Kapitel 7

Geometrische Algorithmen

7.1 Einleitung

Die Geometrie ist eines der ältesten Gebiete der Mathematik, dessen Wurzeln bis in die Antike zurückreichen. Algorithmische Aspekte und die Lösung geometrischer Probleme mit Hilfe von Computern haben aber erst in jüngster Zeit verstärktes Interesse gefunden. Der Grund dafür liegt sicherlich in gewandelten Anforderungen durch neue Anwendungen, die von der Bildverarbeitung, Computer-Graphik, Geographie, Kartographie usw. bis hin zum physischen Entwurf höchstintegrierter Schaltkreise reichen.

So ist in den letzten Jahren ein neues Forschungsgebiet entstanden, das unter dem Namen "Algorithmische Geometrie" (Computational Geometry) inzwischen einen festen Platz innerhalb des Gebiets "Algorithmen und Datenstrukturen" einnimmt. Der Name "Algorithmische Geometrie" geht zurück auf eine im Jahre 1978 erschienene Dissertation von M. Shamos, vgl. [166]. Im CAD-Bereich und in der Computer-Graphik wurde der Begriff allerdings schon früher mit etwas anderer Bedeutung verwendet, vgl. hierzu [59].

Seit der Dissertation von Shamos ist eine wahre Flut von Arbeiten in diesem Gebiet erschienen. Es ist daher völlig unmöglich, die Hunderte von inzwischen untersuchten Problemen und erzielten Einzellösungen auch nur annähernd vollständig und systematisch darzustellen. Um eine bessere und vollständige Übersicht über das Gebiet zu erhalten, sollte der Leser die Bibliographie [43] mit über 600 Einträgen, die Übersichtsarbeit [], die Monographie [122] und die Bücher [149], [42] und [183] konsultieren.

Wir werden uns in diesem Kapitel auf die Darstellung einiger weniger, aber durchaus grundlegender Probleme, Datenstrukturen und Algorithmen beschränken. Im Abschnitt 7.2 stellen wir das Scan-line-Prinzip vor, das sich als Mittel zur Lösung zahlreicher geometrischer Probleme inzwischen bewährt hat. Wie das Divide-and-conquer-Prinzip zur Lösung geometrischer Probleme eingesetzt werden kann, zeigt Abschnitt 7.3. Zur Speicherung und Manipulation von Daten mit [illegible] räumlichen Komponente reichen die bekannten, zur Speicherung von Mengen ganzzahliger Schlüssel geeigneten Strukturen nicht aus. In Abschnitt 7.4 stellen wir einige Strukturen vor, die dafür in Frage kommen, und zwar Segment-, Intervall-, Bereichs- und Prioritäts-Suchbäume. In den Abschnitten 7.2, 7.3 und 7.4 haben wir es in der Regel mit Mengen

iso-orientierter Objekte in der Ebene zu tun, d.h. mit Mengen von Objekten, die zu rechtwinklig gewählten Koordinaten ausgerichtet sind. Beispiele sind Mengen horizontaler und vertikaler Liniensegmente in der Ebene oder Mengen von Rechtecken mit zueinander parallelen Seiten.

In Abschnitt 7.5 zeigen wir, wie Algorithmen, die für Mengen iso-orientierter Objekte entwickelt wurden, übertragen werden können auf Mengen beliebig orientierter Objekte. Das Verfahren ist auf solche Algorithmen anwendbar, die sich auf sogenannte "Skelettstrukturen" stützen, wie sie in Abschnitt 7.4 beschrieben werden. Die vielfältige Verwendbarkeit dieser Strukturen wird auch im Abschnitt 7.6 belegt. Dort werden ein Spezialfall eines Standardproblems aus der Computergraphik, das Hidden-line-Eliminationsproblem, und ein allgemeines Suchproblem behandelt. Eine insbesondere zur Lösung von geometrischen Nachbarschaftsanfragen nützliche Struktur, das sogenannte Voronoi-Diagramm, wird im Abschnitt 7.7 behandelt.

7.2 Das Scan-line-Prinzip

Geometrische Probleme treten in vielen Anwendungsbereichen auf. Wir wollen uns jedoch darauf beschränken, nur einen Anwendungsbereich exemplarisch etwas genauer zu betrachten und geometrische Probleme diskutieren, die beim Entwurf höchstintegrierter Schaltungen auftreten. Man kann den Entwurfsprozeß ganz grob in zwei Phasen einteilen, in die funktionelle und die physikalische Entwurfsphase. Ziel der zweiten Entwurfsphase ist schließlich die Herstellung der Fertigungsunterlagen (Masken) für die Chipproduktion. Vom Standpunkt der algorithmischen Geometrie aus betrachtet geht es hier darum, eine enorm große Anzahl von Rechtecken auf die verschiedenen Schichten (Diffusions-, Polysilikon-, Metall- usw. Schicht) so zu verteilen, daß die von ihnen repräsentierten Transistoren, Widerstände, Leitungen usw. die gewünschten Schaltfunktionen realisieren. Dabei sind zahlreiche Probleme zu lösen, die inhärent geometrischer Natur sind. Beispiele sind:

Das Überprüfen der geometrischen Entwurfsregeln (*design-rule checking*): Hier wird das Einhalten von durch die jeweilige Technologie vorgegebenen geometrischen Bedingungen, wie minimale Abstände, maximale Überlappungen usw., überprüft.

Schaltelement-Extraktion (*feature extraction*): Hier werden aus der geometrischen Erscheinungsform elektrische Schaltelemente und ihre Verbindungen untereinander extrahiert.

Plazierung und Verdrahtung: Die Schaltelemente müssen möglichst platzsparend und so angeordnet werden, daß die notwendigen (elektrischen) Verbindungen leicht herstellbar sind. Für diese beim VLSI-Design auftretenden geometrischen Probleme ist charakteristisch, daß die dabei vorkommenden geometrischen Objekte einfach sind, aber die Anzahl der zu verarbeitenden Daten sehr groß ist. Typisch ist eine Anzahl von 5 bis 10 Millionen iso-orientierter Rechtecke in der Ebene.

In der algorithmischen Geometrie hat man den iso-orientierten Fall besonders intensiv studiert, vgl. hierzu die Übersicht von Wood [195]. Das ist der Fall, in dem alle auftretenden Liniensegmente (also z.B. Rechteckseiten) und Linien parallel zu einer

der Koordinatenachsen verlaufen. Eine der leistungsfähigsten Techniken zur Lösung geometrischer Probleme, das sogenannte *Scan-line-Prinzip*, läßt sich in diesem Fall besonders einfach erklären und führt nicht selten zu optimalen Problemlösungen. Wir erklären dieses Prinzip jetzt genauer (vgl. auch [130]).

Gegeben sei eine Menge von achsenparallelen Objekten in der Ebene, z.B. eine Menge vertikaler und horizontaler Liniensegmente, eine Menge iso-orientierter Rechtecke oder eine Menge iso-orientierter, rechteckiger, einfacher Polygone. Die Idee ist nun, eine vertikale Linie (die sogenannte *Scan-line*) von links nach rechts (oder alternativ: eine horizontale Linie von oben nach unten) über die gegebene Menge zu schwenken, um ein die Menge betreffendes statisches, zweidimensionales geometrisches Problem in eine dynamische Folge eindimensionaler Probl[illegible]zu zerlegen. Die Scan-line teilt zu jedem Zeitpunkt die gegebene Menge von Objekten in drei disjunkte Teilmengen: die *toten* Objekte, die bereits vollständig von der Scan-line überstrichen wurden, die gerade *aktiven* Objekte, die gegenwärtig von der Scan-line geschnitten werden und die noch *inaktiven* (oder: *schlafenden*) Objekte, die erst künftig von der Scan-line geschnitten werden. Die Scan-line definiert eine lokale Ordnung auf der Menge der jeweils aktiven Objekte; sie muß gegebenenfalls den sich ändernden lokalen Verhältnissen angepaßt werden und kann für problemspezifische Aufgaben konsultiert werden. Während man die Scan-line über die Eingabeszene hinwegschwenkt, hält man eine dynamische, d.h. zeitlich veränderliche, problemspezifische *Vertikalstruktur* L aufrecht, in der man sich alle für das jeweils zu lösende Problem benötigten Daten merkt. Eine wichtige Beobachtung ist nun, daß man an Stelle eines kontinuierlichen Schwenks (englisch: sweep) die Scan-line in diskreten Schritten über die gegebene Szene führen kann. Sei Q die aufsteigend sortierte Menge aller x-Werte von Objekten. D.h.: Im Falle einer Menge von horizontalen Liniensegmenten ist Q die Menge der linken und rechten Endpunkte, im Falle einer Menge von Rechtecken ist Q die Menge aller linken und rechten Rechteckseiten, usw. Ganz allgemein wird Q gerade so gewählt, daß sich zwischen je zwei aufeinderfolgenden Punkten in Q weder die Zusammensetzung der Menge der gerade aktiven Objekte noch deren relative Anordnung (längs der Scan-line) ändern. Dann genügt es, Q als Menge der *Haltepunkte* der Scan-line zu nehmen. Statt eines kontinuierlichen Schwenks "springt" man mit der Scan-line von Haltepunkt zu Haltepunkt in aufsteigender x-Reihenfolge.

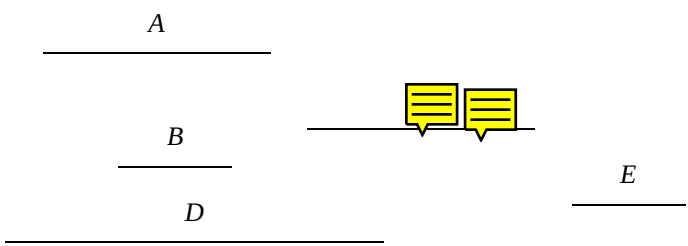
Ein vom jeweils zu lösenden Problem unabhängiger algorithmischer Rahmen für das Scan-line-Prinzip sieht also wie folgt aus:

Algorithmus *Scan-line-Prinzip*
{*liefert zu einer Menge iso-orientierter Objekte problemabhängige Antworten*}
$Q :=$ *objekt- und problemabhängige Folge von Haltepunkten in aufsteigender x-Reihenfolge*;
$L :=$ $\emptyset$; {*angeordnete Menge der jeweils aktiven Objekte*}
while Q *nicht leer* **do**
  begin
    wähle nächsten Haltepunkt aus Q und entferne ihn aus Q;
    $update(L)$ *und gib* (*problemabhängige*) *Teilantwort aus*
  end

Wir wollen das durch diesen Rahmen formulierte Scan-line-Prinzip jetzt auf drei konkrete Probleme anwenden.

7.2.1 Sichtbarkeitsproblem

Als einfachstes Beispiel für die Anwendung des Scan-line-Prinzips bringen wir die Lösung eines bei der Kompaktierung höchstintegrierter Schaltkreise auftretenden *Sichtbarkeitsproblems*. Zur Kompaktierung in y-Richtung müssen Abstandsbedingungen zwischen relevanten Paaren von (Schalt-) Elementen eingehalten werden. Dazu müssen die relevanten Paare zunächst einmal bestimmt werden. Hierzu genügt es, die beteiligten Elemente durch horizontale Liniensegmente darzustellen und die Menge aller Paare zu bestimmen, die sich sehen können (vgl.[163, 108]). Genauer: Zwei Liniensegmente s und s' in einer gegebenen Menge horizontaler Liniensegmente sind gegenseitig sichtbar, wenn es eine vertikale Gerade gibt, die s und s', aber kein weiteres Liniensegment der Menge zwischen s und s' schneidet. Wir betrachten ein Beispiel mit fünf Liniensegmenten A, B, C, D, E:



Die Menge der gegenseitig sichtbaren Paare besteht genau aus den (ungeordneten) Paaren (A,B), (A,D), (B,D), (C,D).

Natürlich könnte man sämtliche gegenseitig sichtbaren Paare in einer Menge von N Liniensegmenten dadurch bestimmen, daß man alle $N(N-1)/2$ Paare von Liniensegmenten betrachtet und für jedes Paar feststellt, ob es gegenseitig sichtbar ist oder nicht. Dieses *naive* Verfahren benötigt wenigstens $\Omega(N^2)$ Schritte. Es ist allerdings keineswegs offensichtlich, wie man für ein Paar von Segmenten schnell feststellt, ob es gegenseitig sichtbar ist oder nicht.

Andererseits kann es aber nur höchstens linear viele gegenseitig sichtbare Paare geben. Denn die Relation "ist gegenseitig sichtbar" läßt sich unmittelbar als ein planarer Graph auffassen: Die Knoten des Graphen sind die gegebenen Liniensegmente; zwei Liniensegmente sind durch eine Kante miteinander verbunden genau dann, wenn sie gegenseitig sichtbar sind. Da ein planarer Graph mit N Knoten aber höchstens $3N-6$ Kanten haben kann, folgt, daß es auch nur höchstens ebensoviele Paare gegenseitig sichtbarer Liniensegmente gibt.

