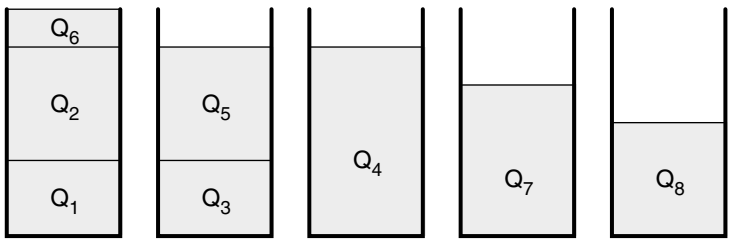


Abb. 7.14 Die mit der *first-fit*-Strategie erzielte Packung benötigt fünf anstelle von vier Behältern.

Für die Objekte aus Abbildung 7.13 ergibt sich dabei die in Abbildung 7.14 gezeigte Verteilung. Die *first-fit*-Strategie geht mit dem Platz nicht ganz so sparsam um wie die optimale Packung

[11] Eine viel kompliziertere Variante dieses Problems tritt beim Verpacken von Umzugsgut in Kisten auf.

und benötigt in diesem Beispiel fünf statt vier Behälter. Allgemein gilt folgende Abschätzung; ihr Beweis ist überraschend einfach.

Theorem 7.9 *Die first-fit-Strategie benötigt höchstens doppelt so viele Behälter wie eine optimale Packung der Objekte.*

Beweis. Angenommen, das *first-fit*-Verfahren benötigt m Behälter für die Verpackung der n Objekte. Alle Behälter bis auf höchstens einen müssen mehr als zur Hälfte gefüllt sein. Wären nämlich für $i < j$ die Behälter mit den Nummern i und j nur halb voll, so hätte *first fit* die Objekte aus dem j-ten Behälter schon im i-ten Behälter unterbringen können! Folglich ist

$$(m-1) \cdot \frac{H}{2} < \sum_{k=1}^{n} h_k \,,$$

also

$$m - 1 < \frac{2}{H} \sum_{k=1}^{n} h_k \leq 2 \left\lceil \frac{\sum_{k=1}^{n} h_k}{H} \right\rceil .$$

Ganz rechts in der Klammer steht der reine Platzbedarf der n Objekte in Behältereinheiten. Die für eine optimale Packung benötigte Anzahl m_{opt} von Behältern kann höchstens größer oder gleich sein. Folglich ist $m \leq 2m_{opt}$. □

Daß es selbst für NP-harte Probleme zuweilen recht effiziente approximative Lösungen geben kann, hatten wir schon am Beispiel des Handlungsreisenden (*traveling salesman*) in Abschnitt 5.3.3 auf Seite 225 gesehen. Die *first-fit*-Strategie für das Verpackungsproblem hat aber noch eine weitere bemerkenswerte Eigenschaft: Sie eignet sich auch als *on-line*-Verfahren, also für Situationen, in denen die Objekte erst während des Verpackens nach und nach eintreffen. Auch dies ist ein typisches Beispiel für eine vom Mangel an Information geprägte Situation.

first fit ist ein *on-line*-Verfahren

Die *first-fit*-Strategie erweist sich in dieser Situation als wettbewerbsfähig: Sie kann es mit der optimalen Lösung aufnehmen und verursacht höchstens doppelt so hohe Kosten wie diese!

Allgemein sei Π ein Problem (etwa das, in einer polygonalen Umgebung einen Zielpunkt aufzusuchen), und sei S eine Strategie, die jedes Beispiel $P \in \Pi$ korrekt löst und dabei die Kosten[12] $K_S(P)$ verursacht. Ferner bezeichne $K_{opt}(P)$ die Kosten einer optimalen Lösung von P. Wir sagen: Die Strategie S ist *kompetitiv*

kompetitive Strategie

[12] Im Unterschied zu Abschnitt 1.2.4 sind hier die Kosten nicht notwendig durch Rechenzeit und Speicherplatzbedarf bestimmt. Es kann sich ebensogut um die Länge eines erzeugten Weges oder die Anzahl von benötigten Behältern handeln, wie wir gesehen haben.

mit Faktor C, wenn es eine Zahl A gibt, so daß für jedes Beispiel $P \in \Pi$ die Abschätzung

$$K_S(P) \leq C \cdot K_{opt}(P) + A$$

gilt. Dabei ist $C \geq 1$ ein sogenannter *kompetitiver Faktor*. Beide Zahlen C und A können reell sein. Sie dürfen aber nur von Π und S abhängen und nicht von P.

kompetitiver Faktor

In dieser Terminologie ist für das *bin-packing*-Problem die Strategie *first fit* kompetitiv mit Faktor 2 und daher erst recht mit jedem Faktor größer als 2. Man kann zeigen, daß *first fit* sogar kompetitiv mit Faktor 1,7 ist, aber für keinen kleineren Faktor.

Wir werden nun weitere Beispiele für kompetitive Bahnplanungsstrategien kennenlernen. Dabei beginnen wir mit einem Problem, das viel elementarer ist als die bisher behandelten Fragen.

7.3.1 Suche nach einer Tür in einer Wand

Angenommen, ein Roboter mit Tastsensor steht vor einer sehr langen Wand.[13] Er weiß, daß es irgendwo eine Tür in der Wand gibt, und möchte auf die andere Seite gelangen. Er weiß aber nicht, ob sich die Tür links oder rechts von seiner aktuellen Position befindet und wie weit sie entfernt ist. Wie soll der Roboter vorgehen?

Er könnte sich eine Richtung – links oder rechts – aussuchen und dann für immer in dieser Richtung an der Wand entlanglaufen. Wenn er zufällig die richtige Richtung rät, löst er sein Problem optimal. Rät er die falsche, löst er es gar nicht. Wenn wir an Strategien interessiert sind, die in *jedem Fall* funktionieren, kommt dieses Vorgehen nicht in Betracht.

Der Roboter muß also seine Bewegungsrichtung immer wieder ändern und abwechselnd den linken und den rechten Teil der Wand nach der Tür absuchen.

Suchrichtung abwechseln

Dabei könnte er folgendermaßen vorgehen: Vom Startpunkt s aus geht er zunächst einen Meter nach rechts und kehrt nach s zurück. Dann geht er zwei Meter nach links und nach s zurück, anschließend drei Meter nach rechts und so fort. Abbildung 7.15 zeigt, welcher Weg sich ergibt, wenn der Roboter seine Suchtiefe jeweils um einen Meter vergrößert.

Suchtiefe inkrementell erhöhen?

Offenbar wird der Roboter bei Verfolgung dieser Strategie die Tür irgendwann erreichen. Angenommen, sie befindet sich $d = l + \varepsilon$ Meter rechts von seinem Startpunkt s, wobei l eine ganze Zahl ist und $\varepsilon > 0$ sehr klein. Wenn der Roboter das rechte Wandstück mit Tiefe l erkundet, wird er die Tür daher gerade eben verfehlen

[13] Wir stellen uns vor, die Wand habe unendliche Länge.

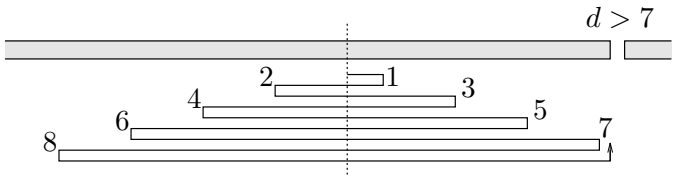


Abb. 7.15 Hier wird die Suchtiefe nach jeder Richtungsänderung um denselben Betrag vergrößert.

und umkehren. Erst beim nächsten Versuch erreicht er sie dann nach $l + \varepsilon$ Metern von s aus.

Insgesamt hat der Roboter eine Wegstrecke der Länge

$$\begin{aligned} 1+1+2+2+\cdots+l+l+(l+1)+(l+1)+l+\varepsilon & \\ = \quad & 2\sum_{i=1}^{l+1} i + l + \varepsilon \\ = \quad & (l+1)(l+2)+l+\varepsilon \\ \in \quad & \Theta(d^2) \end{aligned}$$

zurückgelegt, während der kürzeste Weg zur Tür nur die Länge d hat. Also ist diese Strategie nicht kompetitiv! Sie würde z. B. einen Weg von über 10 Kilometer Länge zurücklegen, um eine Tür zu finden, die 100 Meter entfernt ist.

nicht kompetitiv!

Zum Glück gibt es einen effizienteren Ansatz. Anstatt die Suchtiefe nach jedem Richtungswechsel nur um einen Meter zu vergrößern, *verdoppeln* wir sie jedesmal. Abbildung 7.16 zeigt, welcher Weg sich bei Anwendung dieser Strategie ergibt. Zur rechten Seite betragen die Suchtiefen $2^0, 2^2, 2^4, \ldots$, während zur linken Seite die ungeraden Zweierpotenzen $2^1, 2^3, 2^5, \ldots$ auftreten.

Verdopplung der Suchtiefe

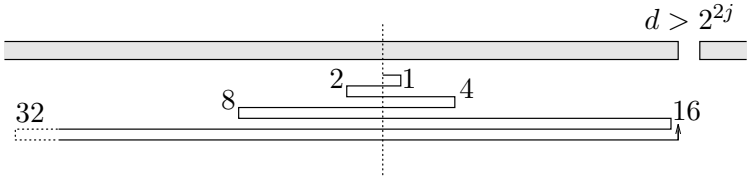


Abb. 7.16 Hier wird die Suchtiefe nach jedem Richtungswechsel verdoppelt.

Auch hier tritt der schlimmste Fall ein, wenn der Roboter die Tür knapp verfehlt und erst bei der nächsten Erkundung findet. Betrachten wir zuerst den Fall, daß die Tür rechts vom Startpunkt liegt. Sei also der Türabstand d zum Startpunkt ein bißchen größer

als eine gerade Zweierpotenz, also $d = 2^{2j} + \varepsilon$. Dann legt der Roboter insgesamt einen Weg der folgenden Länge zurück:

$$\begin{aligned} 2^0 + 2^0 + 2^1 + 2^1 + \cdots + 2^{2j} + 2^{2j} + 2^{2j+1} + 2^{2j+1} + 2^{2j} + \varepsilon & \\ &= 2\sum_{i=0}^{2j+1} 2^i + 2^{2j} + \varepsilon \\ &= 2(2^{2j+2} - 1) + 2^{2j} + \varepsilon \\ &= 9 \cdot 2^{2j} - 2 + \varepsilon \\ &< 9d. \end{aligned}$$

Dieselbe Abschätzung gilt, wenn die Tür im Abstand $d = 2^{2j+1} + \varepsilon$ links vom Startpunkt liegt. Beträgt ihr Abstand dann nur $d = \varepsilon \leq 2^1$, schätzen wir den Weg des Roboters folgendermaßen ab:

$$1 + 1 + \varepsilon < 9\varepsilon + 2 = 9d + 2.$$

Damit haben wir folgendes Resultat bewiesen:

kompetitiv mit Faktor 9

Theorem 7.10 *Die Strategie der abwechselnden Verdopplung der Suchtiefe ist kompetitiv mit dem Faktor 9.*

Mit der Verdopplungsstrategie muß man also schlimmstenfalls 900 Meter zurücklegen, um eine 100 Meter weit entfernte Tür zu finden. Man könnte sich fragen, ob es noch besser geht. Das ist nicht der Fall! Beck und Newman [14] und später Baeza-Yates et al. [11] haben gezeigt, daß die Verdopplungsstrategie optimal ist.

optimal

Theorem 7.11 *Jede kompetitive Strategie zum Auffinden eines Punkts auf einer Geraden hat einen Faktor ≥ 9.*

Der Vollständigkeit halber geben wir hier einen Beweis an. Er zeigt, wie man prinzipiell *lineare Rekursionen* mit elementaren Methoden der Linearen Algebra lösen kann.

lineare Rekursion

Beweis. Sei S eine kompetitive Strategie mit Faktor C, mit der sich ein Punkt auf einer Geraden finden läßt. Wir können die Strategie S durch die Folge $(f_1, f_2, f_3, \ldots)$ der Erkundungsweiten eindeutig beschreiben, wobei wir ohne Einschränkung annehmen dürfen, daß die beiden Wandhälften alternierend erkundet werden und die Erkundungstiefen f_i positiv sind und links wie rechts monoton wachsen. Der Roboter geht also vom Startpunkt s aus f_1 Meter nach rechts und kehrt zurück; dann geht er f_2 Meter nach links und kehrt zurück und so fort.

Weil S nach Voraussetzung kompetitiv mit Faktor C ist, gibt es eine Konstante A, so daß für alle $n \geq 1$ die Abschätzung

$$2\sum_{i=1}^{n+1} f_i + f_n + \varepsilon \leq C(f_n + \varepsilon) + A$$

gilt. Links steht die Länge des Weges, der sich ergibt, wenn beim n-ten Erkundungsgang der gesuchte Punkt gerade um ε verfehlt wird.

Annahme $A = 0$

Diese Aussage gilt für alle $\varepsilon > 0$; also dürfen wir auch $\varepsilon = 0$ setzen. Außerdem nehmen wir zunächst an, daß die additive Konstante A den Wert null hat. Wenn wir die f_n auf die rechte Seite bringen und durch 2 dividieren, ergibt sich

$$\sum_{i=1}^{n-1} f_i + f_{n+1} \leq H f_n \quad \text{mit} \quad H = \frac{C-3}{2}$$

oder

$$f_{n+1} \leq H f_n - \sum_{i=1}^{n-1} f_i. \qquad (*)$$

zu zeigen: $H \geq 3$

Beweisen müssen wir, daß $C \geq 9$ ist, also die Aussage $H \geq 3$. Setzt man in $(*)$ die ebenfalls gültige Abschätzung

$$f_n \leq H f_{n-1} - \sum_{i=1}^{n-2} f_i$$

ein, so erhält man

$$\begin{aligned} f_{n+1} &\leq H^2 f_{n-1} - H\sum_{i=1}^{n-2} f_i - \sum_{i=1}^{n-1} f_i \\ &= (H^2-1) f_{n-1} - (H+1)\sum_{i=1}^{n-2} f_i. \end{aligned}$$

Dieser Ersetzungsvorgang läßt sich iterieren und liefert folgende Aussage:

Übungsaufgabe 7.3 Es seien $(a_i)_i$ und $(b_i)_i$ die durch

$$\begin{aligned} a_0 &= H, & a_{i+1} &:= a_i H - b_i \\ b_0 &= 1, & b_{i+1} &:= a_i + b_i \end{aligned}$$

rekursiv definierten Zahlenfolgen. Man zeige, daß

$$f_{n+1} \leq a_m f_{n-m} - b_m \sum_{i=1}^{n-1-m} f_i$$

gilt für alle $n \geq 1$ und alle $0 \leq m \leq n-1$.

Zunächst halten wir folgende Eigenschaft der Zahlen a_i fest:

Lemma 7.12 *Die Zahlen a_i in Übungsaufgabe 7.3 sind positiv.*

Beweis. Sei $i \geq 0$ beliebig. Angenommen, es wäre $a_i \leq 0$. Dann folgte für $m = i$ und $n = i+1$ aus der Formel in Übungsaufgabe 7.3 die Aussage

$$f_{i+2} \leq a_i f_1 - b_i 0 \leq 0$$

im Widerspruch zu der Voraussetzung, daß alle Erkundungstiefen f_j positiv sind. □

Jetzt geben wir eine geschlossene Darstellung für die rekursiv definierten Zahlen a_i, b_i an.

Lemma 7.13 *Sei $H \neq 3$, dann gilt für die Zahlen a_i, b_i aus Übungsaufgabe 7.3 für alle $i \geq 0$ die Darstellung*

$$\begin{aligned} a_i &= v z^i + \overline{v}\,\overline{z}^i \\ b_i &= v(\overline{z}-1)z^i + \overline{v}(z-1)\overline{z}^i \end{aligned}$$

mit

$$v = \frac{Hz - H - 1}{z - \overline{z}}, \quad \overline{v} = \frac{H\overline{z} - H - 1}{\overline{z} - z};$$

dabei ist

$$z = \frac{1}{2}\left(H + 1 + \sqrt{(H+1)(H-3)}\right)$$

eine Lösung der quadratischen Gleichung

$$t^2 - (H+1)t + H + 1.$$

Hier bedeutet der Querstrich[14] die Umkehrung des Vorzeichens des Wurzelterms; zum Beispiel ist

$$\overline{z} = \frac{1}{2}\left(H + 1 - \sqrt{(H+1)(H-3)}\right)$$

die zweite Lösung der quadratischen Gleichung. Nach Übungsaufgabe 7.3 ist $H = a_0 > 0$, und nach Annahme ist $H \neq 3$. Also hat die Wurzel nicht den Wert 0, und es ist $\overline{z} \neq z$.

Beweis. Man kann die Formel für a_i und b_i ziemlich leicht durch Induktion über i verifizieren, wenn man dabei die Identitäten

$$z\overline{z} = H + 1 = z + \overline{z} \quad \text{und} \quad z - \overline{z} = \sqrt{(H+1)(H-3)}$$

geschickt ausnutzt. Wir wollen statt dessen einen anderen Beweis skizzieren, aus dem auch hervorgeht, wie man überhaupt auf die geschlossene Darstellung für die a_i, b_i kommt.

[14] Wenn $\alpha = \sqrt{(H+1)(H-3)}$ keine rationale Zahl ist, wird durch die Umkehrung des Vorzeichens von α der Konjugationsautomorphismus $z \mapsto \overline{z}$ im quadratischen Zahlkörper $\mathbb{Q}(\alpha)$ über $\mathbb{Q}$ definiert, in dem auch die Zahl z liegt; siehe Lorenz [96].

lineare Rekursion = Matrixmultiplikation

Der Trick besteht darin, die Rekursion als iterierte *Matrixmultiplikation* hinzuschreiben. In der Tat: Es gilt

$$\begin{pmatrix} a_{i+1} \\ b_{i+1} \end{pmatrix} = \begin{pmatrix} H & -1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} a_i \\ b_i \end{pmatrix} = \cdots = \begin{pmatrix} H & -1 \\ 1 & 1 \end{pmatrix}^{i+1} \begin{pmatrix} a_0 \\ b_0 \end{pmatrix}.$$

Jetzt stellen wir den Vektor der Anfangswerte a_0, b_0 als Linearkombination von zwei *Eigenvektoren*[15] V_1, V_2 der Multiplikationsmatrix

$$M = \begin{pmatrix} H & -1 \\ 1 & 1 \end{pmatrix}$$

dar. Dann sind wir sofort fertig! Ist nämlich

$$\begin{pmatrix} a_0 \\ b_0 \end{pmatrix} = v_1 V_1 + v_2 V_2,$$

Eigenwert

und sind z_1, z_2 die *Eigenwerte* der Eigenvektoren V_1, V_2 von M, gilt also $MV_j = z_j V_j$ und dann auch $M^i V_j = z_j^i V_j$ für $j = 1, 2$, so folgt

$$\begin{aligned} \begin{pmatrix} a_i \\ b_i \end{pmatrix} = M^i \begin{pmatrix} a_0 \\ b_0 \end{pmatrix} &= M^i(v_1 V_1 + v_2 V_2) \\ &= v_1 M^i V_1 + v_2 M^i V_2 \\ &= v_1 z_1^i V_1 + v_2 z_2^i V_2, \end{aligned}$$

und die geschlossene Darstellung ergibt sich durch Vergleich der Vektorkoordinaten.

charakteristisches Polynom

Unsere Matrix M hat das *charakteristische Polynom*

$$\begin{aligned} \det(tE - M) &= \begin{vmatrix} t - H & 1 \\ -1 & t - 1 \end{vmatrix} \\ &= t^2 - (H+1)t + H + 1, \end{aligned}$$

wobei det die Determinante bezeichnet und E für die 2×2-Einheitsmatrix steht. Die Eigenwerte sind die Nullstellen, also

$$z, \overline{z} = \frac{1}{2}\left(H + 1 \pm \sqrt{(H+1)(H-3)}\right).$$

Durch Lösung eines linearen Gleichungssystems ergeben sich die zugehörigen Eigenvektoren

$$V = \begin{pmatrix} 1 \\ \overline{z} - 1 \end{pmatrix} \quad \text{und} \quad \overline{V} = \begin{pmatrix} 1 \\ z - 1 \end{pmatrix}$$

[15] Vergleiche z. B. Lorenz [95].

für z und $\overline{z}$. Ebenso läßt sich leicht die lineare Darstellung

$$\begin{aligned}\begin{pmatrix} a_0 \\ b_0 \end{pmatrix} &= \begin{pmatrix} H \\ 1 \end{pmatrix} \\ &= \frac{Hz - H - 1}{z - \overline{z}} \begin{pmatrix} 1 \\ \overline{z} - 1 \end{pmatrix} + \frac{H\overline{z} - H - 1}{\overline{z} - z} \begin{pmatrix} 1 \\ z - 1 \end{pmatrix} \\ &= vV + \overline{v}\overline{V}\end{aligned}$$

ermitteln. Jetzt folgt die Behauptung von Lemma 7.13 durch Einsetzen sofort. □

Den Beweis von Theorem 7.11 können wir nun folgendermaßen zu Ende führen. Angenommen, der kompetitive Faktor C der Strategie S wäre kleiner als 9. Dann ist $H < 3$. Folglich ist die Zahl z in Lemma 7.13 komplex und nicht reell, und der Querstrich bedeutet die Konjugation der komplexen Zahlen.

Die Darstellung der Zahl a_n lautet dann

$$a_n = vz^n + \overline{v}\,\overline{z}^n = 2Re(vz^n), \qquad (**)$$

wobei für eine komplexe Zahl $w = c + di$ mit $Re(w)$ der *Realteil* c von w gemeint ist.

Stellt man sich jede komplexe Zahl $w = c+di$ als Ortsvektor $0w$ des Punktes $w = (c, d)$ im $\mathbb{R}^2$ vor, so läßt sich die Multiplikation von zwei komplexen Zahlen folgendermaßen geometrisch interpretieren: Ihre Winkel zur positiven X-Achse addieren sich, die Längen werden multipliziert.

komplexe Multiplikation, geometrisch betrachtet

Am leichtesten sieht man das an der Darstellung in Polarkoordinaten; für

$$\begin{aligned} v &= r\cos\phi + ir\sin\phi \\ z &= s\cos\psi + is\sin\psi \end{aligned}$$

mit $0 < r, s$ folgt aus den Additionstheoremen der Winkelfunktionen

Additionstheoreme

$$\begin{aligned} vz &= rs(\cos\phi\cos\psi - \sin\phi\sin\psi) + irs(\cos\phi\sin\psi + \sin\phi\cos\psi) \\ &= rs\cos(\phi+\psi) + irs\sin(\phi+\psi). \end{aligned}$$

Die Zahl z in Formel $(**)$ hat einen Winkel $\psi > 0$, weil sie nicht reell ist. Bei jeder Multiplikation mit z wird der Winkel von vz^n um ψ größer. Dann muß irgendwann der Punkt vz^n in der linken Halbebene $\{X \le 0\}$ enthalten sein; für $\psi \le \pi$ ist das unmittelbar klar, wie Abbildung 7.17 zeigt. Einer Linksdrehung um einen Winkel $\psi > \pi$ entspricht eine Rechtsdrehung um $2\pi - \psi < \pi$, die auch irgendwann die linke Halbebene erreicht. Für einen solchen Punkt ist dann

$$a_n = 2Re(vz^n) \le 0,$$

im Widerspruch zu Lemma 7.12, wonach alle a_i positiv sind. Also war die Annahme $C < 9$ falsch!

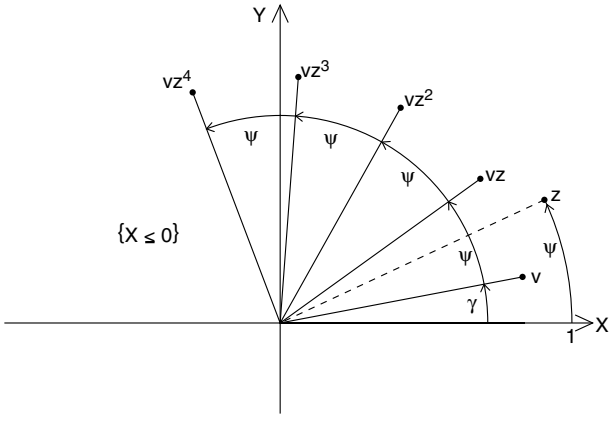


Abb. 7.17 Ist der Winkel ψ von z ungleich null, so muß der Punkt vz^n für geeignetes n in der linken Halbebene liegen.

allgemein: $A \neq 0$

Wir müssen uns nun noch von der Einschränkung befreien, daß die additive Konstante A unserer fiktiven Strategie $S = (f_1, f_2, \ldots)$ den Wert null hat.

Sei also $A > 0$, und sei $\mu > 0$ beliebig. Weil die Erkundungstiefen gegen ∞ streben, können wir einen Index n_0 finden, so daß für alle $n \geq n_0$ gilt

$$Cf_n + A \leq (C + \mu)f_n.$$

Die erste Abschätzung dieses Beweises können wir dann für $\varepsilon = 0$ durch

$$2\sum_{i=n_0}^{n+1} f_i + f_n \leq Cf_n + A \leq (C + \mu)f_n$$

ersetzen und den Beweis ganz analog weiterführen. Es ergibt sich, daß $C+\mu \geq 9$ sein muß, und weil μ beliebig war, folgt die Behauptung $C \geq 9$. Theorem 7.11 ist damit bewiesen: Keine kompetitive Strategie kann bei der Suche nach einer Tür in einer langen Wand einen kleineren Faktor als 9 garantieren! □

Der Ansatz der Matrixmultiplikation eignet sich übrigens auch für andere Typen linearer Rekursion als den, der in Lemma 7.13 auftrat. Hierzu ein Beispiel.

Übungsaufgabe 7.4 Man bestimme eine geschlossene Darstellung für die rekursiv durch

$$f_0 := 1\,, \quad f_1 := 1$$
$$f_{n+2} := f_{n+1} + f_n$$

definierten *Fibonacci-Zahlen*. Fibonacci-Zahlen

Ein alternatives Verfahren zur Lösung linearer Rekursionsgleichungen besteht in der Verwendung erzeugender Funktionen; siehe Graham et al. [64].

Die von Theorem 7.10 vorgestellte Strategie für die Suche nach einem Punkt auf einer Geraden ist übrigens nicht die einzige, die den optimalen kompetitiven Faktor 9 erzielt, wie Teil (i) der folgenden Übungsaufgabe zeigt.

Übungsaufgabe 7.5

(i) Man zeige, daß die Strategie $(f_1, f_2, \ldots)$ mit $f_j = (j+1)2^j$ ebenfalls kompetitiv mit Faktor 9 ist.

(ii) Man bestimme den kompetitiven Faktor der Strategie $(2^0, 2^0, 2^1, 2^1, \ldots, 2^i, 2^i, \ldots)$.

7.3.2 Exponentielle Vergrößerung der Suchtiefe: ein Paradigma

Nicht nur für die Suche nach einem Punkt auf einer Geraden ist das Prinzip der Suchtiefenverdopplung geeignet: Wir haben es hier mit einem Paradigma zu tun, das auch andere Anwendungen erlaubt. Einige davon wollen wir in diesem Abschnitt vorstellen. Dabei beschränken wir uns auf den Nachweis von oberen Schranken für kompetitive Faktoren und verzichten auf Optimalitätsaussagen. Paradigma

Rekapitulieren wir noch einmal das Problem, mit dem wir uns in Abschnitt 7.3.1 beschäftigt haben: Es galt, einen Punkt zu suchen, der in einer von zwei Halbgeraden[16] enthalten ist. Zu diesem Zweck haben wir uns die Halbgeraden abwechselnd vorgenommen und, jedesmal vom Ausgangspunkt beginnend, in exponentiell wachsender Tiefe nach dem Punkt abgesucht. Daß dieses Vorgehen kompetitiv ist, liegt, grob gesagt, an folgender Eigenschaft der geometrischen Summe

$$2^0 + 2^1 + 2^2 + \cdots + 2^n < 2^{n+1}\,:$$

sie ist nur um einen konstanten Faktor größer als ihr letzter Summand 2^n, der die Kosten der optimalen Lösung verkörpert.

[16] Damit meinen wir die Stücke der Wand links und rechts vom Startpunkt.

Verallgemeinerung

So gesehen wirft unser Ergebnis zwei Fragen auf:

- Kommen auch andere Suchräume als Halbgeraden in Betracht?
- Läßt sich das Verfahren auf mehr als zwei Suchräume verallgemeinern?

m Halbgeraden

Eine Antwort auf die zweite Frage findet sich bereits in der Arbeit [11]. Angenommen, m Halbgeraden treffen sich in einem gemeinsamen Startpunkt s, und eine von ihnen enthält den gesuchten Zielpunkt.

Dann läßt sich unser Vorgehen aus dem Fall $m = 2$ folgendermaßen verallgemeinern. Wir legen unter den Halbgeraden eine Reihenfolge fest und durchsuchen sie zyklisch mit den Suchtiefen

$$f_j = \left(\frac{m}{m-1}\right)^j.$$

Theorem 7.14 *Diese Suchstrategie für m Halbgeraden ist kompetitiv mit dem Faktor*

$$2\frac{m^m}{(m-1)^{m-1}} + 1 \leq 2em + 1;$$

dabei ist $e = 2{,}718\ldots$ die Eulersche Zahl.

Beweis. Auch hier tritt der schlimmste Fall ein, wenn der j-te Suchvorgang den Zielpunkt knapp verfehlt, weil seine Entfernung zum Startpunkt $f_j + \varepsilon$ beträgt. Dann wird eine ganze Runde vergeblichen Suchens mit den Tiefen $f_{j+1}, \ldots, f_{j+m-1}$ ausgeführt, bevor die richtige Halbgerade wieder an die Reihe kommt.

Die Gesamtlänge des bis zum Zielpunkt zurückgelegten Weges beträgt daher

$$\begin{aligned} 2\sum_{i=1}^{j+m-1} f_i + f_j + \varepsilon &= 2\sum_{i=1}^{j+m-1}\left(\frac{m}{m-1}\right)^i + f_j + \varepsilon \\ &< 2\frac{(\frac{m}{m-1})^{j+m} - 1}{\frac{m}{m-1} - 1} + f_j + \varepsilon \\ &< 2(m-1)\left(\frac{m}{m-1}\right)^{j+m} + f_j + \varepsilon. \end{aligned}$$

Also ist der kompetitive Faktor nicht größer als

$$\frac{2(m-1)\left(\frac{m}{m-1}\right)^{j+m} + f_j + \varepsilon}{f_j + \varepsilon} \leq 2(m-1)\left(\frac{m}{m-1}\right)^m + 1$$

$$\begin{aligned} &= 2m\left(1+\frac{1}{m-1}\right)^{m-1}+1 \\ &\leq 2me+1. \end{aligned}$$

Dabei haben wir ausgenutzt, daß die Folge $(1+\frac{1}{m})^m$ monoton gegen die Zahl e wächst. □

Beim Suchen eines Punktes auf m Halbgeraden hängt der kompetitive Faktor also von m ab. Man kann übrigens auch hier zeigen, daß für jedes m der obige Faktor $2\frac{m^m}{(m-1)^{m-1}}+1$ optimal ist; siehe Gal [60] and Baeza-Yates et al. [11]. Für $m=2$ ergibt sich der aus Theorem 7.10 bekannte Faktor 9.

Auch beim Suchen auf $m>2$ Halbgeraden kann man alternative Strategien erwägen. Teil (ii) der folgenden Übungsaufgabe verallgemeinert die entsprechende Aussage von Übungsaufgabe 7.5.

Übungsaufgabe 7.6

(i) Welcher Faktor wird beim Suchen auf m Halbgeraden erreicht, wenn man die Suchtiefe bei jedem Halbgeradenbesuch verdoppelt?