Die Anwendung des Scan-line-Prinzips auf das Sichtbarkeitsproblem liefert folgenden Algorithmus:

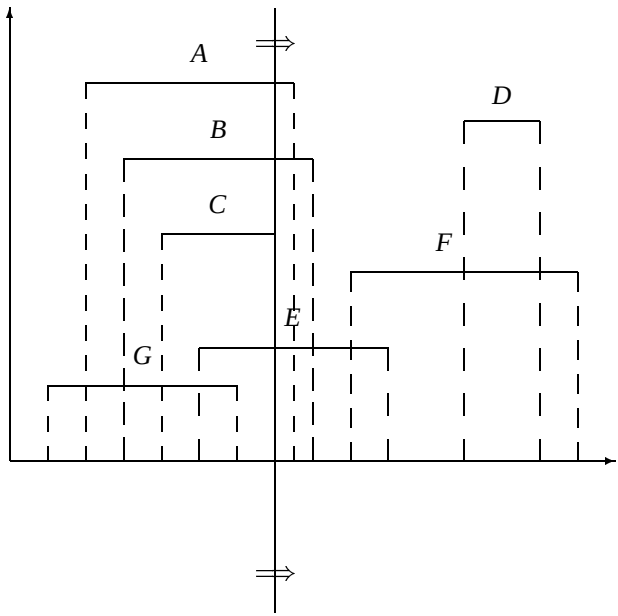
Algorithmus *Sichtbarkeit*
{*liefert zu einer Menge* $S = \{s_1, \ldots, s_N\}$ *horizontaler Liniensegmente in der Ebene die Menge aller Paare von gegenseitig sichtbaren Elementen in* S}
$Q :=$ *Folge der* $2N$ *Anfangs- und Endpunkte von Elementen in* S *in aufsteigender* x*-Reihenfolge*;
$L :=$ $\emptyset$; {*Menge der jeweils aktiven Liniensegmente in aufsteigender* y*-Reihenfolge*}
while Q *ist nicht leer* **do**
  begin
    $p :=$ *nächster* (*Halte*)*-Punkt von* Q;
    if p *ist linker Endpunkt eines Segments* s
      then
        begin
          füge s *in* L *ein*;
          bestimme die Nachbarn s' *und* s'' *von* s *in* L *und gib* (s,s') *und* (s,s'') *als Paare sichtbarer Elemente aus*
        end
      else {p *ist rechter Endpunkt eines Segments* s}
        begin
          bestimme die Nachbarn s' *und* s'' *von* s *in* L;
          entferne s *aus* L;
          gib (s',s'') *als Paar sichtbarer Elemente aus*
        end
  end {*while*}

Um die Formulierung des Algorithmus nicht unnötig zu komplizieren, haben wir stillschweigend einige Annahmen gemacht, die keine prinzipielle Bedeutung haben, d.h. insbesondere die asymptotische Effizienz des Verfahrens nicht beeinflussen. Wir nehmen an, daß sämtliche x-Werte von Anfangs- und Endpunkten sämtlicher Liniensegmente paarweise verschieden sind. Die Menge Q der Haltepunkte besteht also aus $2N$ verschiedenen Elementen. Wir setzen ferner voraus, daß die Bestimmung der Nachbarn eines Liniensegments in der nach aufsteigenden y-Werten geordneten Vertikalstruktur die Existenzprüfung einschließt. Wenn also beispielsweise ein Segment s keinen oberen, wohl aber einen unteren Nachbarn s'' hat, wird nach dem Einfügen von s in L nur das Paar (s,s'') ausgegeben. Bei der Implementation des Verfahrens für die Praxis muß man natürlich all diese Sonderfälle betrachten.

Abbildung 7.1 zeigt ein Beispiel für das Verfahren.

Die Menge L kann man als eine geordnete Menge von Punkten oder *Schlüsseln* auffassen, auf der die Operationen *Einfügen* eines Elementes, *Entfernen* eines Elementes und Bestimmen von Nachbarn, d.h. des *Vorgängers* und des *Nachfolgers* eines Elementes ausgeführt werden. Implementiert man L als balancierten Suchbaum, so kann man jede dieser Operationen in $O(\log n)$ Schritten ausführen, wenn n die maximale Anzahl der Elemente in L ist. Natürlich kann diese Anzahl niemals größer sein als die Gesamtzahl N der gegebenen horizontalen Liniensegmente. Für Entwurfsdaten (VLSI-Masken) als gegebener Menge von Liniensegmenten kann man erwarten, daß jeweils

höchstens $O(\sqrt{N})$ Objekte gerade aktiv sind. Dann benötigt man zur Speicherung von L nur $O(\sqrt{N})$ Platz. An jedem Haltepunkt müssen maximal vier der oben angegebenen Operationen ausgeführt werden; jede Operation benötigt höchstens Zeit $O(\log N)$. Insgesamt ergibt sich damit, daß man alle höchstens $3N-6$ Paare gegenseitig sichtbarer Liniensegmente in einer Menge von N horizontalen Liniensegmenten in Zeit $O(N \log N)$ und Platz $O(N)$ bestimmen kann, wenn man das Scan-line-Prinzip benutzt. Das ist offensichtlich besser als das naive Verfahren.



G	A	A	A	A	A	A
	G	B	B	B	B	B
		G	C	C	C	E
			G	E	E	
				G		

L am jeweiligen Haltepunkt (in aufsteigender y–Reihenfolge)

Ausgabe: $(A,G),(A,B),\ (B,G),(B,C),\ (C,G),(C,E),\ (E,G),(B,E)$

Abbildung 7.1

Wir haben bei der Analyse des Scan-line-Verfahrens zur Lösung des Sichtbarkeitsproblems für eine Menge von N Liniensegmenten stillschweigend angenommen, daß die Menge der Anfangs- und Endpunkte der Liniensegmente bereits in aufsteigender Reihenfolge etwa als Elemente des Arrays Q vorliegt. Denn wir haben den Aufwand für das Sortieren nicht mitgezählt. Weil der für das Sortieren notwendige Aufwand von

der Größenordnung $\Theta(N \log N)$ ist, hätte die Berücksichtigung dieses Aufwands am Ergebnis natürlich nichts verändert. Allerdings legt die stillschweigende Annahme folgende Frage nahe: Gibt es ein Verfahren zur Bestimmung aller höchstens $3N - 6$ Paare von gegenseitig sichtbaren Liniensegmenten in einer Menge von N Liniensegmenten, das in Zeit $O(N)$ ausführbar ist, wenn man annimmt, daß die Anfangs- und Endpunkte der Liniensegmente bereits aufsteigend sortiert gegeben sind? Mit Ausnahme einiger Spezialfälle ist diese Frage bis heute offen, vgl. [163].

Als nächstes Beispiel für die Anwendung des Scan-line-Prinzips behandeln wir die geometrische Grundaufgabe der Bestimmung aller Paare von sich schneidenden Liniensegmenten in der Ebene. Zunächst behandeln wir den iso-orientierten Fall dieses Problems und dann den allgemeinen Fall.

7.2.2 Das Schnittproblem für iso-orientierte Liniensegmente

Gegeben sei eine Menge von insgesamt N vertikalen und horizontalen Liniensegmenten in der Ebene. Gesucht sind alle Paare von sich schneidenden Segmenten. Dieses Problem nennen wir das rechteckige Segment-Schnitt-Problem, kurz RSS-Problem.

Natürlich können wir das RSS-Problem mit der naiven "brute-force"-Methode in $O(N^2)$ Schritten lösen, indem wir sämtliche Paare von Liniensegmenten daraufhin überprüfen, ob sie einen Schnittpunkt haben. Es ist nicht schwer, Beispiele zu finden, für die es kein wesentlich besseres Verfahren gibt. Man betrachte etwa die Menge von $N/2$ horizontalen und $N/2$ vertikalen Liniensegmenten in Abbildung 7.2.

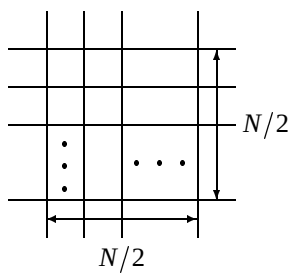


Abbildung 7.2

Hier gibt es $N^2/4$ Paare sich schneidender Segmente. Andererseits gibt es aber auch viele Fälle, in denen die Anzahl der Schnittpunkte klein ist und nicht quadratisch mit der Anzahl der gegebenen Segmente wächst. VLSI-Masken-Daten sind ein wichtiges Beispiel für diesen Fall. Deshalb ist man an Algorithmen interessiert, die in einem solchen Fall besser sind als das naive Verfahren. Wir zeigen jetzt, daß das Scan-line-Prinzip uns ein solches Verfahren liefert.

Zur Vereinfachung der Darstellung des Verfahrens nehmen wir an, daß alle Anfangs- und Endpunkte horizontaler Segmente und alle vertikalen Segmente paarweise verschiedene x-Koordinaten haben. Insbesondere können sich Segmente also nicht überlappen und Schnittpunkte kann es höchstens zwischen horizontalen und vertikalen Segmenten geben. Die Anwendbarkeit des Scan-line-Prinzips ergibt sich nun unmittelbar aus folgender Beobachtung: Merkt man sich beim Schwenken der Scan-line in der Vertikalstruktur L stets die gerade aktiven horizontalen Segmente und trifft man mit der Scan-line auf ein vertikales Segment s, so kann s höchstens Schnittpunkte mit den gerade aktiven horizontalen Segmenten haben. Damit erhalten wir:

Algorithmus *zur Lösung des RSS-Problems*
{*liefert zu einer Menge* $S = \{s_1, \ldots, s_N\}$ *von horizontalen und vertikalen Liniensegmenten in der Ebene die Menge aller Paare von sich schneidenden Segmenten in* S}

$Q :=$ *Menge der x-Koordinaten der Anfangs- und Endpunkte horizontaler Segmente und von vertikalen Segmenten in aufsteigender x-Reihenfolge*;
$L :=$ $\emptyset$; {*Menge der jeweils aktiven horizontalen Segmente in aufsteigender y-Reihenfolge*}
while *Q nicht leer* **do**
  begin
    $p :=$ *nächster* (*Halte*)*-Punkt von Q*;
      if *p ist linker Endpunkt eines horizontalen Segments s*
        then *füge s in L ein*
        else
          if *p ist rechter Endpunkt eines horizontalen Segments s*
            then *entferne s aus L*
            else
              {*p ist x-Wert eines vertikalen Segments s mit unterem Endpunkt* (p, y_u) *und oberem Endpunkt* (p, y_o)}
              bestimme alle horizontalen Segmente t aus L, deren y-Koordinate $y(t)$ *im Bereich* $y_u \leq y(t) \leq y_o$ *liegt und gib* (s,t) *als Paar sich schneidender Segmente aus*
  end {*while*}

Abbildung 7.3 zeigt ein Beispiel für die Anwendung des Verfahrens.

Wir können annehmen, daß Q als sortiertes Array der Länge höchstens $2N$ vorliegt. (Gegebenenfalls müssen die x-Werte der gegebenen Segmente zuvor in Zeit $O(N \log N)$ und Platz $O(N)$ sortiert werden.)

Die Menge L kann man auffassen als eine geordnete Menge von Elementen. Sie besteht genau aus den y-Werten der horizontalen Liniensegmente. Auf dieser Menge werden folgende Operationen ausgeführt: Einfügen eines neuen Elementes, Entfernen eines Elementes und Bestimmen aller Elemente, die in einen gegebenen Bereich $[y_u, y_o]$ fallen. Die letzte Operation nennt man eine *Bereichsanfrage* (englisch: *range query*).

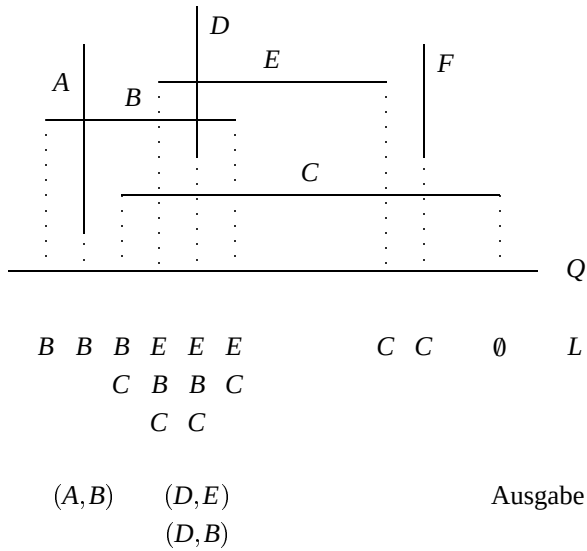


Abbildung 7.3

Eine naheliegende Möglichkeit zur Implementation von L besteht darin, die Elemente in aufsteigender Reihenfolge in den Blättern eines balancierten Blattsuchbaumes zu speichern. Verkettet man benachbarte Blätter zusätzlich doppelt, so kann man die Operationen Einfügen und Entfernen in $O(\log N)$ Schritten ausführen und Bereichsanfragen wie folgt beantworten: Um alle Elemente zu finden, die in einen gegebenen Bereich $[a,b]$ fallen, bestimmt man durch zwei aufeinanderfolgende Suchoperationen im Baum zunächst dasjenige Blatt mit kleinstem Wert größer gleich a und dasjenige Blatt mit größtem Wert kleiner gleich b. Dann läuft man der Kettung entlang und gibt die Elemente aus, die im Bereich $[a,b]$ liegen.

Abbildung 7.4 illustriert das Verfahren und die beschriebene Struktur.

Ist r die Anzahl der Elemente im Bereich $[a,b]$, so kann die Bereichsanfrage offenbar in Zeit $O(\log N + r)$ beantwortet werden.

Diese Implementation des Scan-line-Verfahrens zur Lösung des RSS-Problems erlaubt es also, sämtliche k Paare sich schneidender horizontaler und vertikaler Liniensegmente für eine gegebene Menge von N derartigen Segmenten in Zeit $O(N \log N + k)$ und Platz $O(N)$ zu berichten, wobei natürlich der Platz für die Antwort nicht mitgerechnet wird. Das Verfahren ist damit dem naiven Verfahren überlegen in allen Fällen, in denen k echt schwächer als quadratisch mit N wächst.

Man kann zeigen, vgl. [149], daß auch mindestens $\Omega(N \log N + k)$ Schritte erforderlich sind, um das RSS-Problem zu lösen.

Insgesamt folgt, daß das Scan-line-Verfahren zur Lösung des RSS-Problems zeit- und platzoptimal ist.

Wir überlegen uns nun, wie das Liniensegment-Schnittproblem gelöst werden kann, wenn die gegebene Menge nicht nur aus horizontalen und vertikalen Segmenten besteht.

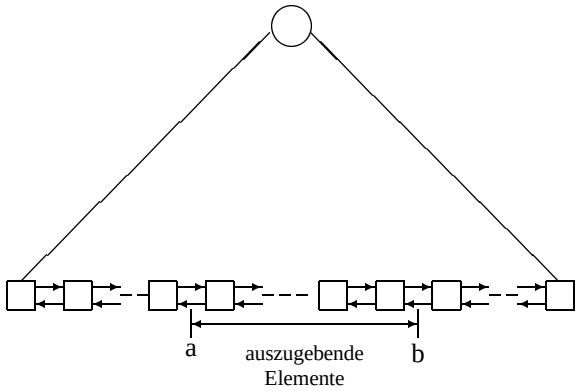


Abbildung 7.4

7.2.3 Das allgemeine Liniensegment-Schnittproblem

Für eine gegebene Menge von beliebig orientierten Liniensegmenten in der Ebene wollen wir die folgenden zwei Probleme lösen:

Schnittpunkttest: Stelle fest, ob es in der gegebenen Menge von N Segmenten wenigstens ein Paar sich schneidender Segmente gibt.

Schnittpunktaufzählung: Bestimme für eine gegebene Menge von N Liniensegmenten alle Paare sich schneidender Segmente.

Beide Probleme lassen sich natürlich auf die naive Weise in $O(N^2)$ Schritten lösen. Wir wollen zeigen, wie man beide Probleme mit Hilfe des Scan-line-Prinzips lösen kann. Um die Diskussion zahlreicher Sonderfälle vermeiden zu können, machen wir die Annahme, daß kein Liniensegment vertikal ist, daß sich in jedem Punkt höchstens zwei Liniensegmente schneiden und schließlich, daß alle Anfangs- und Endpunkte von Liniensegmenten paarweise verschiedene x-Koordinaten haben.