(ii) Welcher Faktor ergibt sich, wenn die Suchtiefe nur zu Beginn einer neuen Runde verdoppelt wird, während der Halbgeradenbesuche einer Runde aber jeweils konstant bleibt?

Im Rest dieses Abschnitts betrachten wir eine sehr nützliche Anwendung der Verallgemeinerung auf $m>2$.

Anwendung

Angenommen, ein Roboter befindet sich an einem Startpunkt s in einem einfachen Polygon P. Seine Aufgabe besteht darin, auf möglichst kurzem Weg zu einem Zielpunkt t zu gelangen.

Suche nach Zielpunkt in Polygon

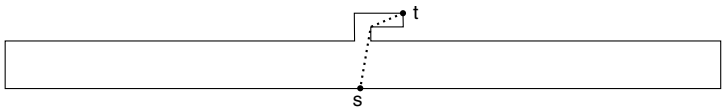


Abb. 7.18 In diesem Polygon ist der kürzeste Weg von s nach t sehr viel kürzer als die beiden Randstücke, die s mit t verbinden.

Lägen s und t beide auf dem Rand von P, könnte der Roboter eine Richtung wählen und dem Polygonrand in dieser Richtung so lange folgen, bis er t erreicht hat. Dieses Verfahren ist aber nicht kompetitiv, wie Abbildung 7.18 zeigt. Daran ändert sich auch nichts, wenn der Roboter den Rand von P abwechselnd links- und rechtsherum absucht und die Erkundungstiefe verdoppelt.

Wenn wir wollen, daß der Roboter sich von der Wand lösen und den Zielpunkt direkt ansteuern kann, müssen wir ihn zunächst mit

Sichtsystem

einem Sichtsystem ausstatten, das ihm an jeder aktuellen Position p in P das *Sichtbarkeitspolygon* $vis(p)$ für seine Bahnplanung zur Verfügung stellt; vergleiche Abschnitt 4.3. Wir können dann annehmen, daß der Zielpunkt t optisch markiert ist und vom Roboter erkannt wird, sobald der ihn sieht. Mit Hilfe dieses Sichtsystems

partielle Karte

kann der Roboter, während er sich umherbewegt, eine *partielle Karte* unterhalten, auf der alle Teile von P eingezeichnet sind, die zu irgendeinem Zeitpunkt schon einmal sichtbar waren. Ein Beispiel ist in Abbildung 7.19 zu sehen. Die partielle Karte ist zu jedem Zeitpunkt ein Teilpolygon von P. Hat sich der Roboter längs eines Weges π von s fortbewegt, enthält die partielle Karte genau die Punkte

$$\bigcup_{p \in \pi} vis(p).$$

Mit welchen Algorithmen man die vom Sichtsystem gelieferte Information am schnellsten für die Aktualisierung der partiellen Karte verwenden kann, ist eine interessante Frage. Wir gehen ihr hier nicht weiter nach, weil wir die Weglänge als das entscheidende Kostenmaß ansehen.

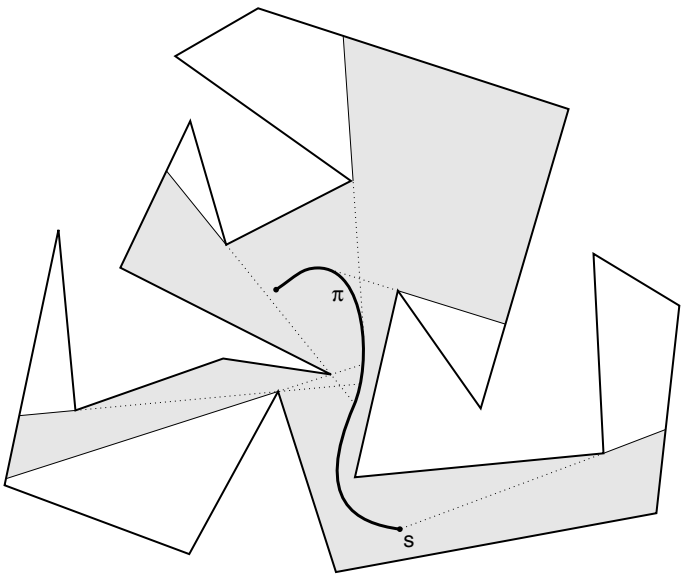


Abb. 7.19 Die partielle Karte des Roboters, nachdem er den Weg π zurückgelegt hat.

Leider gibt es auch für einen Roboter mit Sichtsystem keine Strategie, mit der er in einem *beliebigen* einfachen Polygon mit beschränktem relativem Umweg einen Zielpunkt finden kann!

Lemma 7.15 *Keine Strategie für die Suche nach einem Zielpunkt kann in beliebigen Polygonen mit n Ecken stets einen kleineren Faktor als $\frac{n}{2} - 1$ für das Weglängenverhältnis garantieren.*

Beweis. Abbildung 7.20 zeigt ein Polygon mit n Ecken, das aus $\frac{n}{4}$ vielen Korridoren besteht, die beim Startpunkt s zusammenlaufen und an ihren Ecken abgeknickt sind. Dem Roboter bleibt nichts anderes übrig, als einen Korridor nach dem anderen zu inspizieren und hinter die Ecken zu schauen. Im ungünstigsten Fall liegt der Zielpunkt t am Ende des zuletzt besuchten Korridors. Wenn die Korridore die Länge 1 besitzen und ihre Weiten sowie die Längen der abgeknickten Enden vernachlässigbar klein sind, ergibt sich ein Gesamtweg der Länge

$$(\frac{n}{4} - 1)2 + 1 = \frac{n}{2} - 1$$

gegenüber der Länge 1 des kürzesten Weges von s nach t. □

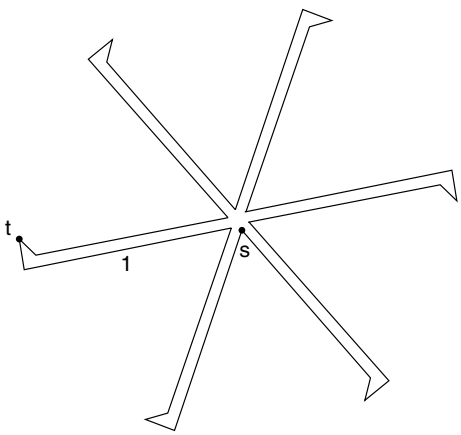


Abb. 7.20 Im ungünstigsten Fall liegt der Zielpunkt am Ende des zuletzt besuchten Korridors.

Wenn wir also überhaupt eine Strategie angeben können, bei der das Verhältnis der Länge des zurückgelegten Weges zur Länge des kürzesten Weges stets beschränkt ist, kann diese Schranke keine Konstante sein. Im schlimmsten Fall kann der zurückgelegte Weg linear von der Anzahl n der Polygonecken abhängen.

Weglängenverhältnis in $\Omega(n)$

Auf solche Polygone, die nur aus mehreren am Startpunkt zusammenlaufenden Korridoren bestehen, können wir die Strategie der exponentiellen Vergrößerung der Suchtiefe aus Theorem 7.14 anwenden, denn die Korridore lassen sich genau wie Halbgeraden behandeln, auch wenn sie mehrere Knickstellen aufweisen. Damit

könnten wir immerhin ein Weglängenverhältnis in $O(n)$ garantieren.

Aber wie verfahren wir mit Polygonen, die nicht nur aus Korridoren bestehen?

Hier kommt uns folgende Struktur zu Hilfe: Wir betrachten die Gesamtheit aller kürzesten Wege im Polygon P, die vom Startpunkt s zu den Eckpunkten von P führen. Wie die folgende Übungsaufgabe zeigt, bilden diese kürzesten Wege einen Baum. Er wird der *Baum der kürzesten Wege* von s in P genannt (engl.: *shortest path tree*) und mit *SPT* bezeichnet.

Baum der kürzesten Wege

Ein Beispiel ist in Abbildung 7.21 zu sehen. Die kürzesten Wege verlaufen wie gespannte Gummibänder, die nur an spitzen Ecken von P abknicken können.

Übungsaufgabe 7.7

(i) Seien s und t zwei Punkte in einem einfachen Polygon P. Man zeige, daß der kürzeste Weg von s nach t, der ganz in P verläuft, eindeutig bestimmt ist und daß es sich dabei um eine polygonale Kette handelt, die nur an spitzen Ecken von P Knickstellen haben kann.

(ii) Sei s ein Punkt in P. Man zeige, daß die kürzesten Wege von s zu den Eckpunkten von P einen Baum bilden.

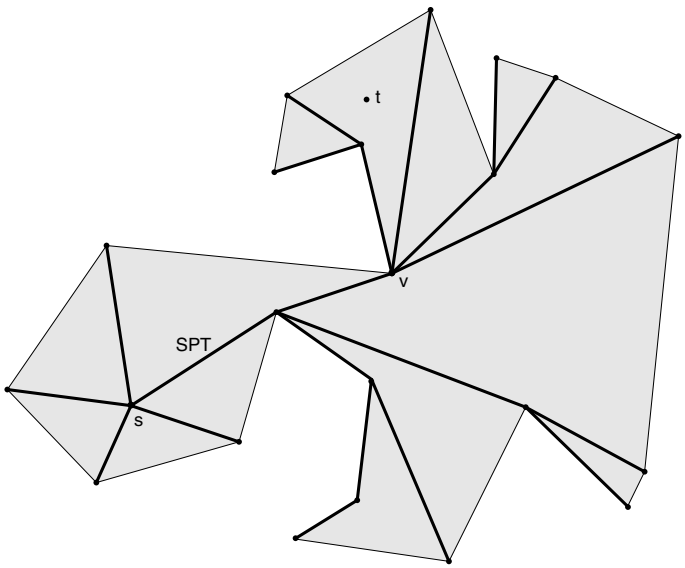


Abb. 7.21 Der Baum *SPT* der kürzesten Wege vom Startpunkt s zu allen Eckpunkten des Polygons P.

Bei einem Polygon P mit n Ecken und einem Startpunkt s im Innern von P hat der Baum SPT genau $n+1$ Knoten, darunter $m \leq n$ viele Blätter.

Der Roboter könnte die m Wege, die in SPT von der Wurzel zu den Blättern führen, wie m disjunkte Halbgeraden behandeln und sie in zyklischer Reihenfolge mit exponentiell wachsender Suchtiefe nach einem Punkt v absuchen, von dem aus der Zielpunkt t sichtbar ist.[17]

Warum gibt es im Baum SPT solche Punkte? Sei v der letzte Knickpunkt auf dem kürzesten Weg π von s nach t; siehe Abbildung 7.21. Dann ist v insbesondere ein Eckpunkt von P und damit ein Knoten von SPT. Außerdem ist von v aus der Zielpunkt t sichtbar.

Ein Blick auf Abbildung 7.21 zeigt, daß es außer v noch andere Knoten in SPT geben kann, die den Zielpunkt t sehen können. Wenn der Roboter einen von ihnen vor v erreicht, würde er in der Praxis von dort direkt zum Ziel laufen. Um die nachfolgende Analyse zu vereinfachen, nehmen wir an, daß der Roboter so lange weiterläuft, bis er den Punkt v erreicht, und erst von dort den Zielpunkt ansteuert.

Mit $w(s,p)$ und $d(s,p)$ bezeichnen wir die Längen des tatsächlich zurückgelegten und des kürzesten Weges von s zu einem Punkt p. Weil v auf dem kürzesten Weg von s nach t liegt, ist $d(s,t) = d(s,v) + |vt|$. Wegen $d(s,v) \leq w(s,v)$ und nach Theorem 7.14 ergäbe sich folgendes Weglängenverhältnis:

$$\frac{w(s,t)}{d(s,t)} = \frac{w(s,v)+|vt|}{d(s,v)+|vt|} \leq \frac{w(s,v)}{d(s,v)} \leq 2em+1.$$

Wir können also einen kompetitiven Faktor erreichen, der linear von m abhängt, der Anzahl der Blätter im Baum aller kürzesten Wege von s.

Dieser Ansatz stößt aber auf folgende Schwierigkeit: Der Roboter kennt weder den Baum SPT noch die Anzahl m seiner Blätter. Wie soll er den Vergrößerungsfaktor $\frac{m}{m-1}$ für die Suchtiefe bilden?

Hier ist Übungsaufgabe 7.6 (ii) von Nutzen: Statt die Suchtiefe jeweils vor dem Besuch des nächsten Weges mit $\frac{m}{m-1}$ zu multiplizieren, verdoppeln wir sie vor jeder neuen Runde. Dabei bleibt das Weglängenverhältnis unterhalb von $8m$.

Tiefenverdopplung vor jeder Runde

Daß der Roboter nicht den gesamten Baum SPT kennt, stellt kein ernsthaftes Problem dar. Er kennt nämlich zu jedem Zeitpunkt den Verlauf von SPT, soweit er sehen kann, und das genügt für die Planung des nächsten Schritts! Genauer gilt folgende Aussage:

[17] Ist t schon von s aus sichtbar, geht der Roboter direkt zum Ziel; diesen trivialen Fall betrachten wir im folgenden nicht mehr.

Übungsaufgabe 7.8 Für jeden Punkt $p \in P$, den der Roboter bereits gesehen hat, kennt er den kürzesten Weg von s nach p.

zyklische Wegliste L

Der Roboter kann die kürzesten Wege zu allen Eckpunkten von P, die schon einmal sichtbar waren, in einer zyklischen Liste L speichern. Er begeht diese Wege rundenweise nacheinander, etwa im Uhrzeigersinn.

Abbildung 7.22 zeigt ein Beispiel. Der Roboter ist gerade auf dem Weg von s über v_1 nach v_2. Dort angekommen, entdeckt er die Eckpunkte v_5, v_6, v_7. Er weiß, daß der kürzeste Weg von s zu jedem von ihnen über v_2 führt, und ersetzt deshalb in L den Weg (s, v_1, v_2) durch die drei benachbarten Wege (s, v_1, v_2, v_5), (s, v_1, v_2, v_6) und (s, v_1, v_2, v_7). Wenn der Roboter die für die aktuelle Runde zur Verfügung stehende Suchtiefe in v_2 noch nicht voll ausgeschöpft hat, fährt er mit der Erkundung von (v_2, v_5) fort. Anschließend kommt dann der Weg (s, v_1, v_2, v_6) an die Reihe.

Aktualisierung von L

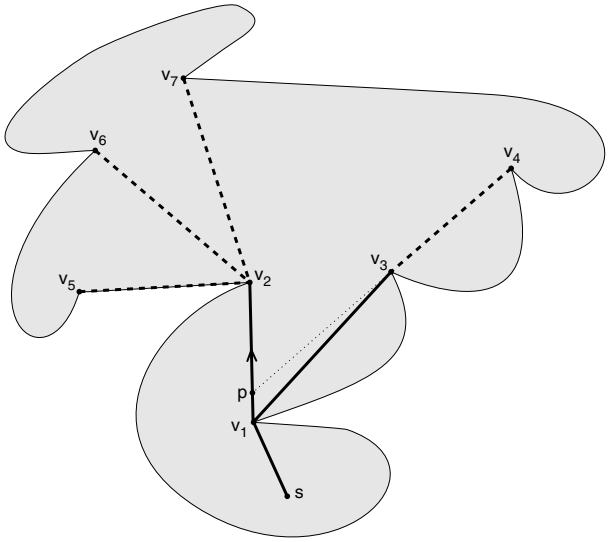


Abb. 7.22 Wenn der Roboter den Punkt v_2 erreicht, entdeckt er die Eckpunkte v_5, v_6, v_7 und weiß, daß ihre kürzesten Wege zu s über v_2 führen.

Während der Roboter noch auf v_2 zugeht, sieht er im Punkt p den Eckpunkt v_4. Er weiß, daß der kürzeste Weg von s zu v_4 über v_3 führt, und könnte in der Liste L den Weg (s, v_1, v_3) bereits jetzt um das Stück (v_3, v_4) verlängern. Ebensogut kann er diese Aktualisierung zurückstellen, bis der Weg (s, v_1, v_3) an der Reihe ist und der Roboter bei seiner Ankunft in v_3 den Punkt v_4 „wiederentdeckt“.