Anders als für eine Menge horizontaler Liniensegmente kann man für eine Menge beliebig orientierter Liniensegmente nur eine lokal gültige Ordnungsrelation "ist oberhalb von" wie folgt definieren. Seien A und B zwei Liniensegmente. Dann heißt *A x-oberhalb von B*, $A \uparrow_x B$, wenn die vertikale Gerade durch x sowohl A als auch B schneidet und der Schnittpunkt von x und A oberhalb des Schnittpunktes von x und B liegt. Im Beispiel von Abbildung 7.5 ist $C \uparrow_x B$, $A \uparrow_x C$ und $A \uparrow_x B$. Für jedes feste x ist $\uparrow_x$ offenbar eine Ordnungsrelation.

Zur Lösung des Schnittpunkttestproblems schwenken wir nun eine vertikale Scanline von links nach rechts über die N gegebenen Liniensegmente. An jeder Stelle x sind die Liniensegmente, die von der Scan-line geschnitten werden, durch $\uparrow_x$ vollständig geordnet. Änderungen der Ordnung sind möglich, wenn die Scan-line auf den linken oder rechten Endpunkt eines Segments trifft, und ferner, wenn die Scan-line einen Schnittpunkt passiert.

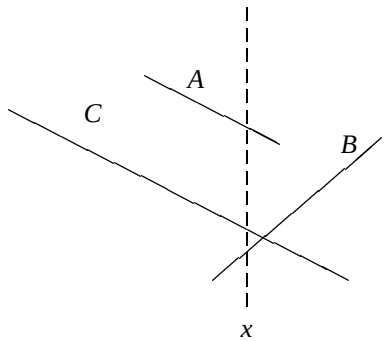


Abbildung 7.5

Für zwei beliebige Segmente A und B gilt: Wenn A und B sich schneiden, dann gibt es eine Stelle x links vom Schnittpunkt, so daß A und B in der Ordnung $\uparrow_x$ unmittelbar aufeinanderfolgen. (Hier machen wir von der Annahme Gebrauch, daß sich höchstens zwei Segmente in einem Punkt schneiden können!) Wenn wir also für je zwei Segmente A und B prüfen, ob sie sich schneiden, sobald sie an einer Stelle x bzgl. $\uparrow_x$ unmittelbar benachbart sind, können wir sicher sein, keinen Schnittpunkt zu übersehen, wenn es überhaupt einen gibt.

Diese Idee führt zu folgendem Algorithmus zur Lösung des Schnittpunkttestproblems:

Algorithmus *zur Lösung des Schnittpunkttestproblems*
{*liefert zu einer Menge* $S = \{s_1, \ldots, s_N\}$ *von Liniensegmenten in der Ebene "ja", falls es ein Paar sich schneidender Segmente in S gibt, und "nein" sonst*}

$Q :=$ *Folge der* $2N$ *Anfangs- und Endpunkte von Elementen in S in aufsteigender x-Reihenfolge*;
$L :=$ $\emptyset$; {*Menge der jeweils aktiven Liniensegmente in* $\uparrow_x$*-Ordnung*}
gefunden := *false*;
while (*Q ist nicht leer*) **and not** *gefunden* **do**
  begin
    p := *nächster Haltepunkt von Q*; {*p habe x-Koordinate p.x*}
      if *p ist linker Endpunkt eines Segments s*
        then
          begin
            füge s entsprechend der an der Stelle p gültigen Ordnung $\uparrow_{p.x}$ *in L ein*;
            bestimme den Nachfolger s' *und den Vorgänger* s'' *von s in L bzgl.* $\uparrow_{p.x}$;
            if $(s \cap s' \neq \emptyset)$ **or** $(s \cap s'') \neq \emptyset$
              then *gefunden* := *true*
          end

```
            else {p ist rechter Endpunkt eines Segments s}
              begin
                bestimme den Nachfolger s' und den Vorgänger s''
                  von s bzgl. der an der Stelle p gültigen Ordnung ↑_{p.x};
                entferne s aus L;
                if s' ∩ s'' ≠ ∅
                  then gefunden := true
              end
        end; {while}
    if gefunden
        then write('ja')
        else write('nein')
```

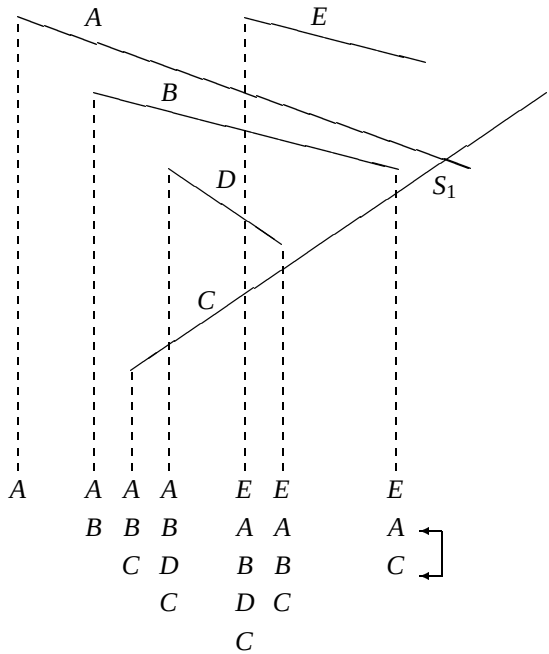
Wir haben hier wieder stillschweigend angenommen, daß die Bestimmung des Nachfolgers oder Vorgängers eines Elements die Existenzprüfung einschließt. Es ist leicht zu sehen, daß L an jeder Halteposition x der Scan-line die gerade aktiven Liniensegmente in korrekter $\uparrow_x$-Anordnung enthält. Das Verfahren muß also einen Schnittpunkt finden, falls es überhaupt einen gibt. Das muß nicht notwendig der am weitesten links liegende Schnittpunkt zweier Segmente in S sein.

Wir verfolgen zwei Beispiele anhand der Abbildung 7.6.

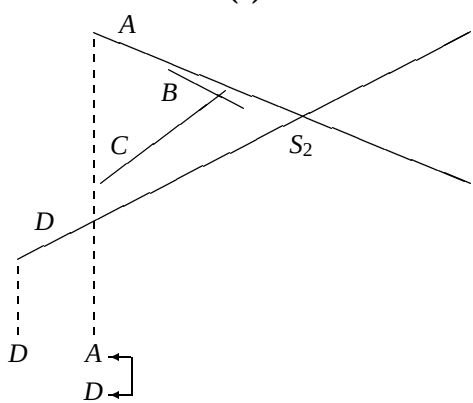
Im Fall (a) hält das Verfahren mit der Antwort "ja", sobald der Schnittpunkt S_1 von A und C gefunden wurde; im Fall (b) findet das Verfahren den Schnittpunkt S_2 von A und D bereits am zweiten Haltepunkt.

Die $2N$ Endpunkte der gegebenen Menge von Liniensegmenten können in $O(N \log N)$ Schritten nach aufsteigenden x-Werten sortiert werden. L kann man als balancierten Suchbaum implementieren. Dann kann jede der an einem Haltepunkt auszuführenden Operationen Einfügen, Entfernen, Bestimmen des Vorgängers und Nachfolgers eines Elementes in $O(\log N)$ Schritten ausgeführt werden. Damit folgt insgesamt, daß man mit Hilfe des Scan-line-Verfahrens in Zeit $O(N \log N)$ und Platz $O(N)$ feststellen kann, ob N Liniensegmente in der Ebene wenigstens einen Schnittpunkt haben oder nicht.

Was ist zu tun, um nicht nur festzustellen, ob in der gegebenen Menge von Liniensegmenten wenigstens ein Paar sich schneidender Segmente vorkommt, sondern um alle Paare sich schneidender Segmente aufzuzählen? Dann dürfen wir den oben angegebenen Algorithmus zur Lösung des Segmentschnittproblems nicht beenden, sobald der erste Schnittpunkt gefunden wurde. Vielmehr setzen wir das Verfahren fort und sorgen dafür, daß die die lokale Ordnung der jeweils gerade aktiven Segmente repräsentierende Vertikalstruktur L auch dann korrekt bleibt, wenn die Scan-line einen Schnittpunkt passiert: Immer wenn die Scan-line den Schnittpunkt s zweier Segmente A und B passiert, wechseln A und B ihren Platz in der unmittelbar links und rechts vom Schnittpunkt gültigen lokalen "oberhalb-von"-Ordnung. Wir müssen also die Scan-line nicht nur an allen Anfangs- und Endpunkten von Liniensegmenten anhalten, sondern auch an allen während des Hinüberschwenkens gefundenen Schnittpunkten. Ein Schnittpunkt liegt stets rechts von der Position der Scan-line, an der er entdeckt wurde. Wir fügen also einfach jeden gefundenen Schnittpunkt in die nach aufsteigenden x-Werten geordnete Schlange der Haltepunkte ein, wenn er sich nicht schon dort befindet.



(a)



(b)

Abbildung 7.6

Algorithmus *zur Lösung des Schnittpunktaufzählungsproblems*
{liefert zu einer Menge $S = \{s_1, \ldots, s_N\}$ *von Liniensegmenten in der Ebene alle Paare* (s_i, s_j) *mit:* $s_i, s_j \in S,\ s_i \cap s_j \neq \emptyset$ *und* $i \neq j$*}*

$Q :=$ *nach aufsteigenden x-Werten angeordnete Prioritäts-Schlange der Haltepunkte; anfangs initialisiert als Folge der* $2N$ *Anfangs- und Endpunkte von Elementen in S in aufsteigender x-Reihenfolge*;
$L :=$ $\emptyset$; *{Menge der jeweils aktiven Segmente in* $\uparrow_x$*-Ordnung}*
while *Q ist nicht leer* **do**
  begin
    $p := min(Q)$;
    minentferne(Q);
    if *p ist linker Endpunkt eines Segments s*
      then
        begin
          Einfügen(s, L);
          $s' :=$ *Nachfolger*(s, L);
          $s'' :=$ *Vorgänger*(s, L);
          if $s \cap s' \neq \emptyset$
            then *Einfügen*$(s \cap s', Q)$;
          if $s \cap s'' \neq \emptyset$
            then *Einfügen*$(s \cap s'', Q)$
        end
      else
        if *p ist rechter Endpunkt eines Segments s*
          then
            begin
              $s' :=$ *Nachfolger*(s, L);
              $s'' :=$ *Vorgänger*(s, L);
              if $s' \cap s'' \neq \emptyset$
                then *Einfügen*$(s' \cap s'', Q)$;
              Entfernen(s, L)
            end
          else *{p ist Schnittpunkt von* s' *und* s''*, d.h.* $p = s' \cap s''$*, und es sei* s' *oberhalb von* s'' *in L}*
            begin
              gib das Paar (s', s'') *mit Schnittpunkt p aus*;
              vertausche s' *und* s'' *in L*;
              {jetzt ist s'' *oberhalb von* s'*}*
              $t'' :=$ *Vorgänger*(s'', L);
              if $s'' \cap t'' \neq \emptyset$
                then *Einfügen*$(s'' \cap t'', Q)$;
              $t' :=$ *Nachfolger*(s', L);
              if $s' \cap t' \neq \emptyset$
                then *Einfügen*$(s' \cap t', Q)$
            end
end *{while}*

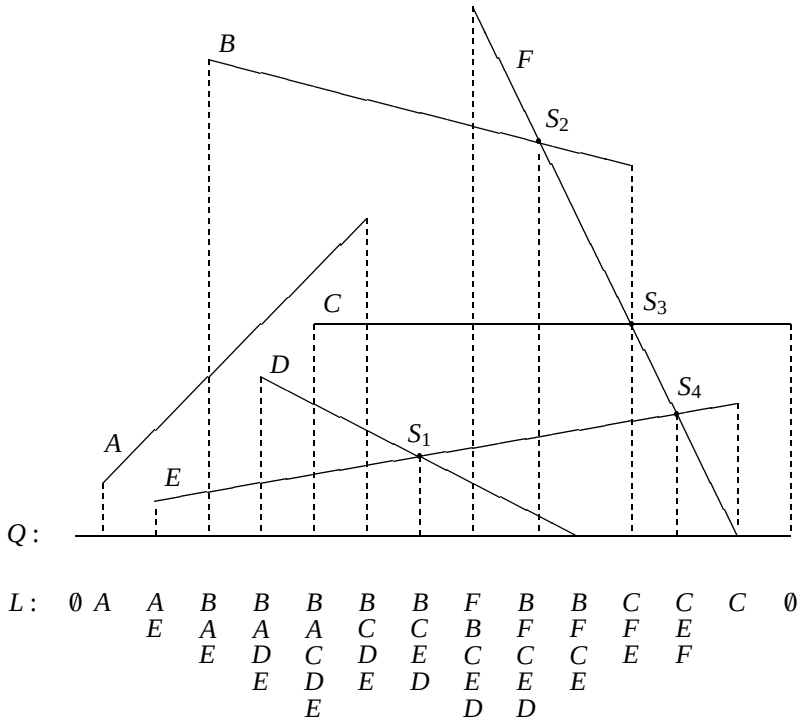


Abbildung 7.7

Um die Formulierung des Verfahrens nicht unnötig zu komplizieren, haben wir nicht nur angenommen, daß keine zwei Anfangs- und Endpunkte von Segmenten dieselbe x-Koordinate haben, sondern auch vorausgesetzt, daß kein Schnittpunkt dieselbe x-Koordinate wie ein Anfangs- oder Endpunkt eines Liniensegmentes hat. Unter dieser Annahme tritt an jeder Halteposition der Scan-line genau eines von drei möglichen Ereignissen ein: Ein Liniensegment beginnt, ein Liniensegment endet, oder es liegt ein Schnittpunkt zweier Liniensegmente vor. In der Realität ist diese Annahme natürlich selten erfüllt und auch nicht notwendig. Man kann beispielsweise vorschreiben, daß bei gleichzeitigem Vorliegen mehrerer Ereignisse am gleichen Haltepunkt $p \in Q$ die verschiedenen Ereignisse wie oben angegeben entsprechend ihren jeweiligen y-Koordinaten abgearbeitet werden.

Abbildung 7.7 zeigt ein Beispiel für das soeben beschriebene Verfahren.