Damit haben wir folgendes Ergebnis:

Theorem 7.16 *Für Roboter mit Sichtsystem gibt es eine kompetitive Strategie für die Suche nach einem Zielpunkt, die in jedem einfachen Polygon mit n Ecken den Faktor $8n$ garantiert.*

Nach Lemma 7.15 ist ein kompetitiver Faktor in $O(n)$ das Beste, was wir erwarten können. Der konstante Faktor 8 läßt sich aber noch verkleinern.

praktische Verbesserungen

Außerdem kann man an der hier beschriebenen Strategie verschiedene praktische Verbesserungen vornehmen, die zwar im *worst case* nicht zu einer weiteren Verkleinerung des kompetitiven Faktors führen, aber in vielen Fällen Weglänge einsparen. So ist es zum Beispiel beim Erkunden der Wege im Baum SPT nicht erforderlich, jedesmal bis zur Wurzel s zurückzukehren, bevor der nächste Weg an die Reihe kommt. Der Roboter braucht nur zu dem Eckpunkt zurückzugehen, bei dem sich die beiden Wege teilen. In manchen Situationen ist nicht einmal das erforderlich: Wenn der Roboter in Abbildung 7.22 am Punkt v_7 angekommen ist, kann er zur Erkundung des nächsten Weges (s, v_1, v_3, v_4) direkt zu v_4 gehen.

Ferner braucht man nicht den gesamten Baum SPT zu begehen: Weil jeder Punkt in P schon von einem inneren Knoten von SPT sichtbar ist, werden die zu den Blättern des Baums führenden Kanten nicht benötigt. Das Applet

`http://www.geometrylab.de/STIP/`

zeigt, wie die verbesserte Strategie arbeitet.

Dieser Ansatz funktioniert nicht nur im Innern einfacher Polygone. Er läßt sich auf beliebige Umgebungen verallgemeinern und stellt damit für Roboter mit Sichtsystem eine effiziente Alternative zum Verfahren *Bug* dar, das in Abschnitt 7.2 vorgestellt wurde.

Bei unserer Suche nach einem Zielpunkt in einem einfachen Polygon waren sowohl die Beschaffenheit der Umgebung als auch die Lage des Ziels zunächst unbekannt. Eine naheliegende Frage lautet: Wieviel hilft es beim Suchen, wenn man die Umgebung schon kennt? Fleischer et al. [57] haben gezeigt, daß man dadurch nur einen konstanten Faktor einsparen kann.

Schließlich sei noch erwähnt, daß das Verfahren der Suchtiefenverdopplung nicht nur auf eindimensionale Suchräume anwendbar ist.

Übungsaufgabe 7.9 Man gebe eine kompetitive Strategie an, um in der Ebene von einem Startpunkt aus eine Halbgerade zu finden.

andere Kostenmaße

Die in realen Anwendungen entstehenden Kosten hängen natürlich nicht nur von der Länge des Weges ab, den der Roboter zurücklegen muß. Demaine et al. [41] haben deshalb beim Suchen auf m Halbgeraden auch für jede Kehrtwendung des Roboters Kosten veranschlagt.

Manche Roboter müssen anhalten, um ihre Umgebung mit dem Lasersystem abzutasten. Fekete et al. [54] haben deshalb neben der Weglänge auch die Anzahl der Scans betrachtet, die der Roboter unterwegs ausführt.

7.4 Suche nach dem Kern eines Polygons

Angenommen, ein Roboter mit Sichtsystem befindet sich in einer ihm unbekannten Umgebung, die durch ein einfaches Polygon P dargestellt wird. Seine Aufgabe besteht darin, sich auf möglichst kurzem Weg zu einem Punkt zu begeben, von dem aus er ganz P überblicken kann.

Diese Aufgabe ist nur lösbar, wenn das Polygon P *sternförmig* ist, also einen nicht-leeren *Kern* besitzt. In Abschnitt 4.4 hatten wir einen Algorithmus angegeben, um für ein gegebenes Polygon P seinen Kern effizient zu berechnen. Hier geht es darum, in einem unbekannten Polygon von einem Startpunkt s aus den nächstgelegenen Punkt k von $ker(P)$ zu erreichen. Die Länge des von s nach k zurückgelegten Weges werden wir mit dem Abstand $|sk|$ vergleichen. Als Punkt in $ker(P)$ kann k ja ganz P sehen, insbesondere den Startpunkt s. Folglich ist der kürzeste Weg in P von s zum Kern von P tatsächlich das Liniensegment sk.

zum nächsten Punkt k von $ker(P)$ gehen

Obwohl der Roboter schon von s aus alle Punkte von $ker(P)$ sehen kann, hat er keine Möglichkeit, sie von den übrigen Punkten von P zu unterscheiden. Abbildung 7.23 zeigt ein Beispiel: Das rechte Polygon entsteht aus dem linken durch Spiegelung an der senkrechten Geraden durch s. Das Sichtbarkeitspolygon von Punkt s aus ist in beiden Polygonen dasselbe, aber die Lage der Kerne ist sehr verschieden!

Man kann aus diesem Beispiel folgende *untere Schranke* für den kompetitiven Faktor herleiten.

Theorem 7.17 *Wenn es überhaupt eine kompetitive Strategie gibt, mit der man zum nächstgelegenen Punkt des Kerns gelangen kann, so beträgt ihr Faktor mindestens* $\sqrt{2}$.

Beweis. Sei S eine kompetitive Strategie, die den Roboter stets zu dem Kernpunkt k führt, der dem Startpunkt s am nächsten liegt. Wir lassen den Roboter die Strategie S auf das Beispiel in Abbildung 7.23 anwenden und beobachten seinen Weg. Die Punkte

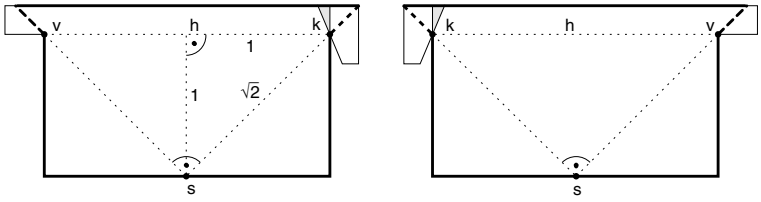


Abb. 7.23 Identische Sichtbarkeitspolygone von s, aber völlig verschiedene Kerne.

s, k und v bilden ein gleichschenkliges, rechtwinkliges Dreieck der Höhe 1. Solange das waagerechte Liniensegment h nicht erreicht ist, kann der Roboter nicht erkennen, ob der linke oder der rechte Fall vorliegt, weil für jeden Punkt p unterhalb von h die Sichtbarkeitspolygone links und rechts identisch sind.

Kurz bevor der Roboter das Segment h erreicht, können wir uns also immer noch für eines der beiden Polygone entscheiden. Wir machen es dem Roboter so schwer wie möglich und wählen das linke, wenn der Roboter im Begriff ist, auf die linke Hälfte von h zu treffen. Steuert er auf die rechte Hälfte von h zu, entscheiden wir uns für das rechte Polygon.

Auf diese Weise muß der Roboter nach Erreichen von h noch einen Weg mit mindestens der Länge 1 zum „richtigen“ Endpunkt k von h zurücklegen. Auch der erste Teil seines Weges von s nach h hat mindestens die Länge 1. Die Entfernung von s nach k beträgt aber nur $\sqrt{2}$!

Damit ergibt sich folgende *untere Schranke* für das Verhältnis der Weglängen:

$$\frac{w(s,k)}{d(s,k)} \geq \frac{1+1}{\sqrt{2}} = \sqrt{2}.$$

Weil die in Abbildung 7.23 eingezeichneten Längen beliebig vergrößert werden können, ist $\sqrt{2}$ eine untere Schranke für den kompetitiven Faktor jeder denkbaren Strategie zum Finden des Kerns. □

Im nächsten Abschnitt werden wir eine Strategie entwickeln, mit der sich der nächstgelegene Punkt im Kern kompetitiv finden läßt. Die untere Schranke $\sqrt{2}$ werden wir dabei allerdings nicht erreichen.

7.4.1 Die Strategie CAB

Der Roboter kennt zwar nicht die genaue Lage des Kerns von P, aber er weiß, daß von dort ganz P sichtbar sein wird. Mehr noch: Bei einer geradlinigen Bewegung von s zu einem Punkt in $ker(P)$ würde das Sichtbarkeitspolygon $vis(p)$ der aktuellen Position monoton wachsen, bis es schließlich gleich P ist.

so laufen, daß $vis(p)$ wächst

Wie man einen geraden Weg zum Kern finden soll, ist nicht klar; aber der Roboter könnte zumindest seinen Weg so wählen, daß sein Sichtbarkeitspolygon beständig wächst!

Auf diesem naheliegenden Ansatz beruht unsere Strategie. Betrachten wir das Polygon P in Abbildung 7.24. Von der aktuellen Position p aus sind vier spitze Ecken nicht voll einsehbar. Jede von ihnen trägt eine künstliche Kante zu $vis(p)$ bei.

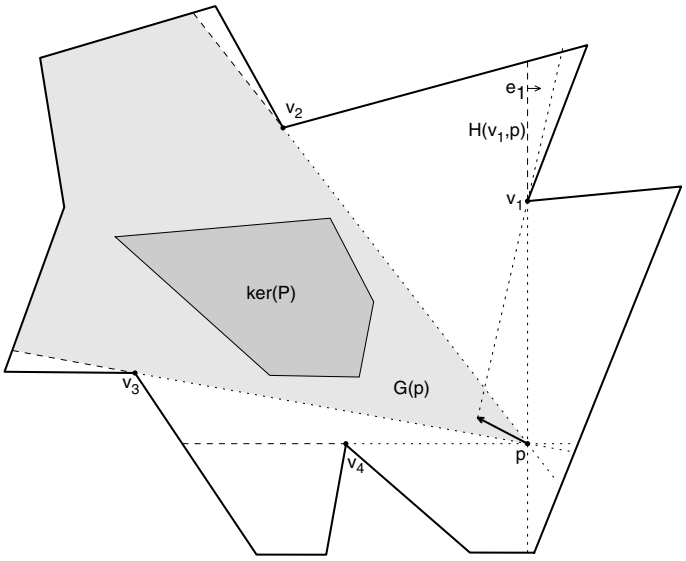


Abb. 7.24 Strategie *CAB* führt den Roboter entlang der Winkelhalbierenden in den hellgrau eingezeichneten Bereich $G(p)$.

Die künstliche Kante e_i von v_i definiert zwei Halbebenen. Eine von ihnen – nennen wir sie $H(v_i, p)$ – liegt auf der von p aus sichtbaren Seite von e_i. Sie enthält den Kern von P. Damit die Kante e_i so um die Ecke v_i rotiert, daß der sichtbare Teil von P größer wird, muß sich der Roboter *in die Halbebene* $H(v_i, p)$ *hineinbewegen*. In Abbildung 7.24 ist dies für die Ecke v_1 eingezeichnet.

Diese Überlegung trifft auf *alle* künstlichen Kanten von $vis(p)$ zu. Wenn das Sichtbarkeitspolygon sich also vergrößern soll, muß

der Roboter in den *Gewinnbereich* Gewinnbereich

$$G(p) = \bigcap_i H(v_i, p)$$

hineinlaufen, wobei der Durchschnitt über alle spitzen Ecken v_i von P gebildet wird, die eine künstliche Kante e_i von $vis(p)$ erzeugen.

Weil sich alle Geraden durch die Kanten e_i im Punkt p treffen, ist der Durchschnitt $G(p)$ der Halbebenen $H(v_i, p)$ ein Winkelbereich am Punkt p, der von nicht mehr als zwei Geraden berandet wird. Die spitzen Ecken, die zusammen mit p diese Geraden definieren, nennen wir die *wesentlichen Ecken* von $vis(p)$. wesentliche Ecke

In Abbildung 7.24 ist der Gewinnbereich $G(p)$ hellgrau dargestellt. Die beiden wesentlichen Ecken sind v_2 und v_3.

Unsere Strategie versucht, von der aktuellen Position p aus jeweils kontinuierlich der *Winkelhalbierenden* zu folgen, die in den Gewinnbereich hineinführt. Dabei ändern sich die wesentlichen Ecken hin und wieder, wie wir gleich noch sehen werden. Der Punkt p und im allgemeinen auch die Winkelhalbierende verändern sich sogar stetig. Aus diesem Grund nennen wir unsere Strategie CAB (=*continuous angular bisector*). Strategie CAB

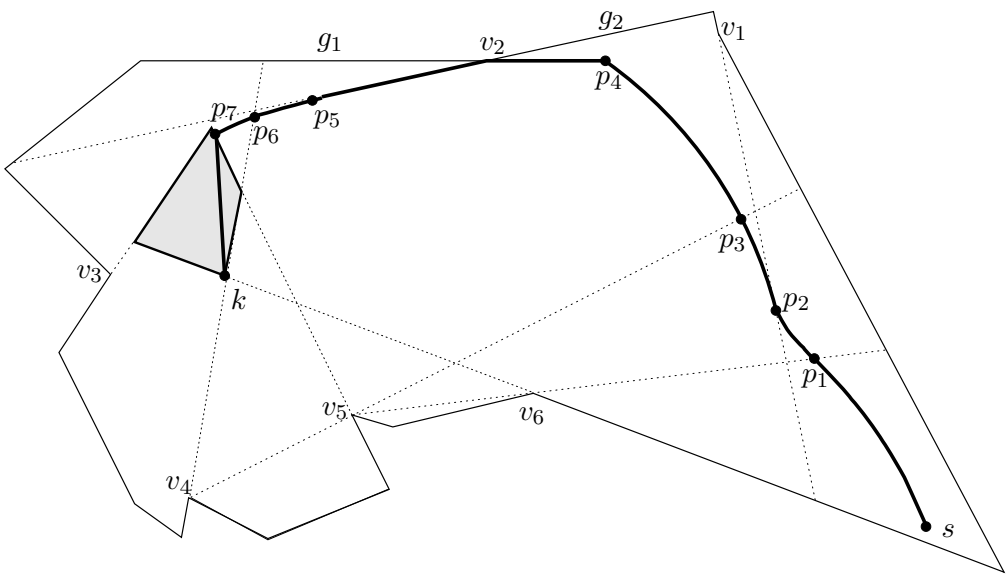


Abb. 7.25 So findet die Strategie CAB vom Startpunkt s zum nächstgelegenen Punkt k des Polygonkerns.

Beispiel für *CAB*

Sehen wir einmal zu, welchen Weg *CAB* im Beispiel von Abbildung 7.25 einschlägt! Vom Startpunkt s aus sind nur zwei spitze Ecken, nämlich v_1 und v_6, nicht voll einsehbar. Diese beiden sind also anfangs die wesentlichen Ecken, und der Roboter muß deshalb zunächst der Winkelhalbierenden von pv_1 und pv_6 folgen, wobei p jeweils die aktuelle Position bedeutet.

In Abbildung 7.26 (i) ist die Situation noch einmal exemplarisch dargestellt. Beide wesentlichen Ecken, v_1 und v_6, liegen auf dem Rand des Gewinnbereichs $G(p)$. Man kann sich nun folgendes überlegen: Wenn der Roboter stets der Winkelhalbierenden von v_1 und v_6 folgt, verändert sich die *Differenz* der Abstände

$$|pv_1| - |pv_6|$$

seiner aktuellen Position p zu den beiden Ecken nicht. Durch diese Bedingung wird der Zweig einer *Hyperbel* mit den Brennpunkten v_1 und v_6 definiert, der sich zum näheren Eckpunkt krümmt; siehe z. B. Bronstein et al. [22]. In dem Spezialfall, daß die Abstände zu den wesentlichen Ecken gleich sind, wird die Hyperbel zur Geraden: dem Bisektor der Eckpunkte. Diese Situation war in Abbildung 7.24 dargestellt.