Beim beschriebenen Verfahren kann es vorkommen, daß ein- und derselbe Schnittpunkt mehrere Male gefunden wird (vgl. etwa Abbildung 7.6 (b)), bei der S_2 zweimal gefunden wird). Damit jeder Schnittpunkt aber nur einmal in Q vermerkt wird, lassen wir dem Einfügen eines Schnittpunkts S in Q eine Suche nach S in Q vorangehen; S wird dann nur bei erfolgloser Suche eingefügt. Neben der Suche nach einem beliebigen Element muß Q also das Einfügen eines beliebigen Elements, die Bestimmung eines

Elements mit kleinstem x-Wert und das Entfernen eines Elements mit kleinstem x-Wert unterstützen. Organisieren wir Q etwa als balancierten Binärbaum, z.B. als AVL-Baum, so kann die notwendige Initialisierung in $O(N \log N)$ Schritten durchgeführt werden. Die Größe des Baums ist stets beschränkt durch die Gesamtzahl der Anfangs-, End- und Schnittpunkte von Liniensegmenten. Das sind höchstens $O(N + N^2) = O(N^2)$. Daher kann man die erforderlichen Operationen stets in $O(\log(N^2)) = O(\log N)$ Schritten ausführen.

Auf der Vertikalstruktur L werden die Operationen Einfügen und Entfernen eines Elementes, Bestimmen des Vorgängers und Nachfolgers eines Elementes und das Vertauschen zweier Elemente ausgeführt. Ohne daß dies im Algorithmus explizit angegeben wird, sind alle diese Operationen abhängig von der am jeweiligen Punkt $p \in Q$ gültigen Ordnung $\uparrow_{p.x}$. Es ist klar, daß L als nach dieser Ordnung sortierter balancierter Suchbaum so implementiert werden kann, daß jede der genannten Operationen in $O(\log N)$ Schritten ausführbar ist, weil L höchstens N Elemente enthält. Nehmen wir nun an, daß es k Schnittpunkte gibt. Dann wird die **while**-Schleife genau $2N + k$ mal durchlaufen. Wir haben bereits gesehen, daß jede Operation auf Q innerhalb der **while**-Schleife in $O(\log(2N + k)) = O(\log N)$ und jede Operation auf L in $O(\log N)$ Schritten ausführbar ist. Bei $2N + k$ Durchläufen werden also insgesamt höchstens $O((N + k) \log N)$ Schritte benötigt.

Man kann also mit Hilfe des Scan-line-Verfahrens alle k Schnittpunkte von N gegebenen Liniensegmenten in der Ebene in $O((N + k) \log N)$ Schritten finden. Das ist besser als das naive Verfahren für nicht zu große k. Chazelle [28] hat zeigen können, daß man mit geschickter Anwendung der im nächsten Abschnitt 7.3 vorgestellten Divide-and-conquer-Technik zu Algorithmen kommt, die das Schnittpunktaufzählungsproblem in $O(N \log^2 N + k)$ bzw. $O(N \log^2 N / \log\log N + k)$ Schritten lösen. Schließlich konnten Chazelle und Edelsbrunner [29] zeigen, daß alle k Schnitte wie im iso-orientierten Fall in $O(N \log N + k)$ Schritten gefunden werden können.

Die von uns skizzierte Implementierung des Scan-line-Verfahrens zur Bestimmung aller k Schnittpunkte einer gegebenen Menge von N Liniensegmenten hat allerdings einen Speicherbedarf von $\Omega(N^2)$ im schlechtesten Fall. Denn Q kann bis zu $2N + k = \Omega(N^2)$ Elemente enthalten. Der Speicherbedarf für Q und damit der Gesamtspeicherbedarf läßt sich jedoch auf $O(N)$ drücken, wenn man wie folgt vorgeht: Man fügt nicht jeden an einer Halteposition $p \in Q$ gefundenen Schnittpunkt in Q ein. Vielmehr sichert man lediglich, daß Q auf jeden Fall den von der jeweils aktuellen Position p der Scanline aus nächsten Schnittpunkt enthält. Dazu nimmt man für jedes aktive Liniensegment s höchstens einen Schnittpunkt in Q auf, nämlich unter allen Schnittpunkten, an denen s beteiligt ist und die man bis zu einer bestimmten Position entdeckt hat, den jeweils am weitesten links liegenden. Mit anderen Worten: Findet man im Verlauf des Verfahrens für ein Segment s einen weiteren Schnittpunkt, an dem s beteiligt ist, und liegt dieser links vom vorher gefundenen Schnittpunkt, so entfernt man den früher gefundenen Schnittpunkt und fügt den neuen in Q ein. Es ist nicht schwer zu sehen, daß man Q so implementieren kann, daß Q nur $O(N)$ Speicherplatz benötigt und alle auf Q auszuführenden Operationen in Zeit $O(\log N)$ ausführbar sind. (Ein balancierter Suchbaum leistet auch hier das Verlangte.) Um für jedes aktive Segment s leicht feststellen zu können, ob schon ein Schnittpunkt in Q ist, an dem s beteiligt ist, und welchen x-Wert dieser Schnittpunkt hat, kann man beispielsweise einen Zeiger von s auf diesen Schnitt-

punkt in Q verwenden. Diese Idee zur Reduktion des Speicherbedarfs geht zurück auf M. Brown [24].

7.3 Geometrisches Divide-and-conquer

Eines der leistungsfähigsten Prinzipien zur algorithmischen Lösung von Problemen ist das Divide-and-conquer-Prinzip. Wir haben bereits im Abschnitt 1.2.2 eine problemunabhängige Formulierung dieses Prinzips angegeben. Wir folgen hier der Darstellung aus [71].

Wenn wir versuchen, dieses Prinzip auf ein geometrisches Problem, wie das im vorigen Abschnitt behandelte Schnittproblem für iso-orientierte Liniensegmente, anzuwenden, stellt sich sofort die Frage: Wie soll man teilen? Eine Aufteilung ohne jede Beachtung der geometrischen Nachbarschaftsverhältnisse scheint wenig sinnvoll. Denn man möchte ja gerade besonderen Nutzen daraus ziehen, daß Schnitte im wesentlichen lokal, also zwischen räumlich nahen Segmenten auftreten. Versucht man aber eine Aufteilung etwa durch eine vertikale Gerade in eine linke und rechte Hälfte, so kann man im allgemeinen nicht verhindern, daß ausgedehnte geometrische Objekte, wie Liniensegmente, Rechtecke, Polygone usw., durchschnitten werden. Einen Ausweg aus dieser Schwierigkeit bietet das Prinzip der getrennten Repräsentation geometrischer Objekte. Wir erläutern dieses Prinzip im Abschnitt 7.3.1 für eine Menge horizontaler Liniensegmente bei Aufteilung durch eine vertikale Gerade und lösen das Schnittproblem für iso-orientierte Liniensegmente nach dem Divide-and-conquer-Prinzip. Im Abschnitt 7.3.2 zeigen wir, wie man Inklusions- und Schnittprobleme für Mengen iso-orientierter Rechtecke in der Ebene nach diesem Prinzip löst.

7.3.1 Segmentschnitt mittels Divide-and-conquer

Um eine gegebene Menge von N vertikalen und horizontalen Liniensegmenten in der Ebene leicht und eindeutig durch eine vertikale Gerade in eine linke und rechte Hälfte teilen zu können, benutzen wir eine getrennte Repräsentationhorizontaler Segmente: Jedes horizontale Segment wird durch das Paar seiner Endpunkte repräsentiert. Anstatt mit einer Menge von vertikalen und horizontalen Segmenten operieren wir mit einer Menge von vertikalen Segmenten und Punkten. Beispielsweise repräsentieren wir die Menge von sieben Segmenten in Abbildung 7.8 durch die Menge von vier Segmenten und sechs Punkten in Abbildung 7.9.

Dabei bezeichnen wir für ein horizontales Segment h den linken Endpunkt von h mit $.h$ und den rechten Endpunkt von h mit $h.$. Wenn wir zur Vereinfachung der Präsentation die Annahme machen, daß keine zwei vertikalen Segmente und Anfangs- oder Endpunkte horizontaler Segmente dieselbe x-Koordinate haben, kann man das Divide-and-conquer-Verfahren zur Lösung des Schnittproblems für eine (getrennt repräsentierte) Menge von iso-orientierten Liniensegmenten in der Ebene wie folgt formulieren:

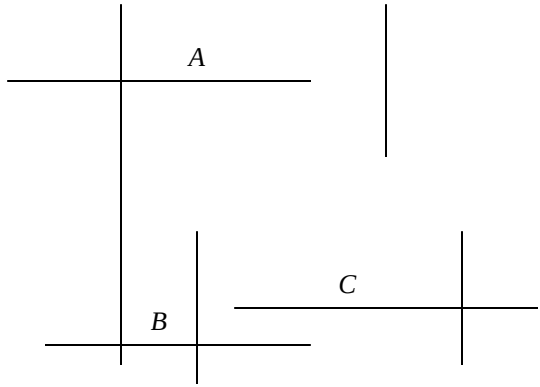


Abbildung 7.8

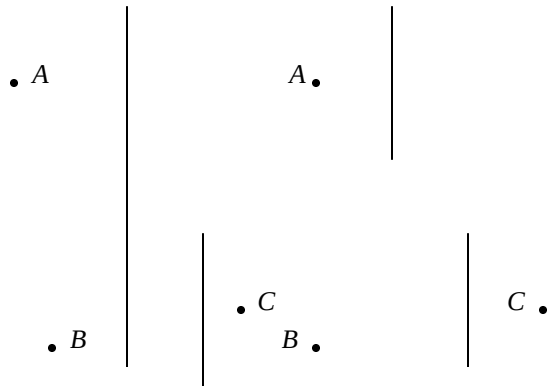
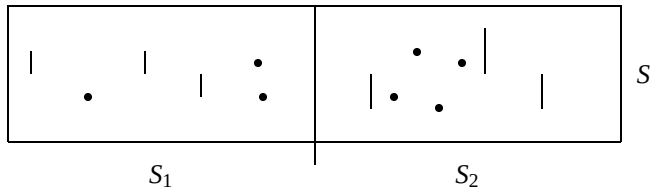


Abbildung 7.9

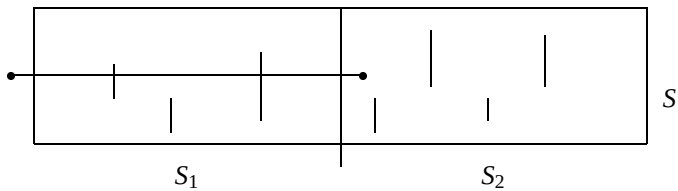
Algorithmus *ReportCuts*(S)
{*liefert zu einer Menge S von vertikalen Liniensegmenten und linken und rechten Endpunkten horizontaler Liniensegmente in der Ebene in getrennter Repräsentation alle Paare von sich schneidenden vertikalen Segmenten in S und horizontalen Segmenten mit linkem oder rechtem Endpunkt in S*}

1. Divide: *Teile S* (*durch eine vertikale Gerade*) *in eine linke Hälfte* S_1 *und eine rechte Hälfte* S_2, *falls S mehr als ein Element enthält; sonst enthält S kein sich schneidendes Paar:*

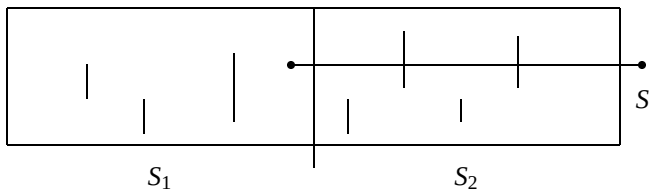


2. Conquer: *ReportCuts*(S_1); *ReportCuts*(S_2);
{*alle Schnitte in* S_1 *oder* S_2 *zwischen Paaren von Segmenten, die wenigstens einmal repräsentiert sind, sind bereits berichtet*}

3. Merge: *Berichte alle Schnitte zwischen vertikalen Segmenten in* S_1 *und horizontalen Segmenten mit rechtem Endpunkt in* S_2, *deren linker Endpunkt nicht in* S_1 *oder* S_2 *vorkommt:*



Berichte alle Schnitte zwischen vertikalen Segmenten in S_2 *und horizontalen Segmenten mit linkem Endpunkt in* S_1, *deren rechter Endpunkt nicht in* S_1 *oder* S_2 *vorkommt:*



Ende *des Algorithmus ReportCuts*

Ein Aufruf des Verfahrens *ReportCuts*(S) für eine gegebene Menge S bewirkt, daß das Verfahren wiederholt für immer kleinere, durch fortgesetzte Aufteilung entstehende Mengen aufgerufen wird, bis es schließlich für Mengen mit nur einem Element abbricht. Für die durch fortgesetzte Aufteilung entstehenden Mengen ist es möglich, daß nur der linke, nicht aber der rechte Endpunkt eines horizontalen Segments oder nur der rechte, nicht aber der linke Endpunkt auftritt. Das macht es erforderlich, sogleich das ganze Verfahren für Mengen dieser Art zu formulieren, so wie es oben geschehen ist. Wir zeigen nun die *Korrektheit* des Verfahrens und benutzen dazu die bereits als Kommentar zum Verfahren angegebene *Rekursionsinvariante*.

Ist S eine Menge von vertikalen Segmenten und linken oder rechten Endpunkten von horizontalen Segmenten, so sind nach Beendigung eines Aufrufs von *ReportCuts*(S) alle Schnitte zwischen vertikalen Segmenten in S und solchen horizontalen Segmenten berichtet, deren linker oder rechter Endpunkt (eventuell auch beide) in S vorkommt. Offenbar gilt diese Bedingung trivialerweise, wenn S nur aus einem einzigen Element besteht. In diesem Fall bricht das Verfahren *ReportCuts* ab; Schnitte werden nicht berichtet.

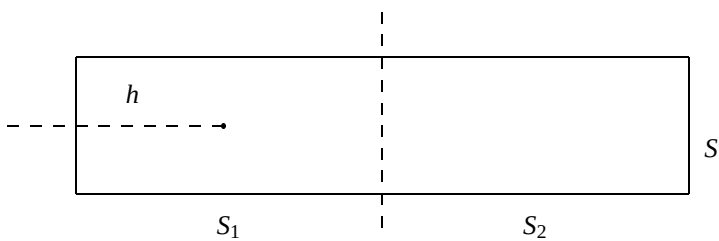
Wir zeigen jetzt: Wird S beim Aufruf von *ReportCuts*(S) aufgeteilt in eine linke Hälfte S_1 und eine rechte S_2 und gilt die Rekursionsinvariante bereits für S_1 und S_2, so gilt sie auch für S.