Hyperbel

Spezialfall: Gerade

Kommen wir zu unserem Beispiel in Abbildung 7.25 zurück. Ab dem Startpunkt s besteht die Bahn zunächst aus einem Hyperbelstück. Im Punkt p_1 wird der Eckpunkt v_5 entdeckt. Er löst als neue wesentliche Ecke seinen Vorgänger v_6 ab, weil die Halbebene $H(v_5, p)$ ab jetzt ein Stück von $H(v_6, p) \cap H(v_1, p)$ abschneidet. Ab p_1 ist die Roboterbahn also auch ein Stück einer Hyperbel; sie krümmt sich aber jetzt nach v_1.[18]

Im Punkt p_2 kann der Roboter hinter die wesentliche Ecke v_1 sehen. Da auf dem Weg von p_1 nach p_2 auch v_6 schon voll einsehbar geworden ist, bleibt gegenwärtig nur die Ecke v_5 übrig. Ab p_2 besteht daher der Gewinnbereich nur noch aus der Halbebene $H(v_5, p)$, und v_5 ist die einzige wesentliche Ecke. Wie sieht in diesem Fall die von *CAB* eingeschlagene Bahn aus? Im aktuellen Punkt p steht die Winkelhalbierende auf der Geraden durch v_5 senkrecht. Also folgt der Roboter einem *Kreisbogen* um v_5, siehe Abbildung 7.26 (iii).

Spezialfall: Kreis

Das nächste Ereignis tritt im Punkt p_3 ein: Die Ecke v_4 wird sichtbar. Ab jetzt gibt es wieder zwei wesentliche Ecken, nämlich v_4 und v_5. Im Unterschied zu früher liegt ab p_3 aber nur v_4 auf dem Rand des Gewinnbereichs, wie Abbildung 7.26 (ii) illustriert. Wenn wir den anderen wesentlichen Eckpunkt v_5 an p spiegeln,

[18] Wir mußten übrigens in Abbildung 7.25 ein wenig nachhelfen, um das Krümmungsverhalten der einzelnen Wegstücke klarer hervortreten zu lassen; dieses Bild ist also nicht ganz exakt.

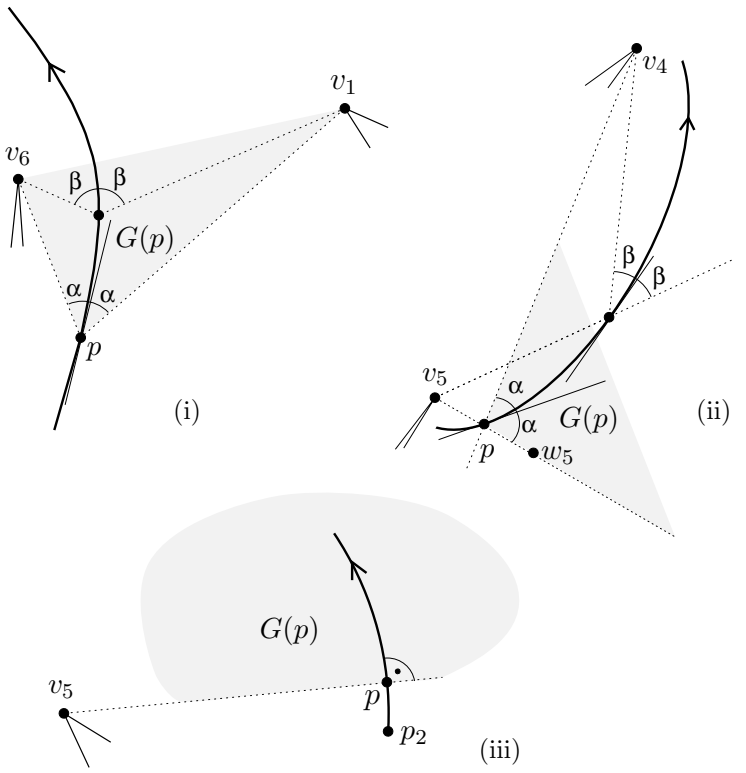


Abb. 7.26 In (i) folgt der Roboter an jedem Punkt p der Winkelhalbierenden von v_1 und v_6. Hierdurch entsteht ein Hyperbelzweig. In (ii) läuft der Roboter stets in Richtung der Winkelhalbierenden von v_4 und dem Spiegelbild w_5 von v_5 an p. Dabei ergibt sich eine Ellipse. Wenn man in (iii) an jedem Punkt p in Richtung der Senkrechten zur Geraden durch v_5 läuft, ergibt sich ein Kreisbogen.

ergibt sich ein Punkt w_5 auf dem Rand von $G(p)$. Der Roboter folgt stets der Winkelhalbierenden von v_4 und dem (sich mit p bewegenden) Punkt w_5. Man kann sich überlegen, daß dabei die *Summe* der Abstände

$$|pv_4| + |pv_5|$$

zu den wesentlichen Ecken konstant bleibt. Durch diese Bedingung wird eine *Ellipse* mit den Brennpunkten v_4 und v_5 definiert.[19] Ellipse

[19] Man kann diese Tatsache zum Zeichnen von Ellipsen verwenden: Die beiden Enden eines Fadens werden in den Brennpunkten fixiert. Dann spannt man den Faden mit einem Zeichenstift und führt den Stift um die Brennpunkte herum.

Wir könnten den vorher aufgetretenen Kreis als Spezialfall einer Ellipse ansehen, bei der die beiden Brennpunkte zusammenfallen.

In Abbildung 7.25 endet das Ellipsenstück um v_4, v_5 im Punkt p_4. Hier gerät unsere Strategie CAB in Bedrängnis: Würde der Roboter nämlich der Ellipse weiter folgen, ginge seine Sicht auf die Kante g_1 mit Endpunkt v_2 verloren.

gleiten

Das darf nicht geschehen! Darum soll der Roboter zunächst an der Verlängerung von g_1 entlanggleiten und ab v_2 an der Verlängerung der ebenfalls sichtbaren Kante g_2.

Am Punkt p_5 sind v_4 und v_5 immer noch die beiden wesentlichen Ecken. Die Winkelhalbierende des Gewinnbereichs, der CAB ja stets zu folgen versucht, zeigt aber nicht länger in die verbotene Halbebene jenseits der Verlängerung von g_2. Also löst sich der Roboter von dieser Kantenverlängerung und setzt seinen Weg mit einem Ellipsenstück fort.

Übungsaufgabe 7.10 Was geschieht am Punkt p_6 in Abbildung 7.25?

Schließlich erreicht der Roboter den Punkt p_7 im Kern des Polygons. Dort könnte er eigentlich bleiben, denn P ist ja nun vollständig sichtbar. Aber die Aufgabenstellung lautete, zu dem Punkt k in $ker(P)$ zu gehen, der dem Startpunkt s am nächsten ist. Weil der Roboter den Kern inzwischen erreicht hat, kennt er das ganze Polygon und kann den zu s nächsten Punkt k bestimmen. Er beendet seinen Weg mit dem Liniensegment von p_7 nach k durch den konvexen Kern von P.

Das oben illustrierte Entlanggleiten kann man folgendermaßen präzisieren. Für jeden Punkt p in P sei $E(p)$ der Durchschnitt der inneren Halbebenen aller Kanten e von P, die den Punkt p in ihrer Verlängerung enthalten und von p aus sichtbar sind. Dazu zählen natürlich auch diejenigen Kanten, die selbst den Punkt p enthalten.

in $E(p)$ bleibt Sichtbarkeit erhalten

Der Roboter soll den *Haltebereich* $E(p)$ nicht verlassen, damit die schon erreichte Sichtbarkeit erhalten bleibt.

Für die meisten Punkte $p \in P$ bedeutet diese Forderung gar keine Einschränkung, weil sie nicht auf einer Kantenverlängerung liegen und $E(p)$ als Durchschnitt einer leeren Menge von Halbebenen die ganze Ebene ist.

Für Punkte auf Polygonkanten oder Verlängerungen von sichtbaren Polygonkanten ist $E(p)$ ein Winkelbereich am Punkt p. In Abbildung 7.25 ist beispielsweise $E(v_2)$ der Durchschnitt der inneren Halbebenen der Kanten g_1 und g_2. Offenbar enthält auch $E(p)$ den Kern von P.

Konflikte wie beim Punkt p_4 treten immer dann auf, wenn die Winkelhalbierende von $G(p)$ nicht in $E(p)$ enthalten ist. In diesem

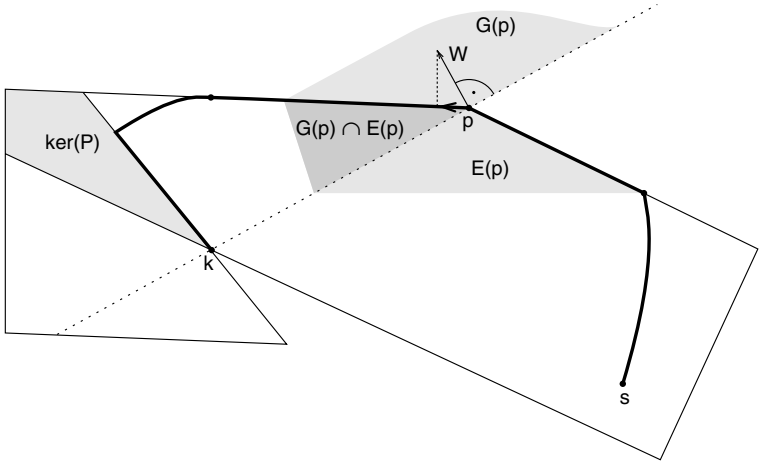


Abb. 7.27 Am Eckpunkt p zeigt die Winkelhalbierende von $G(p)$ aus dem Bereich $E(p)$ heraus. Der Roboter folgt ihrer Projektion auf den Rand von $E(p)$.

Fall gleitet der Roboter am Rand von $E(p)$ entlang, in Richtung der Projektion der Winkelhalbierenden; siehe Abbildung 7.27.

Zusammenfassend können wir unsere Strategie CAB folgendermaßen darstellen: Strategie CAB

$p := s$;
repeat
  berechne Gewinnbereich $G(p)$;
  $W :=$ Winkelhalbierende von $G(p)$;
  berechne Haltebereich $E(p)$;
  if $W \subset E(p)$
    then folge W
    else folge der Projektion von W auf den Rand von $E(p)$
until $vis(p) = P$; (* Kern erreicht *)
gehe geradewegs zum Punkt $k \in ker(P)$, der s am nächsten ist

Die Frage lautet nun: Wie lang ist der von s nach k zurückgelegte Weg im Verhältnis zu $|sk|$? Hiermit beschäftigen wir uns als nächstes.

7.4.2 Eine Eigenschaft der von CAB erzeugten Wege

Auf den ersten Blick scheint es schwierig, über die Länge der von CAB erzeugten Wege eine Aussage zu machen: Sie enthalten viele verschiedenartige Stücke, unter anderem auch Ellipsenstücke, bei Längenberechnung schwierig

denen das Integral für die Weglänge keine geschlossene Darstellung besitzt.

Die Anzahl der Wegstücke läßt sich folgendermaßen abschätzen:

Lemma 7.18 *Der von Strategie CAB in einem Polygon mit n Ecken erzeugte Weg besteht aus nur $O(n)$ vielen Stücken.*

Beweis. Da sich nach Definition der Strategie das Sichtbarkeitspolygon ständig vergrößert, kann die Bahn des Roboters keine Schlingen enthalten.

Ein Wegstück kann enden, wenn *CAB* eine neue wesentliche Ecke entdeckt oder wenn eine schon entdeckte wesentliche Ecke vollständig sichtbar wird. Weil jede Ecke von P nur einmal entdeckt oder vollständig sichtbar werden kann, gibt es nur $O(n)$ viele solche Ereignisse.

Außerdem kann es vorkommen, daß der Roboter auf den Rand des Haltebereichs stößt, also auf eine der n Kanten von P oder ihre Verlängerung, und daran entlangzugleiten beginnt. Wenn sich der Roboter von einer solchen Kantenverlängerung wieder löst, folgt er einer konvexen Bahn, die ihn von der Verlängerung wegführt. Er kann sie also nur dann ein weiteres Mal treffen, wenn sich zwischendurch seine Bahnkurve ändert. Das wiederum kann nur geschehen, wenn sich eine wesentliche Ecke ändert, also insgesamt nur $O(n)$ mal.

Schließlich endet ein Wegstück, wenn der Roboter den Kern erreicht und danach durch den Kern zum Punkt k geht. Dies geschieht aber jeweils nur einmal. □

Glücklicherweise läßt sich das Problem mit den Ellipsenstücken elegant umgehen. Jeder Weg, der von der Strategie *CAB* erzeugt wird, weist nämlich eine interessante strukturelle Eigenschaft auf, die uns bei der Längenabschätzung hilft. Wir definieren zunächst diese Eigenschaft und zeigen dann, daß sie auf *CAB*-Wege zutrifft.

Sei π ein Weg, der aus endlich vielen glatten Stücken besteht. Liegt dann der Punkt p im Innern eines glatten Stücks, hat π im Punkt p eine eindeutig bestimmte Tangente T und eine Normale N, die auf T senkrecht steht. Ist p ein Punkt von π, an dem zwei glatte Stücke zusammentreffen, so definieren die einseitigen Normalen N_1, N_2 der beiden glatten Kurvenstücke ein ganzes Bündel von Geraden durch p; siehe Abbildung 7.28. Jede von ihnen gilt als Normale von π im Punkt p.

Normale

Ein stückweise glatter, von s nach t orientierter Weg π heißt *selbstnähernd*, wenn für jeden Punkt p und jede Normale N in p folgende Eigenschaft erfüllt ist: Das restliche Wegstück von p

selbstnähernder Weg

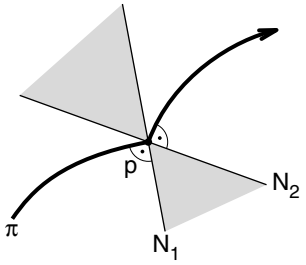


Abb. 7.28 Das Normalenbündel an einer Knickstelle p.

bis t liegt ganz in der abgeschlossenen Halbebene *vor*[20] der Normalen N.

Abbildung 7.29 zeigt ein Beispiel für einen selbstnähernden Weg. Bei entgegengesetzter Orientierung wäre dieser Weg nicht selbstnähernd: Schon die Normale durch t schneidet ja den Rest des Weges.

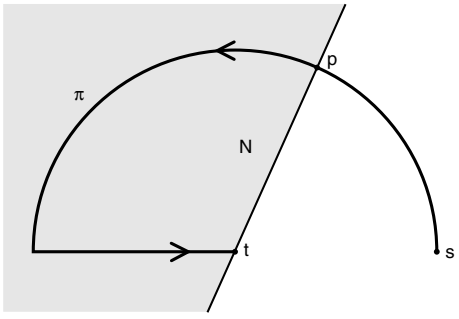


Abb. 7.29 Bei einem selbstnähernden orientierten Weg liegt für jeden Punkt p der Rest des Weges ab p vor der Normalen durch p.

Was bedeutet die Bezeichnung „selbstnähernd“? Wenn man eine Gerade in einem Punkt p rechtwinklig überquert – wie der Weg π seine Normale –, kommt man jedem Punkt näher, der in Laufrichtung vor der Geraden liegt. Ein selbstnähernder Weg π hat also die Eigenschaft, daß er in jedem Punkt p allen seinen „zukünftigen“ Punkten näher kommt. Man kann diese Eigenschaft äquivalent auch folgendermaßen beschreiben: Sind a, b, c drei aufeinander folgende Punkte von π, so liegt b näher an c als a.

logarithmische Spirale

Ein besonders interessantes Exemplar ist die *logarithmische Spirale* in Abbildung 7.30. Sie läßt sich durch die Forderung definieren, daß der Winkel α zwischen der Tangente und der Geraden

[20] Das ist die Halbebene, die der Roboter im Punkt p vor sich sieht.

durch das Zentrum an jedem Punkt konstant ist. Für eine gewisse Zahl α in der Nähe von 74,66° ist jede Normale gleichzeitig Tangente an einem später durchlaufenen Punkt – die Spirale ist also gerade eben noch selbstnähernd!

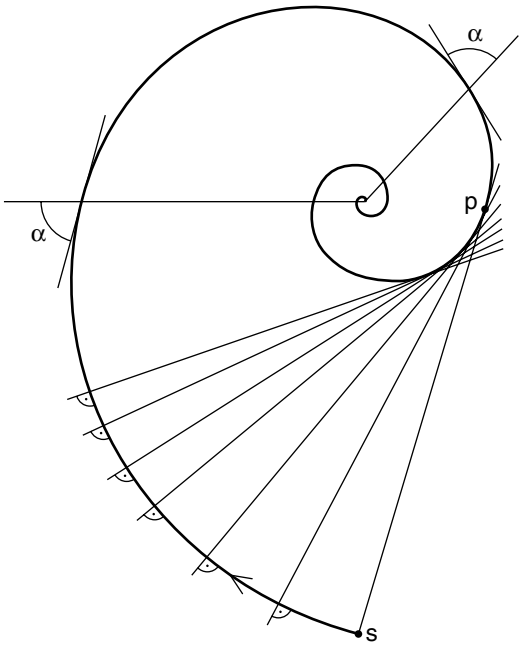


Abb. 7.30 Bei der logarithmischen Spirale mit Anstellwinkel $\alpha \approx 74{,}66°$ ist jede Normale gleichzeitig eine Tangente.

Für die Analyse der *CAB*-Wege ist nun folgende Aussage wichtig:

Theorem 7.19 *Jeder von der Strategie CAB erzeugte Weg ist selbstnähernd.*

Beweis. Zunächst betrachten wir nur den Weg π' des Roboters vom Startpunkt s bis zum ersten Punkt des Kerns von P. In Abbildung 7.25 wäre also π' das Wegstück von s bis p_7.

Kern des Sichtbarkeitspolygons

Für jeden Punkt $p \in \pi'$ untersuchen wir den Kern des Sichtbarkeitspolygons von p. Dieses Teilpolygon $ker(vis(p))$ ist konvex und enthält p als Eckpunkt; vergleiche Abbildung 7.31. Genauer gilt:

Lemma 7.20 *In einer hinreichend kleinen Umgebung eines Punktes $p \in P$ stimmt $ker(vis(p))$ mit dem Durchschnitt der Winkelbereiche $G(p) \cap E(p)$ überein.*

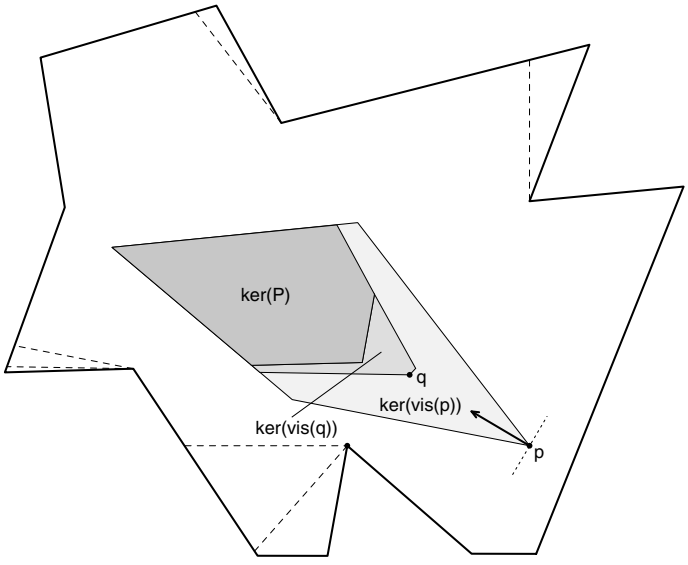


Abb. 7.31 Wird das Sichtbarkeitspolygon größer, so schrumpft sein Kern.

Beweis. Der Winkelbereich $G(p) \cap E(p)$ ist nach Definition Durchschnitt aller inneren Halbebenen von

- künstlichen Kanten des Sichtbarkeitspolygons $vis(p)$ und
- Kanten, deren Verlängerung p enthält und die von p aus sichtbar sind. Auch sie gehören zu $vis(p)$.

Andererseits ist der Kern des Polygons $vis(p)$ der Durchschnitt der inneren Halbebenen *aller* seiner Kanten. Außer den oben aufgeführten zählen dazu nur solche Kanten, deren Verlängerung den Punkt p nicht enthält und deshalb eine hinreichend kleine Umgebung von p nicht schneidet. □

Die Strategie *CAB* lenkt den Roboter so, daß sein Sichtbarkeitspolygon immer größer wird und schließlich mit ganz P übereinstimmt. Das folgende Lemma zeigt, daß dabei der Kern des Sichtbarkeitspolygons ständig schrumpft, bis er schließlich mit $ker(P)$ übereinstimmt; siehe auch Abbildung 7.31.

$vis(p)$ wächst, $ker(vis(p))$ schrumpft

Lemma 7.21 *Seien p, q Punkte im Polygon P mit $vis(p) \subseteq vis(q)$. Dann gilt*

$$ker(P) \subseteq ker(vis(q)) \subseteq ker(vis(p)).$$

Beweis. Daß die erste Inklusion $ker(P) \subseteq ker(vis(q))$ gilt, hatten wir bereits in Übungsaufgabe 4.16 auf Seite 195 gesehen.

Jetzt betrachten wir das Teilpolygon $P' := vis_P(q)$ von P.[21] Nach Voraussetzung ist

$$p \in vis_P(p) \subseteq vis_P(q) = P'.$$

Wir können also die schon bewiesene erste Inklusion auf p und P' anwenden und erhalten

$$ker(vis_P(q)) = ker(P') \subseteq ker(vis_{P'}(p)).$$

Nun gilt $vis_{P'}(p) \subseteq vis_P(p)$, denn wenn ein Liniensegment px in P' liegt, so liegt es erst recht in der größeren Menge P.

Die Umkehrung gilt aber auch: Liegt x in $vis_P(p)$, so folgt nach Voraussetzung

$$px \subset vis_P(p) \subseteq vis_P(q) = P',$$

also $x \in vis_{P'}(p)$. Also ist $vis_{P'}(p) = vis_P(p)$, und insgesamt folgt die zweite Inklusion

$$ker(vis_P(q)) \subseteq ker(vis_P(p)) \qquad \square$$

Daß der Kern des Sichtbarkeitspolygons immer kleiner wird, hat folgende Konsequenz:

restlicher Weg im Kern enthalten

Korollar 7.22 *Für jeden Punkt p auf dem Weg des Roboters ist der Rest des Weges ab p ganz in $ker(vis(p))$ enthalten.*

Beweis. Jeder Punkt $q \in \pi'$, der vom Roboter später besucht wird, liegt auf dem Rand von $ker(vis(q)) \subseteq ker(vis(p))$, und das letzte Segment des Weges ist in $ker(P) \subseteq ker(vis(p))$ enthalten. $\square$

Eine Normale im Bahnpunkt p kann aber niemals das Innere von $ker(vis(p))$ schneiden! Denn der Roboter folgt ja in der Regel der Winkelhalbierenden des Bereichs $G(p)$, der einen Winkel von höchstens 180° hat; siehe Abbildung 7.31.

Wo aber die Winkelhalbierende von $G(p)$ den Bereich $E(p)$ verläßt, gleitet der Roboter am Rand von $G(p) \cap E(p)$ entlang, und dieser Bereich kann dann höchstens einen Winkel von 90° haben; siehe Abbildung 7.27.

Der Kern von $vis(p)$, und nach Korollar 7.22 damit auch der restliche Weg ab dem Punkt p, ist also in der abgeschlossenen Halbebene hinter der Normalen enthalten. Der Weg des Roboters ist deshalb selbstnähernd. Damit ist Theorem 7.19 bewiesen. $\square$

[21] Der untere Index in $vis_P(q)$ drückt aus, in welchem Polygon die Sichtbarkeit bestimmt wird.

Was ist nun mit diesem Theorem gewonnen? Wenn man versucht, auf begrenztem Raum lange selbstnähernde Wege zu zeichnen, macht man folgende Beobachtung: Selbstnähernde Wege benötigen viel Platz, um sich zu winden, weil sie ihre alten Normalen nie wieder kreuzen dürfen. Dieser Eindruck läßt sich folgendermaßen präzisieren:

Theorem 7.23 *Ein selbstnähernder Weg kann höchstens* 5,3331 ... *mal so lang sein wie der Abstand seiner Endpunkte. Diese Schranke ist scharf.*

Auf den Beweis müssen wir aus Platzgründen verzichten. Er findet sich bei Icking und Klein [78], wo auch die Strategie *CAB* eingeführt wird. Das wichtigste Hilfsmittel besteht in folgender Aussage:

Lemma 7.24 *Ein selbstnähernder Weg ist höchstens so lang wie der Umfang seiner konvexen Hülle.*

Im Fall der Spirale in Abbildung 7.30 gilt in Lemma 7.24 sogar die Gleichheit, wie man sich folgendermaßen veranschaulichen kann: Zur konvexen Hülle der Spirale gehört ihr Anfangsstück von s nach p und das Liniensegment sp. Stellen wir uns sp als einen Faden vor, der im Punkt p fixiert ist. Wenn wir den Faden gespannt halten und über die innere Spirale aufwickeln, läuft der Punkt s die ganze Spirale entlang. Die Strecke sp ist also genauso lang wie der Teil der Spirale im Inneren ihrer konvexen Hülle.

Übungsaufgabe 7.11 Man setze Lemma 7.24 voraus und folgere, daß ein selbstnähernder Weg höchstens 2π-mal so lang sein kann wie der Abstand seiner Endpunkte.

Aus Theorem 7.23 folgt, daß die Strategie *CAB* zum Finden des nächstgelegenen Punktes im Kern kompetitiv mit dem Faktor 5,34 ist.

CAB kompetitiv mit Faktor 5,34

Rote [122] hat gezeigt, daß für Kurven, die sogar in beiden Richtungen selbstnähernd sind, – sogenannte *Kurven mit wachsenden Sehnen* – das Verhältnis zwischen Kurvenlänge und Abstand der Endpunkte höchstens $\frac{2}{3}\pi \approx 2{,}094$ beträgt.

Kurven mit wachsenden Sehnen

Unter Verwendung dieses Resultats konnten Lee und Chwa [93] zeigen, daß die Strategie *CAB* sogar $\pi + 1$-kompetitiv ist. Eine andere Strategie, die stets stückweise geradlinige Wege erzeugt, und den kompetitiven Faktor $1 + 2\sqrt{2} = 3,828\ldots$ besitzt, wurde von Lee et. al. [94] angegeben.

Damit beenden wir unsere Beschäftigung mit kompetitiven Strategien für die Bewegungsplanung von Robotern. Von dem hier behandelten Problem, in einer unbekannten Umgebung ein Ziel zu finden, gibt es zahlreiche Varianten. So lassen sich zum Beispiel für spezielle Polygone, die *Straßen* genannt werden, Strategien mit konstantem kompetitiven Faktor angeben; siehe etwa Icking et. al. [79]. Wie das Suchen in Straßen funktioniert, zeigt das Applet

`http://www.geometrylab.de/SIRIUS/`

Erkunden einer Umgebung

Außerdem ist das *Erkunden* einer nicht bekannten Umgebung wichtig. Hierfür werden bei Deng et al. [42] und Hoffmann et al. [75] Strategien vorgestellt; siehe die Applets

`http://www.geometrylab.de/ScoutStep/`

und

`http://www.geometrylab.de/SAM/`

Lokalisierung

Ebenso interessant ist das Problem der *Lokalisierung*, bei dem es darum geht, anhand der lokalen Sichtinformation in bekannter Umgebung den eigenen Standort zu bestimmen. Hierzu findet sich Näheres in Guibas et al. [67] und Dudek et al. [50]; vergleiche auch [77]. Weitere interessante Probleme ergeben sich, wenn mehrere Roboter eine Aufgabe gemeinsam lösen sollen; siehe z. B. Cieliebak et al. [33] und Aronov et al. [7].

Über die Berechnung kürzester Wege in bekannten Umgebungen existiert eine Fülle an Ergebnissen, zum Beispiel der Übersichtsartikel [106] von Mitchell oder das einführende Buch von Gritzmann [65].

Auch außerhalb der Algorithmischen Geometrie gibt es zahlreiche Anwendungen für kompetitive Strategien, zum Beispiel bei der Bevorratung physischer Speicherseiten im Hauptspeicher eines Rechners (Paging), bei der Verteilung von Jobs in Multiprozessorsystemen (Scheduling) oder beim Aufbau von Verbindungen in Kommunikationsnetzwerken (Routing). Hier läßt sich ebenfalls das Phänomen beobachten, daß die verwendeten Strategien selbst meistens sehr einfach sind, während ihre Analyse manchmal harte Probleme aufgibt.

Wer sich für dieses Gebiet interessiert, findet bei Borodin und El-Yaniv [21], Fiat und Woeginger [55] oder Ottmann et al. [115] einen Überblick über kompetitive Strategien innerhalb und außerhalb der Algorithmischen Geometrie.

Lösungen der Übungsaufgaben

Übungsaufgabe 7.1 Sei T der gerichtete Graph, dessen Knoten die Gebiete der Ebene sind, die nach Ausschneiden der polygonalen Ketten übrigbleiben.[22] Zwei Knoten werden durch eine Kante verbunden, wenn ihre Gebiete an eine gemeinsame Kette angrenzen. Weil die Ketten sich nicht schneiden, umschließt dann ein Gebiet das andere vollständig und unmittelbar; die Kante in T wird vom äußeren zum inneren Gebiet orientiert. Einen geschlossenen Weg aus lauter gleich orientierten Kanten kann es in T nicht geben. Jeder Knoten kann nur *eine* eingehende Kante haben, denn jedes Gebiet kann nur von *einem* anderen Gebiet unmittelbar umschlossen werden. Also ist T ein Baum, dessen Wurzel aus dem unbeschränkten Gebiet besteht. Bäume sind aber 2-färbbar, wie man durch Induktion sofort sieht.

Übungsaufgabe 7.2 Für jedes i gilt

$$|D_i t| = |D_i A_{i+1}| + |A_{i+1} t| \geq |D_i A_{i+1}| + |D_{i+1} t|,$$

denn A_{i+1} liegt auf der Geraden von D_i nach t, ist aber mindestens so weit davon entfernt wie D_{i+1}. Damit ergibt sich

$$\begin{aligned} |st| = |sA_1| + |A_1 t| &\geq |sA_1| + |D_1 t| \\ &\geq |sA_1| + |D_1 A_2| + |D_2 t| \\ &\geq |sA_1| + |D_1 A_2| + |D_2 A_3| + |D_3 t| \\ &\geq \dots \end{aligned}$$

Übungsaufgabe 7.3 Der Beweis erfolgt für festes n durch Induktion über m. Für $m = 0$ entspricht die Behauptung der Formel $(*)$ auf Seite 338. Gelte also für $m < n - 1$

$$f_{n+1} \leq a_m f_{n-m} - b_m \sum_{i=1}^{n-1-m} f_i \,.$$

Wir setzen Abschätzung $(*)$ für f_{n-m} ein und erhalten

$$\begin{aligned} f_{n+1} &\leq a_m H f_{n-m-1} - a_m \sum_{i=1}^{n-m-2} f_i - b_m \sum_{i=1}^{n-1-m} f_i \\ &= (a_m H - b_m) f_{n-m-1} - (a_m + b_m) \sum_{i=1}^{n-2-m} f_i \\ &= a_{m+1}\, f_{n-(m+1)} - b_{m+1} \sum_{i=1}^{n-1-(m+1)} f_i. \end{aligned}$$

[22] Nach dem Jordanschen Kurvensatz hat jede Kette ein inneres und ein äußeres Gebiet. Jedes von ihnen kann weitere Ketten enthalten.