Dazu betrachten wir ein beliebiges horizontales Segment h, dessen linker oder rechter Endpunkt in S vorkommt. Wir müssen zeigen, daß nach Beendigung des Aufrufs *ReportCuts*(S) alle Schnitte von h mit vertikalen Segmenten in S berichtet worden sind. Folgende Fälle sind möglich:

Fall 1: Beide Endpunkte von h liegen in S_1. Da die Rekursionsinvariante nach Annahme für S_1 gilt, folgt, daß nach Beendigung des Aufrufs *ReportCuts*(S_1) im Conquer-Schritt alle Schnitte von h mit vertikalen Elementen in S_1 berichtet sind. h kann keine weiteren Schnitte mit vertikalen Segmenten in S haben.

Im *Fall 2*, daß beide Endpunkte von h in S_2 liegen, gilt das Analoge für Schnitte zwischen h und vertikalen Segmenten in S_2.

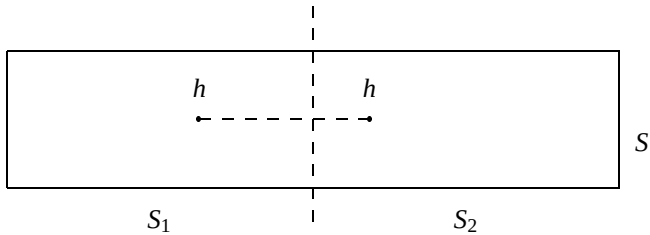
Fall 3: Nur der rechte Endpunkt von h ist in S_1.



Von den vertikalen Segmenten in S kann h nur solche schneiden, die in S_1 vorkommen. Diese sind aber nach dem Aufruf von *ReportCuts*(S_1) bereits berichtet, da nach Annahme die Rekursionsinvariante für S_1 gilt.

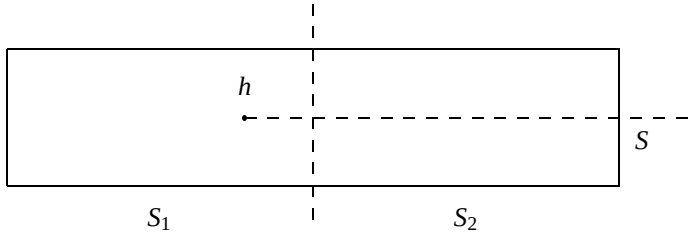
Im *Fall 4*, daß nur der linke Endpunkt von h in S_2 liegt, gilt das Analoge nach Beendigung des Aufrufs *ReportCuts*(S_2).

Fall 5: Der linke Endpunkt von h liegt in S_1 und der rechte Endpunkt von h in S_2:



Da die Rekursionsinvariante für S_1 und S_2 gilt, folgt, daß nach Beendigung des Aufrufs *ReportCuts*(S_1) und *ReportCuts*(S_2) alle möglichen Schnitte von h mit vertikalen Segmenten in S bereits berichtet sind.

Fall 6: Der linke Endpunkt von h liegt in S_1, aber der rechte Endpunkt von h liegt weder in S_1 noch in S_2:



h kann Schnitte mit vertikalen Segmenten in S_1 und S_2 haben. Die Gültigkeit der Rekursionsinvariante für S_1 sichert, daß nach Beendigung des Aufrufs *ReportCuts*(S_1) alle Schnitte von h mit vertikalen Segmenten in S_1 bereits berichtet sind. Es genügt also, im Merge-Schritt alle Schnitte zwischen h und vertikalen Segmenten in S_2 zu bestimmen, um alle Schnitte von h mit vertikalen Segmenten in S zu berichten.

Der *Fall 7*, daß der rechte Endpunkt von h in S_2, aber der linke Endpunkt von h weder in S_1 noch in S_2 liegt, ist völlig symmetrisch zum Fall 6. Auch hier haben wir den Merge-Schritt gerade so eingerichtet, daß alle möglichen Schnitte zwischen h und vertikalen Segmenten in S berichtet werden.

Insgesamt ist die Gültigkeit der Rekursionsinvariante für S damit nachgewiesen. Für eine möglichst effiziente Implementation des Verfahrens kommt es darauf an, die Schnitte im Merge-Schritt schnell und möglichst mit einem zur Anzahl dieser Schnitte proportionalen Aufwand zu bestimmen. Dazu dienen drei Mengen $L(S)$, $R(S)$ und $V(S)$, die wir jeder Menge S zuordnen:

$$L(S) = \{y(h) \mid h \text{ ist horizontales Liniensegment mit: } .h \in S \text{ aber } h. \notin S\}$$

$$R(S) = \{y(h) \mid h \text{ ist horizontales Liniensegment mit: } .h \notin S \text{ aber } h. \in S\}$$

$$V(S) = \text{Menge der durch die vertikalen Segmente in } S \text{ definierten } y\text{-Intervalle} = \{[y_u(v), y_o(v)] \mid v \text{ ist vertikales Liniensegment in } S\}$$

In diesen Definitionen haben wir mit $y(h)$ die y-Koordinate eines horizontalen Segmentes h bezeichnet und mit $y_u(v)$ bzw. $y_o(v)$ die untere bzw. obere y-Koordinate eines vertikalen Segmentes v.

Nehmen wir an, daß wir vor Beginn des Merge-Schrittes die Mengen

$$L(S_i), \quad R(S_i), \quad V(S_i), \quad i = 1,2$$

bereits kennen. Dann kann man den Merge-Schritt auch so formulieren:
Bestimme alle Paare (h,v) mit (a) oder (b):

$$\text{(a)} \qquad \begin{array}{l} y(h) \in R(S_2) \setminus L(S_1), \\ [y_u(v), y_o(v)] \in V(S_1), \\ y_u(v) \le y(h) \le y_o(v) \end{array}$$

$$\text{(b)} \qquad \begin{array}{l} y(h) \in L(S_1) \setminus R(S_2), \\ [y_u(v), y_o(v)] \in V(S_2), \\ y_u(v) \le y(h) \le y_o(v) \end{array}$$

Aus $L(S_i)$, $R(S_i)$, $V(S_i)$, $i = 1,2$, erhält man die $S = S_1 \cup S_2$ zugeordneten Mengen offenbar wie folgt:

$$\begin{array}{rcl} L(S) & := & (L(S_1) \setminus R(S_2)) \cup L(S_2) \\ R(S) & := & (R(S_2) \setminus L(S_1)) \cup R(S_1) \\ V(S) & := & V(S_1) \cup V(S_2) \end{array}$$

Falls S nur aus einem einzigen Element besteht, können wir diese Mengen leicht wie folgt initialisieren:

Fall 1: $S = \{.h\}$, d.h. S enthält nur den linken Endpunkt eines horizontalen Segments h.

$$L(S) := \{y(h)\};\ R(S) := \emptyset;\ V(S) := \emptyset$$

Fall 2: $S = \{h.\}$, d.h. S enthält nur den rechten Endpunkt eines horizontalen Segments h.

$$L(S) := \emptyset;\ R(S) := \{y(h)\};\ V(S) := \emptyset$$

Fall 3: $S = \{v\}$, d.h. S enthält nur das vertikale Segment v.

$$L(S) := \emptyset;\ R(S) := \emptyset;\ V(S) := \{[y_u(v), y_o(v)]\}$$

Zur Implementation des Verfahrens speichern wir nun die gegebene Menge S von vertikalen Segmenten und linken und rechten Endpunkten horizontaler Segmente in einem nach aufsteigenden x-Werten sortierten Array. Dann kann das Teilen im Divide-Schritt in konstanter Zeit ausgeführt werden. Die einer Menge S zugeordneten Mengen $L(S)$ und $R(S)$ implementieren wir als nach aufsteigenden y-Werten sortierte, verkettete lineare Listen. $V(S)$ wird ebenfalls als nach unteren Endpunkten, also nach y_u-Werten sortierte, verkettete lineare Liste implementiert.

$L(S)$, $R(S)$ und $V(S)$ können dann aus den $L(S_i)$, $R(S_i)$, $V(S_i)$, $i = 1,2$, zugeordneten Listen in $O(|S|)$ Schritten gebildet werden, indem man die bereits vorhandenen Listen ähnlich wie beim Sortieren durch Verschmelzen parallel durchläuft. Schließlich kann man im Merge-Schritt alle r Paare (h,v), die die oben angegebenen Bedingungen (a) oder (b) erfüllen, mit Hilfe dieser Listen bestimmen in $O(|S|+r)$ Schritten.

Bezeichnen wir mit $T(N)$ die Anzahl der Schritte, die erforderlich ist, um das Verfahren *ReportCuts*(S) bei dieser Implementation für eine Menge S mit N Elementen auszuführen, wenn wir den Aufwand für das Sortieren von S und die Ausgabe nicht mitrechnen, so gilt folgende Rekursionsformel:

$$T(N) = \underbrace{O(1)}_{\text{Divide}} + \underbrace{2T(\frac{N}{2})}_{\text{Conquer}} + \underbrace{O(N)}_{\text{Merge}}$$

und $T(1) = O(1)$.

Es ist wohlbekannt, daß diese Rekursionsformel die Lösung $O(N \log N)$ hat. Rechnen wir noch den Aufwand zur Ausgabe der insgesamt k Paare sich schneidender Segmente hinzu, so erhalten wir (inklusive Sortieraufwand):

Alle k Paare sich schneidender horizontaler und vertikaler Liniensegmente in einer gegebenen Menge von N derartigen Segmenten kann man mit Hilfe eines Divide-and-conquer-Verfahrens in Zeit $O(N \log N + k)$ und Platz $O(N)$ bestimmen.

Das ist dieselbe Zeit- und Platz-Komplexität, die auch das im vorigen Abschnitt besprochene Scan-line-Verfahren zur Lösung dieses Schnittproblems hat. Vergleicht man die Implementationen beider Verfahren, so fällt auf, daß das Divide-and-conquer-Verfahren mit einfachen Datenstrukturen auskommt: Verkettete, aufsteigend sortierte lineare Listen genügen. Im Falle des Scan-line-Verfahrens haben wir zu Bereichs-Suchbäumen modifizierte, balancierte Suchbäume benutzt.

7.3.2 Inklusions- und Schnittprobleme für Rechtecke

Das Divide-and-conquer-Prinzip läßt sich zur Lösung zahlreicher weiterer geometrischer Probleme benutzen, wenn man es zugleich mit dem Prinzip der getrennten Repräsentation der gegebenen geometrischen Objekte verbindet. Wir skizzieren kurz, wie man das Punkteinschluß- und das Rechteckschnittproblem in der Ebene auf diese Weise lösen kann.

Das Punkteinschluß-Problem für eine gegebene Menge von Rechtecken und Punkten in der Ebene ist das Problem, alle Paare (Punkt, Rechteck) zu bestimmen, für die das Rechteck den Punkt einschließt. Für das in Abbildung 7.10 angegebene Beispiel ist also die Antwort (p,A), (q,A), (r,A), (q,B), (r,B), (s,B), (s,C).

Um eine gegebene Menge von Punkten und Rechtecken in der Ebene eindeutig in eine linke und eine rechte Hälfte zerlegen zu können, wählen wir zunächst eine getrennte Repräsentation für die Rechtecke: Jedes Rechteck wird durch seinen linken und seinen rechten Rand repräsentiert. Eine Menge von Rechtecken und Punkten wird also repräsentiert durch eine Menge von vertikalen Liniensegmenten und Punkten. Nun kann man einen Algorithmus *ReportInc* analog zum Algorithmus *ReportCuts* wie folgt entwerfen:

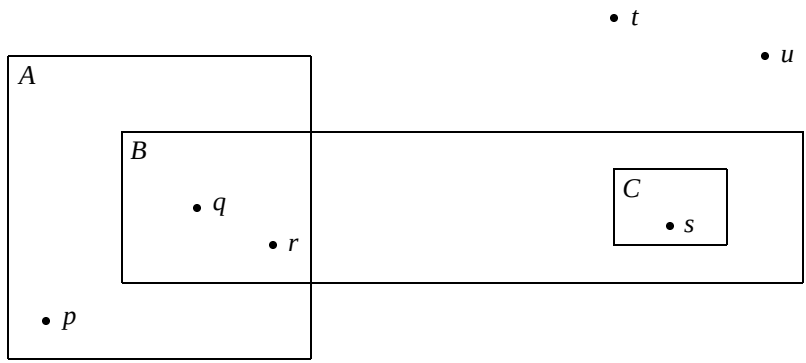


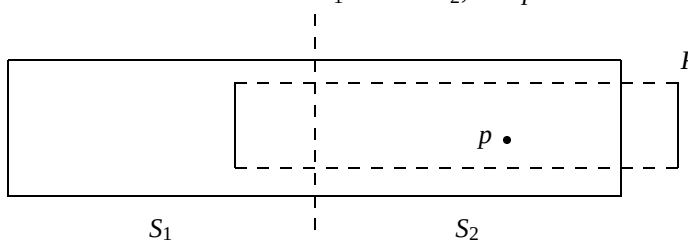
Abbildung 7.10

Algorithmus *ReportInc*(S)
{*liefert zu einer Menge S von linken und rechten Rändern von Rechtecken (in getrennter Repräsentation) und Punkten in der Ebene alle Paare* (p,R) *von Punkten p und Rechtecken R mit Rand in S mit* $p \in R$}

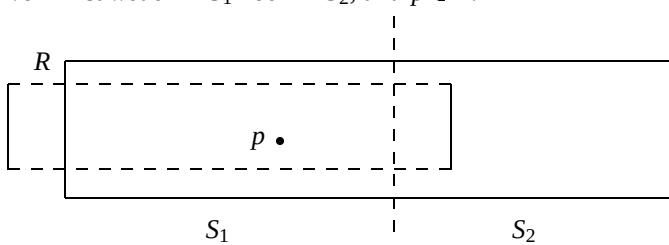
1. Divide: *Teile S (durch eine vertikale Gerade) in eine linke Hälfte* S_1 *und eine rechte Hälfte* S_2*, falls S mehr als ein Element enthält; falls S nur aus einem Element besteht, ist nichts zu berichten;*

2. Conquer: *ReportInc*(S_1); *ReportInc*(S_2);

3. Merge: *Berichte alle Paare* (p,R) *mit:* $p \in S_2$*, der linke Rand von R ist in* S_1*, aber der rechte Rand von R ist weder in* S_1 *noch in* S_2*, und* $p \in R$:



Berichte alle Paare (p,R) *mit:* $p \in S_1$*, der rechte Rand von R ist in* S_2*, aber der linke Rand von R ist weder in* S_1 *noch in* S_2*, und* $p \in R$*:*



Ende *des Algorithmus ReportInc*

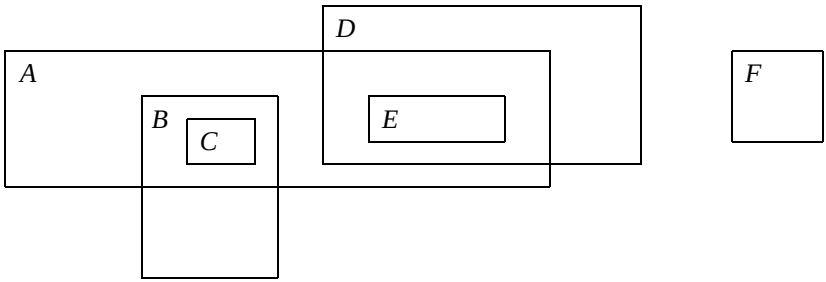


Abbildung 7.11

Der Nachweis der Korrektheit verläuft genauso wie im Falle des Algorithmus *ReportCuts* im vorigen Abschnitt:
Man zeigt, daß nach Ausführung eines Aufrufs *ReportInc*(S) für eine Menge von Punkten und (Rechtecke repräsentierenden) vertikalen Segmenten gilt: Alle Paare (p,R) von Inklusionen zwischen einem Punkt p und einem Rechteck R sind berichtet, für jeden Punkt p aus S und jedes Rechteck R, das in S wenigstens einmal (also: durch seinen linken oder rechten Rand oder durch beide) repräsentiert ist.

Für eine effiziente Implementation des Verfahrens kommt es offenbar darauf an, die im Merge-Schritt benötigten Mengen vertikaler Segmente effizient zu bestimmen, die eine linke (bzw. rechte) Rechteckseite in S_1 (bzw. S_2) repräsentieren, deren korrespondierende rechte (bzw. linke) Rechteckseite aber weder in S_1 noch in S_2 vorkommt. Das kann man ähnlich wie im Falle des Algorithmus *ReportCuts* im Abschnitt 7.3.1 machen und sichern, daß diese Mengen in konstanter Zeit initialisiert und in linearer Zeit im Merge-Schritt konstruiert werden können. Damit reduziert sich die im Merge-Schritt des Algorithmus *ReportInc* zu lösende Aufgabe auf das Problem, für eine nach unteren Endpunkten sortierte Menge von Intervallen und eine aufsteigend sortierte Menge von Punkten alle Paare (Punkt, Intervall) zu bestimmen, für die das Intervall den Punkt enthält. Es ist leicht zu sehen, daß das in einer Anzahl von Schritten möglich ist, die proportional zur Anzahl der Intervalle und Punkte und der Größe der Antwort ist. Insgesamt folgt:

Für eine Menge S von N Rechtecken und Punkten in der Ebene kann man alle k Paare (p,R) mit: p Punkt in S, R Rechteck in S und $p \in R$ mit Hilfe des Divide-and-conquer-Prinzips berichten in Zeit $O(N \log N + k)$ und Platz $O(N)$.

Die im Abschnitt 7.3.1 angegebene Lösung des rechteckigen Segmentschnittproblems und die hier skizzierte Lösung des Punkteinschlußproblems liefern zugleich auch eine Lösung des Rechteckschnittproblems für eine Menge iso-orientierter Rechtecke in der Ebene. Das ist das Problem, für eine gegebene Menge solcher Rechtecke alle Paare sich schneidender Rechtecke zu berichten. Dabei ist mit Rechteckschnitt sowohl Kantenschnitt als auch Inklusion gemeint.

Für das in Abbildung 7.11 angegebene Beispiel ist die gesuchte Antwort also die Menge:

$$\{(A,B),\quad (A,C),\quad (A,E),\quad (A,D),\quad (B,C),\quad (E,D)\}$$

Zur Lösung des Rechteckschnittproblems bestimmt man zunächst mit Hilfe des Verfahrens aus Abschnitt 7.3.1 alle Paare von Rechtecken, die sich an einer Kante schneiden. Dann wählt man für jedes der Rechtecke einen dieses Rechteck repräsentierenden Punkt, z.B. den Mittelpunkt, und bestimmt für die Menge aller Rechtecke und so erhaltenen Punkte alle Inklusionen von Punkten in Rechtecken. Das liefert alle Paare von Rechtecken, die sich vollständig einschließen (und außerdem manche, die sich schneiden). Insgesamt kann man auf diese Weise alle k Paare von sich schneidenden Rechtecken in einer Menge von N iso-orientierten Rechtecken in Zeit $O(N \log N + k)$ und Platz $O(N)$ bestimmen.

Wir bemerken abschließend, daß man das Rechteckschnittproblem auch direkt nach dem Divide-and-conquer-Prinzip lösen kann, ohne einen Umweg zu machen über das rechteckige Segmentschnitt- und das Punkteinschlußproblem. Weitere Beispiele für die Anwendung des Divide-and-conquer-Prinzips zur Lösung geometrischer Probleme findet man in [71] und [73].

7.4 Geometrische Datenstrukturen

Ganzzahlige Schlüssel kann man auffassen als Punkte auf der Zahlengeraden, also als nulldimensionale geometrische Objekte. Für sie ist charakteristisch, daß sie auf natürliche Weise geordnet sind. Eine große Vielfalt an Datenstrukturen zur Speicherung von Schlüsselmengen steht zur Auswahl. Je nachdem welche Operationen auf den Schlüsselmengen ausgeführt werden sollen, können wir Strukturen wählen, die die gewünschten Operationen besonders gut unterstützen. Zur Lösung der in den Abschnitten 7.2 und 7.3 behandelten geometrischen Probleme, des Sichtbarkeitsproblems und verschiedener Schnittprobleme für Liniensegmente in der Ebene, reichten die bekannten Strukturen aus. Es ist uns jedesmal gelungen, das geometrische Problem auf die Manipulation geeignet gewählter Schlüsselmengen zu reduzieren.

Schon für Mengen von Punkten in der Ebene, erst recht für ausgedehnte geometrische Objekte, wie Liniensegmente, Rechtecke usw., reichen die bekannten Strukturen nicht mehr aus, wenn man typisch geometrische Operationen unterstützen möchte. Solche Operationen sind z.B.: Für eine gegebene Menge von Punkten in der Ebene und einen gegebenen, zweidimensionalen Bereich, berichte alle Punkte, die in den gegebenen Bereich fallen. Oder: Für eine gegebene Menge von Liniensegmenten in der Ebene und ein gegebenes Segment, berichte alle Segmente der Menge, die das gegebene Segment schneidet. Wir wollen in diesem Abschnitt einige neue, inhärent geometrische Datenstrukturen kennenlernen und zeigen, wie sie zur Lösung einer geometrischen Grundaufgabe benutzt werden können. Als Beispiel wählen wir das Rechteckschnittproblem. Das ist das Problem, für eine gegebene Menge von Rechtecken alle Paare sich schneidender Rechtecke zu finden. Im Abschnitt 7.4.1 zeigen wir zunächst, wie das Problem mit Hilfe des Scan-line-Prinzips gelöst werden kann und welche Anforderungen an für eine Lösung geeignete Datenstrukturen zu stellen sind. In den folgenden Abschnitten besprechen wir dann im einzelnen Segment-Bäume, Intervall-Bäume und Prioritäts-Suchbäume, die sämtlich zur Lösung des Rechteckschnittproblems geeignet sind. Die-

se Datenstrukturen müssen nicht nur typisch geometrische Operationen unterstützen, wie sie zur Lösung des Rechteckschnittproblems verwendet werden. Sie müssen auch das Einfügen und Entfernen geometrischer Objekte erlauben. Wir entwerfen alle drei Strukturen nach demselben Prinzip als halbdynamische, sogenannte Skelettstrukturen: Anstatt Strukturen zu benutzen, deren Größe sich der Menge der jeweils vorhandenen geometrischen Objekte voll dynamisch anpaßt, schaffen wir zunächst ein anfänglich leeres Skelett über einem diskreten Raster, das allen im Verlauf des Scan-line-Verfahrens benötigten Objekten Platz bietet. Dieses Vorgehen hat nicht nur den Vorzug größerer Einfachheit und Einheitlichkeit, es bietet auch die Basis für die Übertragung der in diesem Abschnitt für Mengen iso-orientierter Objekte entwickelten Verfahren auf nicht-iso-orientierte Objekte im Abschnitt 7.5.

7.4.1 Reduktion des Rechteckschnittproblems

Sei eine Menge von N iso-orientierten Rechtecken in der Ebene gegeben, d.h. alle linken und rechten und alle oberen und unteren Rechteckseiten sind zueinander parallel. Um nicht zahlreiche Sonderfälle diskutieren zu müssen, nehmen wir an, daß zwei Rechteckseiten höchstens einen Punkt gemeinsam haben können, und ferner, daß alle oberen und unteren Rechteckseiten paarweise verschiedene y-Koordinaten haben. Die Lösung des *Rechteckschnittproblems* verlangt, alle Paare sich schneidender Rechtecke zu berichten. "Rechteckschnitt" umfaßt dabei sowohl Kantenschnitt als auch Inklusion. Gerade das Entdecken aller Inklusionen erfordert zusätzlichen Aufwand. Denn um alle Paare von Rechtecken zu finden, die sich an einer Kante schneiden, können wir einfach das Scan-line-Verfahren zur Lösung des Schnittproblems für Mengen horizontaler und vertikaler Liniensegmente nehmen. Anstatt nun — wie im Falle der Anwendung des Divide-and-conquer-Prinzips, vgl. Abschnitt 7.3.2 — nur die Rechteckinklusionen mit Hilfe des Scan-line-Verfahrens zu bestimmen, wenden wir das Scan-line-Prinzip direkt auf das Rechteckschnittproblem an.

Wir schwenken eine horizontale Scan-line von oben nach unten über die gegebene Menge von Rechtecken. Dabei merken wir uns in einer Horizontalstruktur L stets die gerade aktiven Rechtecke, genauer die Schnitte der jeweils aktiven Rechtecke mit der Scan-line, also eine Menge von (horizontalen) Intervallen. Jedesmal, wenn wir auf einen oberen Rand eines Rechtecks treffen, bestimmen wir alle Intervalle in L, die sich mit dem oberen Rand überlappen. Das sind genau die Intervalle, die zu gerade aktiven Rechtecken gehören, die einen nichtleeren Durchschnitt mit R haben. Außerdem müssen wir in L ein neues Intervall einfügen, wenn wir auf den oberen Rand eines Rechtecks treffen, und aus R ein Intervall entfernen, wenn wir auf den unteren Rand eines Rechtecks treffen. Auf diese Weise reduzieren wir also das statische Schnittproblem für eine Menge von Rechtecken in der Ebene auf eine dynamische Folge von Überlappungsproblemen für horizontale Intervalle. Für ein Rechteck R bezeichnen wir die x-Koordinaten des linken und rechten Rands mit $x_l(R)$ und $x_r(R)$. $[x_l(R), x_r(R)]$ ist also ein R repräsentierendes Intervall. Wir nehmen stets an, daß $[x_l(R), x_r(R)]$ einen Verweis auf R enthält; mit anderen Worten: Man kann erkennen, welches Rechteck ein Intervall repräsentiert.

Jetzt formulieren wir das Scan-line-Verfahren zur Lösung des Rechteckschnittproblems:

Algorithmus *Rechteckschnitt*
{*liefert zu einer Menge von N iso-orientierten Rechtecken in der Ebene die Menge aller k Paare von sich schneidenden Rechtecken*}
$Q :=$ *Folge der* $2N$ *oberen und unteren Rechteckseiten in abnehmender y-Reihenfolge*;
$L :=$ $\emptyset$; {*Menge der Schnitte der gerade aktiven Rechtecke mit der Scan-line*}
while *Q ist nicht leer* **do**
  begin
    $q :=$ *nächster Haltepunkt von Q*;
    if q *ist oberer Rand eines Rechtecks R,* $q = [x_l(R), x_r(R)]$
      then
        begin
          bestimme alle Rechtecke R' *derart, daß das Intervall* $[x_l(R'), x_r(R')]$ *in L ist und* $[x_l(R), x_r(R)] \cap [x_l(R'), x_r(R')] \neq \emptyset$ *und gebe* (R, R') *aus*;
          füge $[x_l(R), x_r(R)]$ *in L ein*
        end
      else {q *ist unterer Rand eines Rechtecks R*}
        entferne $[x_l(R), x_r(R)]$ *aus L*
  end

Abbildung 7.12 zeigt ein Beispiel für die Anwendung des Verfahrens. An der in diesem Beispiel gezeigten vierten Haltestelle der Scan-line enthält L die drei Intervalle $[.B,B.] = [x_l(B), x_r(B)]$, $[.C,C.] = [x_l(C), x_r(C)]$ und $[.D,D.] = [x_l(D), x_r(D)]$. L trifft auf den oberen Rand von A. Also müssen alle Intervalle in L bestimmt werden, die sich mit dem Intervall $[.A,A.] = [x_l(A), x_r(A)]$ überlappen. Das ist nur das Intervall $[.B,B.]$. Also wird nur das Paar (A,B) ausgegeben und anschließend $[.A,A.]$ in L eingefügt.

Man beachte, daß alle Intervalle, die jemals in L eingefügt werden, aus L entfernt werden oder für die Überlappungen festgestellt werden müssen, Intervalle über einer diskreten Menge von höchstens $2N$ Endpunkten sind: Das ist die Menge der x-Koordinaten der linken und rechten Rechteckseiten. Wir können uns die Menge der möglichen Intervallgrenzen als mit der Menge der Rechtecke gegeben denken. Da es offenbar nur auf die relative Anordnung der Intervallgrenzen ankommt, können wir der Einfachheit halber sogar annehmen, daß die Intervallgrenzen ganzzahlig und äquidistant sind.

Damit haben wir die Implementierung des Verfahrens reduziert auf das Problem, eine Datenstruktur zur Speicherung einer Menge L von Intervallen $[a,b]$ mit $a,b \in \{1,\ldots,n\}$ zu finden, so daß folgende Operationen auf L ausführbar sind: Das *Einfügen* eines Intervalls in L, das *Entfernen* eines Intervalls aus L und das Ausführen von *Überlappungsfragen*, d.h. für ein gegebenes Intervall I: Bestimme alle Intervalle I' aus L, die sich mit I überlappen, d.h. für die $I \cap I' \neq \emptyset$ gilt.

Verschiedene Implementationen für L führen unmittelbar zu verschiedenen Lösungen des Rechteckschnittproblems. Wir besprechen zunächst zwei Möglichkeiten, die sich durch folgende weitere Reduktion der Überlappungsfrage ergeben.