Übungsaufgabe 7.4 Offenbar ist

$$\begin{pmatrix} f_{n+1} \\ f_{n+2} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} f_n \\ f_{n+1} \end{pmatrix} = \cdots = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n+1} \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Die hier auftretende Matrix hat das charakteristische Polynom

$$\begin{vmatrix} t & -1 \\ -1 & t-1 \end{vmatrix} = t^2 - t - 1$$

und die Eigenwerte, d. h. Nullstellen des Polynoms,

$$\frac{1+\sqrt{5}}{2} \quad \text{und} \quad \frac{1-\sqrt{5}}{2}$$

mit den Eigenvektoren

$$\begin{pmatrix} 1 \\ \frac{1+\sqrt{5}}{2} \end{pmatrix} \quad \text{und} \quad \begin{pmatrix} 1 \\ \frac{1-\sqrt{5}}{2} \end{pmatrix}.$$

Für den Vektor der Anfangswerte gilt die Darstellung

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \begin{pmatrix} 1 \\ \frac{1+\sqrt{5}}{2} \end{pmatrix} - \frac{1-\sqrt{5}}{2} \begin{pmatrix} 1 \\ \frac{1-\sqrt{5}}{2} \end{pmatrix} \right),$$

und damit ergibt sich

$$\begin{pmatrix} f_{n+1} \\ f_{n+2} \end{pmatrix} = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+2} \begin{pmatrix} 1 \\ \frac{1+\sqrt{5}}{2} \end{pmatrix} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+2} \begin{pmatrix} 1 \\ \frac{1-\sqrt{5}}{2} \end{pmatrix} \right),$$

also

$$f_{n+1} = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+2} \right).$$

Übungsaufgabe 7.5

(i) Der schlimmste Fall tritt ein, wenn der gesuchte Punkt den Abstand $f_j + \varepsilon$ vom Startpunkt hat. Dann hat der bei der Suche zurückgelegte Weg die Länge

$$2 \sum_{i=1}^{j+1} (i+1)2^i + (j+1)2^j + \varepsilon,$$

und als kompetitiven Faktor erhalten wir

$$\begin{aligned} \frac{2 \sum_{i=1}^{j+1} (i+1)2^i + (j+1)2^j + \varepsilon}{(j+1)2^j + \varepsilon} &\leq \frac{2 \sum_{i=1}^{j+1} (i+1)2^i}{(j+1)2^j} + 1 \\ &= \frac{2(j+1)2^{j+2}}{(j+1)2^j} + 1 \\ &= 9. \end{aligned}$$

Hierbei haben wir

$$\sum_{i=1}^{j+1}(i+1)2^i = (j+1)2^{j+2}$$

benutzt; diese Formel erhält man, indem man in der Gleichung

$$\sum_{i=0}^{j+2} X^i = \frac{X^{j+3}-1}{X-1}$$

auf beiden Seiten die Ableitung bildet und dann $X=2$ einsetzt.
Diese Strategie liefert also auch den optimalen Faktor 9 und kommt dabei noch etwas schneller voran, weil die Erkundungstiefe statt 2^j jetzt $(j+1)2^j$ beträgt.

(ii) Hier ergibt sich im schlimmsten Fall der Faktor

$$\begin{aligned}\frac{4\sum_{i=0}^{j} 2^i + 2\cdot 2^{j+1} + 2^j + \varepsilon}{2^j+\varepsilon} &\leq \frac{4\sum_{i=0}^{j} 2^i}{2^j} + 4 + 1\\ &= \frac{4(2^{j+1}-1)}{2^j} + 5\\ &\leq 13.\end{aligned}$$

Weil ε beliebig klein und j beliebig groß gewählt werden darf, sind die Abschätzungen scharf. Also kann diese Strategie keinen kleineren Faktor als 13 erreichen.

Übungsaufgabe 7.6

(i) Für $f_j = 2^j$ ergibt sich der kompetitive Faktor $2^{m+1}+1$, indem man im Beweis von Theorem 7.14 den Term $\frac{m}{m-1}$ durch 2 ersetzt. Dieser Kompetitivitätsfaktor hängt nicht linear, sondern sogar exponentiell von der Anzahl der Halbgeraden ab!

(ii) Im schlimmsten Fall verfehlt die Suche mit Tiefe 2^j den Zielpunkt um ein $\varepsilon > 0$, und unmittelbar danach beginnt eine neue Runde mit der Suchtiefe 2^{j+1}. Dann beträgt das Weglängenverhältnis

$$\begin{aligned}\frac{2m\sum_{i=0}^{j} 2^i + 2(m-1)2^{j+1} + 2^j + \varepsilon}{2^j+\varepsilon} &\leq \frac{2m2^{j+1} + 2(m-1)2^{j+1}}{2^j} + 1\\ &\leq 8m-3.\end{aligned}$$

Hier bleibt also der Faktor linear in m! Für $m=2$ ergibt sich übrigens der Wert 13, wie schon in Übungsaufgabe 7.5 (ii).

Übungsaufgabe 7.7

(i) Kann man den Punkt t schon von s aus sehen, so ist das Liniensegment st der kürzeste Weg von s nach t im Polygon P. Andernfalls liegt t in einer Höhle von $vis(s)$, also auf der von s nicht einsehbaren Seite einer künstlichen Kante e von $vis(s)$, die durch eine spitze Ecke s' von P verursacht wird; siehe Abbildung 7.32.

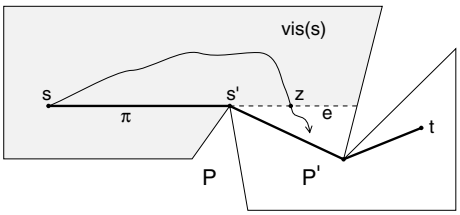


Abb. 7.32 Der kürzeste Weg von s nach t führt über s'.

Jeder Weg von s nach t muß die künstliche Kante e kreuzen. Sei z der erste Kreuzungspunkt; dann kann der Weg nur kürzer werden, wenn man sein Anfangsstück durch sz ersetzt. Also beginnt der kürzeste Weg π von s nach t mit einem Liniensegment, das von s zumindest bis s' führt. Daran muß sich ein kürzester Weg von s' nach t anschließen, der die Kante e nicht mehr kreuzen kann.

Jetzt betrachten wir das in Abbildung 7.32 weiß dargestellte Teilpolygon P' von P. Weil P' zumindest eine Ecke weniger besitzt als P, können wir per Induktion annehmen, daß die Behauptung (i) für den kürzesten Weg in P' von s' nach t bereits bewiesen ist, und sind fertig.

(ii) Seien p_1, p_2 zwei beliebige Punkte in P. Würden die kürzesten Wege π_i von s nach p_i sich erst trennen und dann in einem Punkt q wieder treffen, so hätten wir *zwei* kürzeste Wege von s nach q: die Anfangsstücke von π_1 und π_2. Das widerspräche Behauptung (i). Wenn von s ausgehende kürzeste Wege sich also einmal getrennt haben, können sie nie wieder zusammenkommen. Hieraus folgt ihre Baumgestalt.

Übungsaufgabe 7.8 Wenn der Punkt p schon einmal sichtbar war, gehört er – ebenso wie der Startpunkt s – zur aktuellen partiellen Karte P'. Dann enthält P' auch den kürzesten Weg π von s nach p in P. Würde nämlich π das Teilpolygon P' über eine Kante e, die nicht zum Rand von P gehört, verlassen und wieder betreten, könnte man diese Schlinge durch ein Stück von e ersetzen und dadurch π verkürzen. Also ist π auch der kürzeste Weg *in* P', der s mit p verbindet, und als solcher dem Roboter bekannt.

Übungsaufgabe 7.9 Der Roboter geht vom Startpunkt s zunächst 1 Meter nach rechts zum Punkt p_0 und folgt dann dem Kreis um s durch p_0, bis er sich wieder in p_0 befindet. Ab $j = 0$ werden dann folgende Anweisungen ausgeführt:

repeat
gehe von p_j um 2^j Meter nach rechts zu p_{j+1};
folge dem Kreis von s durch p_{j+1}, bis p_{j+1} wieder erreicht ist;
$j := j + 1$;
until Halbgerade H erreicht

Jedes p_i hat den Abstand 2^i zum Startpunkt s. Im schlimmsten Fall wird die Halbgerade H erst kurz vor Beendigung des Kreises durch p_{i+1} angetroffen, obwohl sie den Kreis durch p_i fast berührt; siehe Abbildung 7.33. Der kürzeste Weg von s nach H hat dann die Länge 2^i, während bei der Suche ein Weg der Länge

$$\begin{aligned} 2^0 + 2\pi 2^0 + \sum_{j=0}^{i} \left(2^j + 2\pi 2^{j+1}\right) &= 1 + 2\pi + (1 + 4\pi)(2^{i+1} - 1) \\ &< (1 + 4\pi)2^{i+1} \end{aligned}$$

entsteht. Damit ergibt sich ein kompetitiver Faktor von $(1 + 4\pi)2 = 27{,}13\ldots$

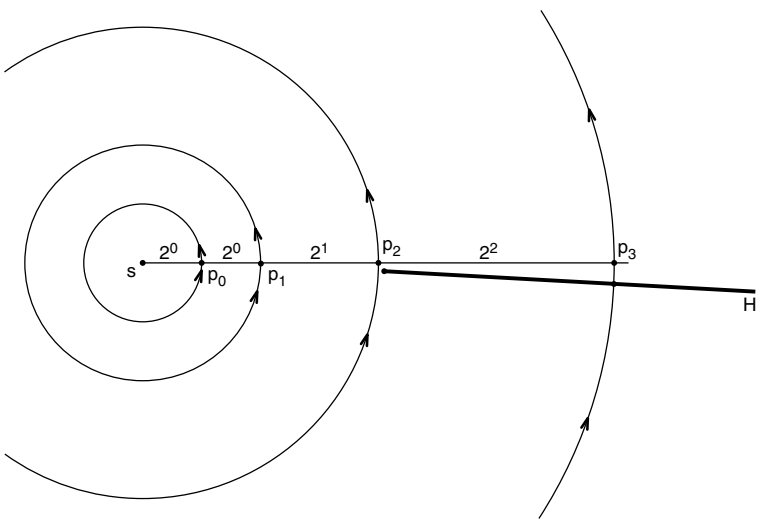


Abb. 7.33 Der Kreis durch p_i hat den Radius 2^i.

Übungsaufgabe 7.10 Am Punkt p_6 kann der Roboter die Ecke bei v_4 voll einsehen. Nur v_5 ist noch nicht einsehbar. Also wird der Weg mit einem Kreisbogenstück um v_5 fortgesetzt.

Übungsaufgabe 7.11 Sei ρ ein selbstnähernder Weg von s nach t. Dann gilt für alle Punkte p auf ρ die Abschätzung $|st| \geq |pt|$. Folglich ist ganz ρ im Kreis C um t durch s enthalten. Weil der Kreis C konvex ist, enthält er die konvexe Hülle $ch(\rho)$ von ρ. Nach Lemma 4.3 auf Seite 158 ist der Umfang von $ch(\rho)$ höchstens so lang wie der von C. Mit Lemma 7.24 ergibt sich daher die Abschätzung

$$\text{Länge}(\rho) \leq \text{Umfang}(ch(\rho)) \leq \text{Umfang}(C) = 2\pi|st|.$$

Literatur

[1] H. Abelson, A. A. diSessa. *Turtle Geometry*. MIT Press, Cambridge, 1980.

[2] S. Abramowski, H. Müller. *Geometrisches Modellieren*. BI-Wissenschaftsverlag, Mannheim, 1991.

[3] H. Alt, P. Braß, M. Godau, C. Knauer, C. Wenk. Computing the Hausdorff distance of geometric patterns and shapes. In B. Aronov, S. Basu, J. Pach, M. Sharir, editors, *Discrete and Computational Geometry – the Goodman-Pollack Festschrift*, volume 25 of *Algorithms and Combinatorics*, pages 65 – 76. Springer-Verlag, 2003.

[4] H. Alt, O. Cheong, A. Vigneron. The voronoi diagram of curved objects. Manuscript, 2004.

[5] H. Alt, C. K. Yap. Algorithmic aspect of motion planning: a tutorial, part 1. *Algorithms Rev.*, 1(1):43–60, 1990.

[6] H. Alt, C. K. Yap. Algorithmic aspect of motion planning: a tutorial, part 2. *Algorithms Rev.*, 1(2):61–77, 1990.

[7] B. Aronov, M. de Berg, A. F. van der Stappen, P. Švestka, J. Vleugels. Motion planning for multiple robots. *Discrete Comput. Geom.*, 22(4):505–525, 1999.

[8] G. Aumann, K. Spitzmüller. *Computerorientierte Geometrie*. BI-Wissenschaftsverlag, Mannheim, 1993.

[9] F. Aurenhammer. Voronoi diagrams: A survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, Sept. 1991.

[10] F. Aurenhammer, R. Klein. Voronoi diagrams. In J.-R. Sack, J. Urrutia, editors, *Handbook of Computational Geometry*, pages 201–290. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.

[11] R. Baeza-Yates, J. Culberson, G. Rawlins. Searching in the plane. *Inform. Comput.*, 106:234–252, 1993.

[12] I. J. Balaban. An optimal algorithm for finding segment intersections. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 211–219, 1995.

[13] M. Barner, F. Flohr. *Analysis II.* Walter de Gruyter, Berlin, 1983.

[14] A. Beck, D. J. Newman. Yet more on the linear search problem. *Israel Journal of Mathematics*, 8:419 – 429, 1970.

[15] J. L. Bentley. *Programming Pearls.* Addison-Wesley, Reading, MA, 1985.

[16] J. L. Bentley, T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28(9):643–647, Sept. 1979.

[17] J. L. Bentley, M. I. Shamos. Divide-and-conquer in multidimensional space. In *Proc. 8th Annu. ACM Sympos. Theory Comput.*, pages 220–230, 1976.

[18] J.-D. Boissonnat, M. Teillaud. On the randomized construction of the Delaunay tree. *Theoret. Comput. Sci.*, 112:339–354, 1993.

[19] J.-D. Boissonnat, M. Yvinec. *Algorithmic Geometry.* Cambridge University Press, UK, 1998. Translated by Hervé Brönnimann.

[20] B. Bollobás. *Extremal Graph Theory.* Academic Press, New York, 1978.

[21] A. Borodin, R. El-Yaniv. *Online Computation and Competitive Analysis.* Cambridge University Press, Cambridge, UK, 1998.

[22] I. N. Bronstein, K. A. Semendjajew, G. Musiol, H. Mühlig. *Taschenbuch der Mathematik.* Verlag Harry Deutsch, Frankfurt am Main, 5th edition, 2000.

[23] C. Burnikel, K. Mehlhorn, S. Schirra. On degeneracy in geometric computations. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 16–23, 1994.

[24] B. Chazelle. An optimal convex hull algorithm and new results on cuttings. In *Proc. 32nd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 29–38, 1991.

[25] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, 6(5):485–524, 1991.

[26] B. Chazelle. *The Discrepancy Method.* Cambridge University Press, 2000.

[27] B. Chazelle, H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *J. ACM*, 39(1):1–54, 1992.

[28] D. Cheriton, R. E. Tarjan. Finding minimum spanning trees. *SIAM J. Comput.*, 5:724–742, 1976.

[29] L. P. Chew, R. L. Drysdale, III. Voronoi diagrams based on convex distance functions. In *Proc. 1st Annu. ACM Sympos. Comput. Geom.*, pages 235–244, 1985.

[30] L. P. Chew, R. L. Drysdale, III. Finding largest empty circles with location constraints. Technical Report PCS-TR86-130, Dartmouth College, 1986.

[31] L. P. Chew, K. Kedem, M. Sharir, B. Tagansky, E. Welzl. Voronoi diagrams of lines in 3-space under polyhedral convex distance functions. In *Proc. 6th ACM-SIAM Sympos. Discrete Algorithms*, pages 197–204, 1995.

[32] V. Chvátal. A greedy heuristic for the set-covering problem. *Math. Oper. Res.*, 4:233–235, 1979.