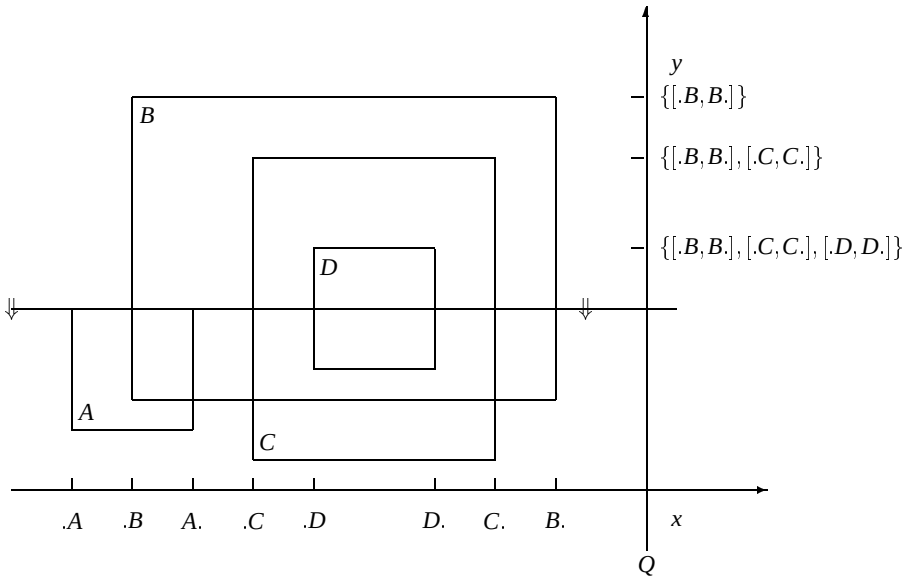
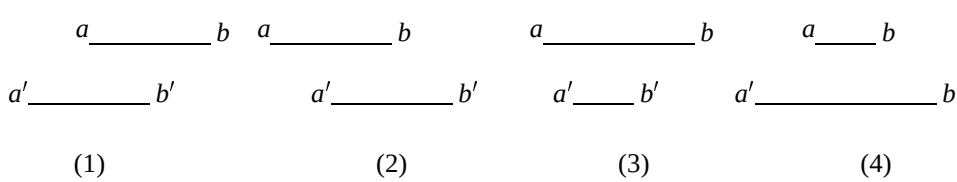


Abbildung 7.12

Nehmen wir an, es sollen alle Intervalle $[a', b']$ bestimmt werden, die sich mit einem gegebenen Intervall $[a, b]$ überlappen. Es gibt offenbar genau die folgenden vier Möglichkeiten für eine Überlappung:



D.h., es ist $a' \in [a, b]$, wie im Fall (2) und (3), oder es ist $a \in [a', b']$, wie im Fall (1) und (4). Die Überlappungsfrage kann damit reduziert werden auf eine Bereichsanfrage (range query) und eine sogenannte inverse Bereichsanfrage oder *Aufspießfrage* (*stabbing query*). Denn es gilt:

$$\begin{aligned} & \{[a', b'] \mid [a', b'] \cap [a, b] \neq \emptyset\} \\ = \; & \{[a', b'] \mid a \text{ spießt } [a', b'] \text{ auf } \} \cup \{[a', b'] \mid a' \text{ liegt im Bereich } [a, b]\} \end{aligned}$$

Dabei sagen wir: Ein Punkt spießt ein Intervall auf, wenn das Intervall den Punkt enthält.

Um also für ein gegebenes Intervall $[a,b]$ alle überlappenden Intervalle $[a',b']$ zu finden, genügt es offenbar:

1. alle Intervalle $[a',b']$ zu finden, die der linke Randpunkt a aufspießt, und
2. alle Intervalle $[a',b']$ zu finden, deren linker Randpunkt a' im Bereich $[a,b]$ liegt.

Die zweite Aufgabe ist mit bereits wohlbekannten Mitteln leicht lösbar: Man speichere alle linken Randpunkte in einem Bereichs-Suchbaum wie in Abschnitt 7.2.2 beschrieben.

Es genügt also, die erste Aufgabe zu lösen und eine Struktur zu entwerfen, die das Einfügen und Entfernen von Intervallen und das Beantworten von Aufspieß-Anfragen unterstützt. Wir bringen zwei Varianten einer derartigen Struktur, den Segment-Baum in Abschnitt 7.4.2 und den Intervall-Baum in Abschnitt 7.4.3.

7.4.2 Segment-Bäume

Segment-Bäume sind ein erstes Beispiel einer halb-dynamischen Skelettstruktur: Man baut zunächst ein leeres Skelett zur Aufnahme von Intervallen mit Endpunkten aus einer gegebenen Menge $\{1,\ldots,n\}$. Man kann in dieses Skelett Intervalle einfügen oder daraus entfernen. Ferner kann man für einen gegebenen Punkt feststellen, welche aktuell vorhandenen Intervalle er aufspießt.

Jedes Intervall $[a,b]$ mit $a,b \in \{1,\ldots,n\}$ kann man sich zusammengesetzt denken aus einer Folge von elementaren *Segmenten* $[i,i+1]$, $1 \leq i < n$. Ein Segment-Baum wird nun wie folgt konstruiert: Man baut einen vollständigen Binärbaum, also einen Binärbaum, der auf jedem Niveau die maximale Knotenzahl hat. Die Blätter repräsentieren die elementaren Segmente. Jeder innere Knoten repräsentiert die Vereinigung (der Folge) der elementaren Segmente an den Blättern im Teilbaum dieses Knotens. Die Wurzel repräsentiert also das Intervall $[1,n]$. Das ist das leere Skelett eines Segment-Baumes. Das Skelett kann nun dynamisch mit Intervallen gefüllt werden, indem man den Namen eines einzufügenden Intervalls an genau diejenigen Knoten schreibt, die am nächsten bei der Wurzel liegen und ein Intervall repräsentieren, das vollständig in dem einzufügenden Intervall enthalten ist. Abbildung 7.13 zeigt das Beispiel eines Segment-Baumes, der die Intervalle $\{A,\ldots,F\}$ mit Endpunkten in $\{1,\ldots,9\}$ enthält. An jedem Knoten sind das von ihm repräsentierte Intervall als durchgezogene Linie und die Liste der Namen von Intervallen angegeben, die diesem Knoten zugeordnet wurden (aus Gründen der Darstellung liegen Lücken zwischen Intervallen).

Bezeichnen wir mit $I(p)$ das durch den Knoten p des Segment-Baumes repräsentierte Intervall, so gilt: Der Name eines Intervalls I tritt in der Intervall-Liste des Knotens p auf genau dann, wenn $I(p) \subseteq I$ gilt und für keinen Knoten p' auf dem Pfad von der Wurzel zu p $I(p') \subseteq I$ gilt.

Daraus ergibt sich sofort folgendes Verfahren zum Einfügen eines Intervalls I:

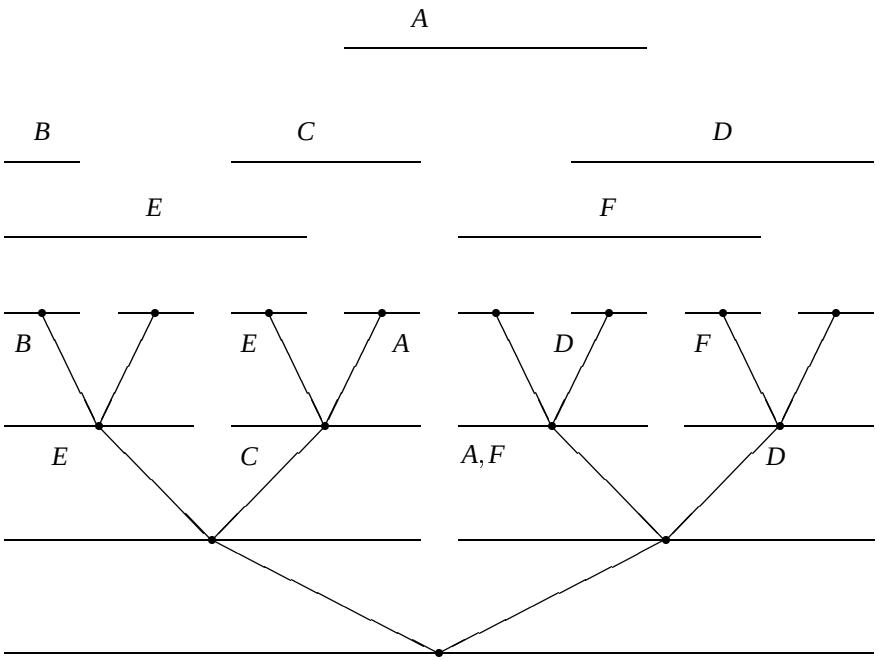


Abbildung 7.13

```
procedure Einfügen (I : Intervall; p : Knoten);
{anfangs ist p die Wurzel des Segment-Baumes}
  if I(p) ⊆ I
    then
      füge I in die Intervall-Liste von p ein und fertig
    else
      begin
        if (p hat linken Sohn p_λ) and (I(p_λ) ∩ I ≠ ∅)
          then Einfügen(I, p_λ);
        if (p hat rechten Sohn p_ρ) and (I(p_ρ) ∩ I ≠ ∅)
          then Einfügen(I, p_ρ)
      end
```

Auf den ersten Blick könnte man den Verdacht haben, daß diese rekursiv formulierte Einfügeprozedur im schlimmsten Fall für sämtliche Knoten eines Segment-Baumes aufgerufen wird. Das ist jedoch keineswegs der Fall, wie folgende Überlegung zeigt: Wird die Einfügeprozedur nach einem Aufruf von *Einfügen*(I, p) für beide Söhne p_λ und p_ρ eines Knotens p aufgerufen und bricht die Prozedur nicht bereits für einen dieser beiden Söhne ab, so kann die Einfügeprozedur für höchstens zwei der Enkel von p erneut aufgerufen werden. Das zeigt Abbildung 7.14. In dieser Abbildung ist durch "$*$"

ein Aufruf der Einfügeprozedur und durch "†" angedeutet, daß das Einfüge-Verfahren hier abbricht, da diese Knoten ein ganz in I enthaltenes Intervall repräsentieren.

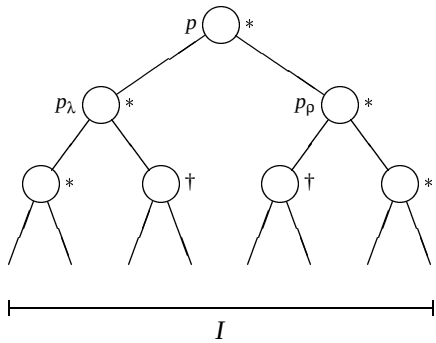


Abbildung 7.14

Die Folge der rekursiven Aufrufe der Einfügeprozedur kann man daher stets als einen sich höchstens einmal gabelnden Pfad darstellen, wie ihn Abbildung 7.15 zeigt.

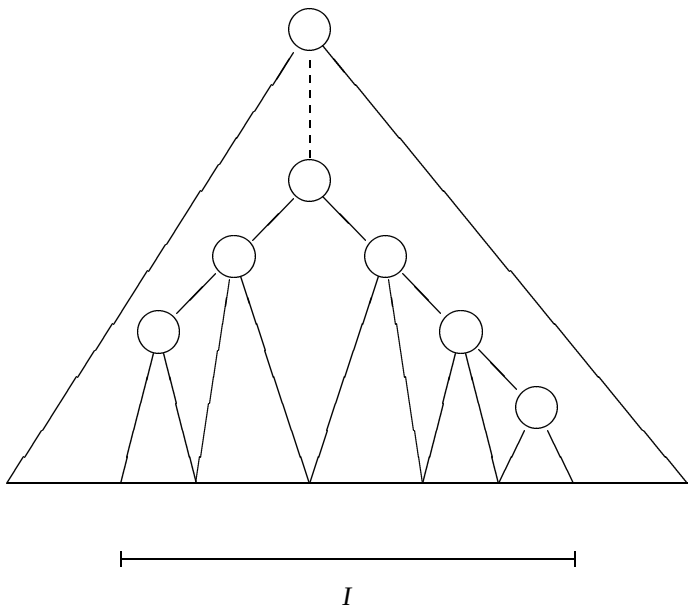


Abbildung 7.15

Aus dieser Überlegung kann man schließen:

1. Das Einfügen eines Intervalls ist in $O(\log N)$ Schritten ausführbar.
2. Jedes Intervall I kommt in höchstens $O(\log N)$ Intervall-Listen vor.

Denn der Segment-Baum mit $(N-1)$ Segmenten hat die Höhe $\log N$.

Wir haben allerdings stillschweigend vorausgesetzt, daß das Einfügen eines Intervalls (bzw. eines Intervall-Namens) in die zu einem Knoten des Segment-Baumes gehörende Intervall-Liste in konstanter Schrittzahl möglich ist. Das ist leicht erreichbar, wenn wir die Intervall-Listen als verkettete Listen implementieren und neue Intervalle stets am Anfang oder Ende einfügen. Man beachte aber, daß wir dann unter Umständen Schwierigkeiten haben, ein Intervall in einer zu einem Knoten gehörenden Intervall-Liste zu finden und daraus gegebenenfalls zu entfernen. Man beachte schließlich noch, daß die Intervall-Listen auf einem beliebigen Pfad im Segment-Baum von der Wurzel zu einem Blatt paarweise disjunkt sein müssen. Denn sobald ein Intervall in die Liste eines Knotens p aufgenommen wurde, wird es in keine Liste eines Nachfolgers von p eingefügt.

Wie können Aufspieß-Fragen beantwortet werden? Um für einen gegebenen Punkt x alle im Segment-Baum gespeicherten Intervalle zu finden, die x aufspießt, benutzen wir den Segment-Baum als Suchbaum für x. D.h. wir suchen nach dem Elementarsegment, das x enthält. Wir geben dann alle Intervalle in allen Listen aus, die zu Knoten auf dem Suchpfad gehören. Denn das sind genau sämtliche Intervalle, die x aufspießt. Genauer: Wir rufen die folgende Prozedur *report* für die Wurzel des Segment-Baumes und den Punkt x auf.

```
procedure report (p : Knoten; x : Punkt);
{ohne Einschränkung ist x ∈ I(p)}
gebe alle Intervalle der Liste von p aus;
if p ist Blatt
   then fertig
   else
      begin
         if (p hat einen linken Sohn p_λ) and (x ∈ I(p_λ))
            then report(p_λ, x);
         if (p hat einen rechten Sohn p_ρ) and (x ∈ I(p_ρ))
            then report(p_ρ, x)
      end
```

Natürlich kann niemals zugleich $x \in I(p_\lambda)$ und $x \in I(p_\rho)$ sein. Daher werden in der Tat genau $\lceil \log_2 N \rceil$ Intervall-Listen betrachtet. Der Aufwand, die Intervalle auszugeben, ist damit proportional zu $\log N$ und zur Anzahl der Intervalle, die x enthalten. Insgesamt haben wir damit eine Struktur mit folgenden Charakteristika: Das Einfügen eines Intervalls ist in Zeit $O(\log N)$ möglich; die zum Beantworten einer Aufspieß-Frage erforderliche Zeit ist $O(\log N + k)$, wobei k die Größe der Antwort ist. Die Struktur hat den Speicherbedarf $O(N \log N)$.