[33] M. Cieliebak, P. Flocchini, G. Prencipe, N. Santoro, P. Widmayer. Solving the robots gathering problem. In *Proc. 30th Internat. Colloq. Automata Lang. Program.*, volume 2719 of *Lecture Notes Comput. Sci.*, pages 1181–1196. Springer-Verlag, 2003.

[34] K. L. Clarkson, P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.

[35] R. Cole, M. T. Goodrich. Optimal parallel algorithms for polygon and point-set problems. *Algorithmica*, 7:3–23, 1992.

[36] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 2nd edition, 2001.

[37] M. de Berg. *Efficient algorithms for ray shooting and hidden surface removal.* Ph.D. dissertation, Dept. Comput. Sci., Utrecht Univ., Utrecht, Netherlands, 1992.

[38] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Berlin, Germany, 2nd edition, 2000.

[39] F. Dehne, H. Noltemeier. Clustering methods for geometric objects and applications to design problems. *Visual Comput.*, 2(1):31–38, Jan. 1986.

[40] F. Dehne, J.-R. Sack. A survey of parallel computational geometry algorithms. In *Proc. International Workshop on Parallel Processing by Cellular Automata and Arrays*, volume 342 of *Lecture Notes Comput. Sci.*, pages 73–88. Springer-Verlag, 1988.

[41] E. D. Demaine, S. P. Fekete, S. Gal. Online searching with turn cost. *Theor. Comput. Sci.*, Submitted.

[42] X. Deng, T. Kameda, C. H. Papadimitriou. How to learn an unknown environment I: the rectilinear case. Technical Report CS-93-04, Department of Computer Science, York University, Canada, 1993.

[43] R. Descartes. *Principia Philosophiae.* Ludovicus Elzevirius, Amsterdam, 1644.

[44] O. Devillers, S. Meiser, M. Teillaud. Fully dynamic Delaunay triangulation in logarithmic expectedtime per operation. *Comput. Geom. Theory Appl.*, 2(2):55–80, 1992.

[45] G. Di Battista, P. Eades, R. Tamassia, I. G. Tollis. *Graph Drawing*. Prentice Hall, Upper Saddle River, NJ, 1999.

[46] C. Dierke. *Weltatlas*. Georg Westermann Verlag, Braunschweig, 1957.

[47] R. Diestel. *Graphentheorie*. Springer-Verlag, 2nd edition, 2000.

[48] H. Djidjev, A. Lingas. On computing the Voronoi diagram for restricted planar figures. In *Proc. 2nd Workshop Algorithms Data Struct.*, volume 519 of *Lecture Notes Comput. Sci.*, pages 54–64. Springer-Verlag, 1991.

[49] J.-M. Drappier. Enveloppes convexes. Technical report, Centre CPAO, ENSTA Palaiseau, 1983.

[50] G. Dudek, K. Romanik, S. Whitesides. Localizing a robot with minimum travel. In *Proc. 6th ACM-SIAM Sympos. Discrete Algorithms*, pages 437–446, 1995.

[51] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, West Germany, 1987.

[52] H. Edelsbrunner, R. Seidel. Voronoi diagrams and arrangements. *Discrete Comput. Geom.*, 1:25–44, 1986.

[53] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, S. Schönherr. On the design of CGAL a computational geometry algorithms library. *Softw. – Pract. Exp.*, 30(11):1167–1202, 2000.

[54] S. Fekete, R. Klein, A. Nüchter. Online searching with an autonomous robot. In *Proc. 6th Workshop Algorithmic Found. Robot.*, pages 335 – 350, 2004.

[55] A. Fiat, G. Woeginger, editors. *On-line Algorithms: The State of the Art*, volume 1442 of *Lecture Notes Comput. Sci.* Springer-Verlag, 1998.

[56] U. Finke, K. Hinrichs. Overlaying simply connected planar subdivisions in linear time. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 119–126, 1995.

[57] R. Fleischer, T. Kamphans, R. Klein, E. Langetepe, G. Trippen. Competitive online approximation of the optimal search ratio. In *Proc. 12th Annu. European Sympos. Algorithms*, volume 3221 of *Lecture Notes Comput. Sci.*, pages 335 – 346, Heidelberg, 2004. Springer-Verlag.

[58] M. Formann, T. Hagerup, J. Haralambides, M. Kaufmann, F. T. Leighton, A. Simvonis, E. Welzl, G. Woeginger. Drawing graphs in the plane with high resolution. *SIAM J. Comput.*, 22:1035–1052, 1993.

[59] S. J. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[60] S. Gal. Minimax solutions for linear search problems. *SIAM J. Appl. Math.*, 27:17 – 30, 1974.

[61] M. R. Garey, D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, NY, 1979.

[62] M. R. Garey, D. S. Johnson. Crossing number is NP-complete. *SIAM J. Algebraic Discrete Methods*, 4(3):312–316, 1983.

[63] J. E. Goodman, J. O'Rourke, editors. *Handbook of Discrete and Computational Geometry*, volume 27 of *Discrete Mathematics and Its Applications*. CRC Press LLC, Boca Raton, FL, 2nd edition, 2004.

[64] R. L. Graham, D. E. Knuth, O. Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading, MA, second edition, 1994.

[65] P. Gritzmann, R. Brandenberg. *Das Geheimnis des kürzesten Weges – Ein mathematisches Abenteuer*. Springer-Verlag, 3rd edition, 2005.

[66] J. Gudmundsson, H. J. Haverkort, M. van Kreveld. Constrained higher order Delaunay triangulations. In *Proc. 19th European Workshop Comput. Geom.*, pages 105 – 108, 2003.

[67] L. J. Guibas, R. Motwani, P. Raghavan. The robot localization problem in two dimensions. In *Proc. 3rd ACM-SIAM Sympos. Discrete Algorithms*, pages 259–268, 1992.

[68] L. J. Guibas, J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, 4(2):74–123, Apr. 1985.

[69] R. H. Güting, S. Dieker. *Datenstrukturen und Algorithmen*. B. G. Teubner, Stuttgart, 2nd edition, 2003.

[70] C. Gutwenger, M. Jünger, S. Leipert, P. Mutzel, M. Percan, R. Weiskircher. Advances in C-planarity testing of clustered graphs. In M. Goodrich, S. Kobourov, editors, *Proc. 10th Internat. Sympos. Graph Drawing*, volume 2528 of *Lecture Notes Comput. Sci.*, pages 220 – 325. Springer-Verlag, 2002.

[71] H. W. Hamacher. *Mathematische Lösungsverfahren für planare Standortprobleme*. Verlag Vieweg, Wiesbaden, 1995.

[72] A. Hemmerling. *Labyrinth Problems: Labyrinth-Searching Abilities of Automata*. B. G. Teubner, Leipzig, 1989.

[73] A. Hemmerling. Navigation without perception of coordinates and distances. *Mathematical Logic Quarterly*, 40:237–260, 1994.

[74] K. Hinrichs, J. Nievergelt, P. Schorn. Plane-sweep solves the closest pair problem elegantly. *Inform. Process. Lett.*, 26:255–261, 1988.

[75] F. Hoffmann, C. Icking, R. Klein, K. Kriegel. A competitive strategy for learning a polygon. In *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, pages 166–174, 1997.

[76] F. Hurtado, R. Klein, E. Langetepe, V. Sacristán. The weighted farthest color voronoi diagram on trees and graphs. *Computational Geometry: Theory and Applications*, 27:13–26, 2004.

[77] C. Icking, R. Klein. Competitive strategies for autonomous systems. In H. Bunke, T. Kanade, H. Noltemeier, editors, *Modelling and Planning for Sensor Based Intelligent Robot Systems*, pages 23–40. World Scientific, Singapore, 1995.

[78] C. Icking, R. Klein. Searching for the kernel of a polygon: A competitive strategy. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 258–266, 1995.

[79] C. Icking, R. Klein, E. Langetepe, S. Schuierer, I. Semrau. An optimal competitive strategy for walking in streets. *SIAM J. Comput.*, 33:462–486, 2004.

[80] C. Icking, R. Klein, N.-M. Lê, L. Ma. Convex distance functions in 3-space are different. *Fundam. Inform.*, 22:331–352, 1995.

[81] C. Icking, G. Rote, E. Welzl, C. Yap. Shortest paths for line segments. *Algorithmica*, 10:182–200, 1993.

[82] B. Joe. On the correctness of a linear-time visibility polygon algorithm. *Internat. J. Comput. Math.*, 32:155–172, 1990.

[83] B. Joe, R. B. Simpson. Correction to Lee's visibility polygon algorithm. *BIT*, 27:458–473, 1987.

[84] M. Kaufmann, D. Wagner, editors. *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes Comput. Sci.* Springer-Verlag, 2001.

[85] D. G. Kirkpatrick, R. Seidel. The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 15:287–299, 1986.

[86] R. Klein. *Concrete and Abstract Voronoi Diagrams*, volume 400 of *Lecture Notes Comput. Sci.* Springer-Verlag, 1989.

[87] R. Klein, K. Mehlhorn, S. Meiser. Randomized incremental construction of abstract Voronoi diagrams. *Comput. Geom. Theory Appl.*, 3(3):157–184, 1993.

[88] R. Klein, O. Nurmi, T. Ottmann, D. Wood. A dynamic fixed windowing problem. *Algorithmica*, 4:535–550, 1989.

[89] M. Laszlo. *Computational Geometry and Computer Graphics in C++*. Prentice Hall, Englewood Cliffs, NJ, 1996.

[90] D. T. Lee. Visibility of a simple polygon. *Comput. Vision Graph. Image Process.*, 22:207–221, 1983.

[91] D. T. Lee, F. P. Preparata. An optimal algorithm for finding the kernel of a polygon. *J. ACM*, 26(3):415–421, July 1979.

[92] D. T. Lee, C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Inform.*, 9:23–29, 1977.

[93] J.-H. Lee, K.-Y. Chwa. Tight analysis of a self-approaching strategy for the online kernel-search problem. *Inform. Process. Lett.*, 69:39–45, 1999.

[94] J.-H. Lee, C.-S. Shin, J.-H. Kim, S. Y. Shin, K.-Y. Chwa. New competitive strategies for searching in unknown star-shaped polygons. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 427–429, 1997.

[95] F. Lorenz. *Lineare Algebra I.* BI-Wissenschaftsverlag, Mannheim, 1982.

[96] F. Lorenz. *Einführung in die Algebra, Band 1.* BI-Wissenschaftsverlag, Mannheim, 1987.

[97] T. Lozano-Pérez. Spatial planning: A configuration space approach. *IEEE Trans. Comput.*, C-32:108–120, 1983.

[98] V. J. Lumelsky, A. A. Stepanov. Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2:403–430, 1987.

[99] J. Matoušek. *Geometric Discrepancy*, volume 18 of *Algorithms and Combinatorics*. Springer-Verlag, 1999.

[100] J. Matoušek. *Lectures on Discrete Geometry*, volume 212 of *Graduate Texts in Mathematics*. Springer-Verlag, 2002.

[101] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, Germany, 1984.

[102] K. Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, volume 2 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, Germany, 1984.

[103] K. Mehlhorn. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*, volume 3 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, Germany, 1984.

[104] K. Mehlhorn, S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 1999.

[105] S. Meiser. *Zur Konstruktion abstrakter Voronoidiagramme*. Ph.D. thesis, Dept. Comput. Sci., Univ. Saarlandes, Saarbrücken, Germany, 1993.

[106] J. S. B. Mitchell. Geometric shortest paths and network optimization. In J.-R. Sack, J. Urrutia, editors, *Handbook of Computational Geometry*, pages 633–701. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.

[107] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[108] J. I. Munro, M. H. Overmars, D. Wood. Variations on visibility. In *Proc. 3rd Annu. ACM Sympos. Comput. Geom.*, pages 291–299, 1987.

[109] G. Narasimhan, M. Smid. *Geometric Spanner Networks*. Cambridge University Press, to appear.

[110] J. Nievergelt, K. H. Hinrichs. *Algorithms and Data Structures: With Applications to Graphics and Geometry*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[111] C. Ó'Dúnlaing, C. K. Yap. A "retraction" method for planning the motion of a disk. *J. Algorithms*, 6:104–111, 1985.

[112] A. Okabe, B. Boots, K. Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons, Chichester, UK, 1992.

[113] J. O'Rourke. *Art Gallery Theorems and Algorithms*. The International Series of Monographs on Computer Science. Oxford University Press, New York, NY, 1987.

[114] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, 2nd edition, 1998.

[115] T. Ottmann, S. Schuierer, C. A. Hipke. Kompetitive Analyse für Online-Algorithmen: Eine kommentierte Bibliographie. Technical Report 61, Institut für Informatik, Universität Freiburg, Germany, 1994.

[116] T. Ottmann, P. Widmayer. *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag, Heidelberg, 4th edition, 2002.

[117] M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes Comput. Sci.* Springer-Verlag, Heidelberg, West Germany, 1983.

[118] J. Pach, P. K. Agarwal. *Combinatorial Geometry*. John Wiley & Sons, New York, NY, 1995.

[119] J. Pach, M. Sharir. On vertical visibility in arrangements of segments and the queue size in the Bentley-Ottmann line sweeping algorithm. *SIAM J. Comput.*, 20:460–470, 1991.

[120] F. P. Preparata, M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[121] H. Raynaud. Sur l'enveloppe convex des nuages de points aleatoires dans R^n. *J. Appl. Probab.*, 7:35–48, 1970.

[122] G. Rote. Curves with increasing chords. *Math. Proc. Camb. Phil. Soc.*, 115:1–12, 1994.

[123] J.-R. Sack, J. Urrutia, editors. *Handbook of Computational Geometry*. North-Holland, Amsterdam, 2000.

[124] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.

[125] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[126] P. Schorn. *Robust Algorithms in a Program Library for Geometric Computation*, volume 32 of *Informatik-Dissertationen ETH Zürich*. Verlag der Fachvereine, Zürich, 1991.

[127] F. Schulz. Zur Berechnung des Kerns von einfachen Polygonen. Diplomarbeit, FernUniversität Hagen, Fachbereich Informatik, 1996.

[128] J. T. Schwartz, C.-K. Yap, editors. *Algorithmic and Geometric Aspects of Robotics*, volume 1 of *Advances in Robotics*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1987.

[129] C. Schwarz, M. Smid, J. Snoeyink. An optimal algorithm for the on-line closest-pair problem. *Algorithmica*, 12:18–29, 1994.

[130] O. Schwarzkopf. Dynamic maintenance of geometric structures made easy. In *Proc. 32nd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 197–206, 1991.

[131] R. Seidel. Constrained Delaunay triangulations and Voronoi diagrams with obstacles. Technical Report 260, IIG-TU Graz, Austria, 1988.

[132] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom. Theory Appl.*, 1(1):51–64, 1991.

[133] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete Comput. Geom.*, 6:423–434, 1991.

[134] R. Seidel. Backwards analysis of randomized geometric algorithms. In J. Pach, editor, *New Trends in Discrete and Computational Geometry*, volume 10 of *Algorithms and Combinatorics*, pages 37–68. Springer-Verlag, 1993.

[135] M. I. Shamos, D. Hoey. Closest-point problems. In *Proc. 16th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 151–162, 1975.

[136] M. Sharir, P. K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications*. Cambridge University Press, New York, 1995.

[137] J. Stillwell. *Classical Topology and Combinatorial Group Theory.* Springer-Verlag, New York, 1993.

[138] C. Thomassen. The graph genus problem is NP-complete. *J. Algorithms*, 10:568 – 576, 1989.

[139] M. J. van Kreveld. *New Results on Data Structures in Computational Geometry.* Ph.D. dissertation, Dept. Comput. Sci., Utrecht Univ., Utrecht, Netherlands, 1992.

[140] J. J. Verkuil. Idual: Visualizing duality transformations. Technical report, Dept. Comput. Sci., Univ. Utrecht, 1992.

[141] F. F. Yao. Computational geometry. In R. A. Earnshaw, B. Wyvill, editors, *Algorithms in Complexity*, pages 345–490. Elsevier, Amsterdam, 1990.

Index

E

Y

Z