Um ein Intervall aus dem Segment-Baum zu entfernen, können wir im Prinzip genauso vorgehen wie beim Einfügen: Wir bestimmen zunächst die $O(\log N)$ Knoten, in deren Intervall-Listen das zu entfernende Intervall vorkommt und entfernen es dann aus jeder dieser Listen. Da wir jedoch nicht wissen, wo das Intervall in diesen Listen vorkommt, bleibt uns nichts anderes übrig, als jede dieser Listen von vorn nach hinten zu durchsuchen. Das kann im schlimmsten Fall $O(N)$ Schritte für jede Liste kosten — ein nicht akzeptabler Aufwand. Wir wollen vielmehr erreichen, daß wir für jedes Intervall I alle Vorkommen von I in Intervall-Listen von Knoten des Segment-Baumes in einer Anzahl von Schritten bestimmen können, die proportional zu $\log N$ und zur Anzahl dieser Vorkommen ist.

Wir lösen das Problem folgendermaßen: Als Grundstruktur nehmen wir einen Segment-Baum, wie wir ihn bisher beschrieben haben. Darüberhinaus speichern wir alle im Segment-Baum vorkommenden Intervallnamen in einem alphabetisch sortierten Wörterbuch ab. D.h., in einer Struktur, die das Suchen, Einfügen und Entfernen eines Intervallnamens in $O(\log N)$ Schritten erlaubt, wenn wir eine Implementation durch balancierte Bäume verwenden und N die insgesamt vorhandene Zahl von Intervallnamen ist. Jeder Intervallname I dieses Wörterbuches zeigt auf den Anfang einer verketteten Liste von Zeigern, die auf alle Vorkommen von I in der Grundstruktur weisen. Insgesamt erhalten wir damit eine Struktur, die grob wie in Abbildung 7.16 dargestellt werden kann.

Da wir den Segment-Baum im wesentlichen unverändert gelassen haben, können wir Aufspieß-Fragen wie bisher beantworten. Beim Einfügen eines neuen Intervalls müssen wir natürlich den Namen dieses Intervalls zusätzlich in das Wörterbuch einfügen und auch die verkettete Liste von Zeigern auf sämtliche Vorkommen des Intervalls im Segment-Baum aufbauen. Da jedes Intervall an höchstens $\log N$ Stellen im Segment-Baum vorkommen kann, ist der Gesamtaufwand für das auf diese Weise veränderte Einfügen eines Intervalls immer noch von der Größenordnung $O(\log N)$. Das Entfernen eines Intervalls kann jetzt genau umgekehrt zum Einfügen ebenfalls in $O(\log N)$ Schritten ausgeführt werden: Man sucht den Namen I des zu entfernenden Intervalls im Wörterbuch, findet dort die Verweise auf alle Vorkommen von I im Segment-Baum und kann I zunächst dort und anschließend auch im Wörterbuch löschen. Der gesamte Speicherbedarf dieser Struktur ist offenbar $O(N \log N)$.

Wir haben jetzt alles beisammen zur Lösung des eingangs gestellten Problems, alle k Paare von sich schneidenden Rechtecken in einer gegebenen Menge von N isoorientierten Rechtecken mit Hilfe des Scan-line-Verfahrens zu bestimmen. Man verwendet als Horizontalstruktur ein Paar von zwei dynamischen, also Einfügungen und Streichungen erlaubenden Strukturen, einen Bereichs-Suchbaum zur Speicherung der linken Endpunkte der jeweils gerade aktiven Intervalle (= Schnitte der jeweils aktiven Rechtecke mit der Scan-line), und einen Segment-Baum für die jeweils aktiven Intervalle, um ein Wörterbuch für die Intervallnamen erweitert, wie eben beschrieben.

Die Strukturen liefern insgesamt eine Möglichkeit zur Implementation einer Menge L von N Intervallen derart, daß das Einfügen und Entfernen eines Intervalls stets in $O(\log N)$ Schritten möglich ist und alle r Intervalle aus L, die sich mit einem gegebenen Intervall überlappen, in Zeit $O(\log N + r)$ bestimmt werden können. Daher gilt:

Das Rechteckschnittproblem kann nach dem Scan-line-Verfahren mit Hilfe von Segment-Bäumen in Zeit $O(N \log N + k)$ und Platz $O(N \log N)$ gelöst werden. Dabei

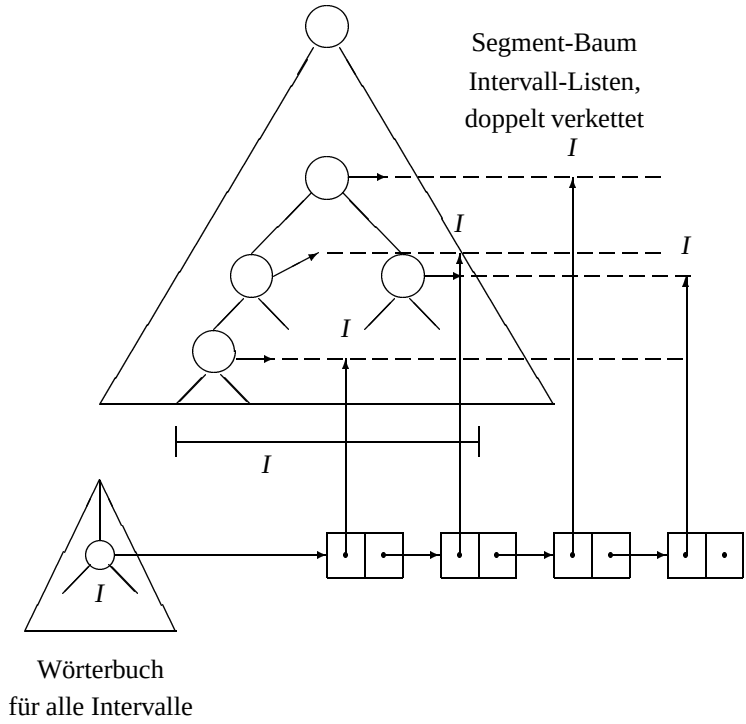


Abbildung 7.16

ist N die Anzahl der gegebenen Rechtecke und k die Anzahl der Paare sich schneidender Rechtecke.

Wir vergleichen dieses Ergebnis mit der in Abschnitt 7.3.2 erhaltenen Divide-and-conquer-Lösung desselben Problems: Die Laufzeit beider Verfahren ist dieselbe, aber der Speicherbedarf der Scan-line-Lösung ist nicht linear beschränkt. Wir werden im nächsten Abschnitt Intervall-Bäume als Alternative zu Segment-Bäumen vorstellen, die ebenfalls in einer dem Scan-line-Prinzip folgenden Lösung des Rechteckschnittproblems verwendet werden können, die aber nur linearen Speicherbedarf haben. Darauf kann man eine Scan-line-Lösung des Rechteckschnittproblems gründen, die Zeitbedarf $O(N \log N + k)$ und Platzbedarf $O(N)$ hat.

Segment-Bäume sind jedoch unabhängig von ihrer Verwendung in diesem Abschnitt von eigenem Interesse. Gerade die redundante Abspeicherung von Intervallen an vielen Knoten und die Freiheit, die Intervallnamen in den Knotenlisten beliebig, und damit auch nach neuen Kriterien anzuordnen, sind der Schlüssel zu weiteren Anwendungen (vgl. hierzu die Abschnitte 7.5 und 7.6).

Schließlich bemerken wir noch, daß man Segment-Bäume auch voll dynamisch machen kann in dem Sinne, daß ihre Größe nicht von der Größe des Skeletts, sondern nur von der Anzahl der jeweils gerade vorhandenen Intervalle abhängt. Einfüge- und

Entferne-Operationen sind aber noch komplizierter und damit einer Implementierung für die Praxis noch weniger zugänglich.

7.4.3 Intervall-Bäume

Wir wollen jetzt eine Datenstruktur zur Speicherung einer Menge von $O(N)$ Intervallen mit Endpunkten in einer diskreten Menge von $O(N)$ Endpunkten vorstellen, die nur linearen Speicherbedarf hat und die Operationen Einfügen eines Intervalls, Entfernen eines Intervalls und Aufspieß-Fragen in Zeit $O(\log N)$ bzw. $O(\log N + k)$ auszuführen erlaubt. Es dürfte unmittelbar klar sein, daß wir damit auch eine Verbesserung des Scanline-Verfahrens zur Lösung des Rechteckschnittproblems erhalten.

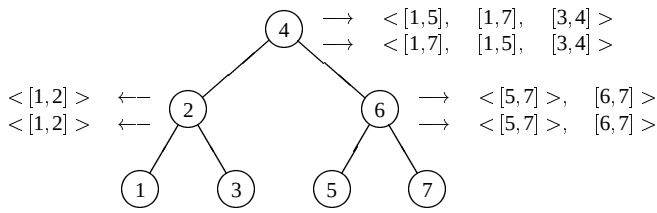
Da wir es stets nur mit einer endlichen Menge von Intervallen zu tun haben, können wir ohne Einschränkung annehmen, daß die Intervallgrenzen einer gegebenen Menge von höchstens N Intervallen auf die ganzen Zahlen $1,\ldots,s$ fallen, wobei $s \leq 2N$ ist.

Ein Intervall-Baum zur Speicherung einer Menge von Intervallen mit Endpunkten in $\{1,\ldots,s\}$ besteht aus einem Skelett und sortierten Intervallisten, die mit den Knoten des Skeletts des Intervall-Baumes verbunden sind. Das Skelett des Intervall-Baumes ist ein vollständiger Suchbaum für die Schlüsselmenge $\{1,\ldots,s\}$. Jeder innere Knoten dieses Suchbaumes ist mit zwei sortierten Intervall-Listen verbunden, einer *u*-Liste und einer *o*-Liste. Die *u*-Liste ist eine nach aufsteigenden unteren Endpunkten sortierte Liste von Intervallen und die *o*-Liste eine nach absteigenden oberen Endpunkten sortierte Liste von Intervallen. Ein Intervall $[l,r]$ mit $l,r \in \{1,\ldots,s\}, l \leq r$, kommt in der *u*-Liste und *o*-Liste desjenigen Knotens im Skelett des Intervall-Baumes mit minimaler Tiefe vor, dessen Schlüssel im Intervall $[l,r]$ liegt.

Folgendes Beispiel zeigt einen Intervall-Baum für die Menge

$$\{[1,2],[1,5],[3,4],[5,7],[6,7],[1,7]\}$$

von Intervallen mit Endpunkten in $\{1,\ldots,7\}$.



In diesem Beispiel sind die *u*-Listen stets oben und die *o*-Listen unten an die jeweils zugehörigen Knoten geschrieben. Alle nicht explizit dargestellten *u*- und *o*-Listen sind leer. (Offenbar müssen die den Blättern zugeordneten Listen immer leer sein, wenn man nicht Intervalle $[i,i]$ mit $1 \leq i \leq s$ zuläßt!)

Bezeichnen wir für einen Knoten p eines Intervall-Baumes den Schlüssel von p mit $p.key$, den linken Sohn von p mit p_λ und den rechten mit p_ρ, so kann das Verfahren zum Einfügen eines Intervalls $I = [.I,I.]$ in einen Intervall-Baum wie folgt beschrieben werden:

```
procedure Einfügen (I : Intervall; p : Knoten);
{anfangs ist p die Wurzel des Intervall-Baumes; I ist ein Intervall
 mit linkem Endpunkt .I und rechtem Endpunkt I.}
if p.key ∈ I
   then
      füge I entsprechend seinem unteren Endpunkt in die u-Liste
         von p und entsprechend seinem oberen Endpunkt in die
         o-Liste von p ein und fertig!
   else
      if p.key < .I
         then Einfügen(I, p_ρ)
            else {p.key > I.}
               Einfügen(I, p_λ)
```

Für jedes Intervall I und jeden Knoten p gilt, daß I entweder $p.key$ enthalten muß oder aber I liegt ganz rechts von $p.key$ (dann ist $p.key < .I$) oder I liegt ganz links von $p.key$ (dann ist $p.key > I.$). Da wir angenommen hatten, daß alle möglichen Intervallgrenzen als Schlüssel von Knoten im Skelett des Segment-Baumes vorkommen, ist klar, daß das rekursiv formulierte Einfüge-Verfahren hält. Implementiert man die u-Liste und o-Liste eines jeden Knotens als balancierten Suchbaum, folgt, daß das Einfügen eines Intervalls in einer Anzahl von Schritten ausgeführt werden kann, die höchstens linear von der Höhe des Intervallbaum-Skeletts und logarithmisch von der Länge der einem Knoten zugeordneten u- und o-Listen abhängt. Mit der zu Beginn dieses Abschnitts gemachten Annahme sind das $O(\log N)$ Schritte.

Das Entfernen eines Intervalls I erfolgt natürlich genau umgekehrt zum Einfügen: Man bestimmt ausgehend von der Wurzel des Intervall-Baumes den Knoten p mit geringster Tiefe, für den $p.key \in I$ gilt. (Einen derartigen Knoten muß es stets geben!) Dann entfernt man I aus den sortierten u- und o-Listen von p. Offenbar kann man das ebenfalls in $O(\log N)$ Schritten ausführen.

Nun überlegen wir uns noch, wie Aufspieß-Fragen beantwortet werden können. Dabei nehmen wir an, daß der Punkt x, für den wir alle im Intervall-Baum gespeicherten Intervalle finden wollen, die x aufspießt, einer der Schlüssel des Skeletts ist. Das ist keine wesentliche Annahme, sondern soll lediglich sichern, daß eine Suche nach x im Skelett des Intervall-Baumes stets erfolgreich endet, und die Präsentation des Verfahrens vereinfacht wird.

Zur Bestimmung der Intervalle, die ein gegebener Punkt aufspießt, suchen wir im Skelettbaum nach x. Die Suche beginnt bei der Wurzel und endet beim Knoten mit Schlüssel x. Ist p ein beliebiger Knoten auf diesem Pfad und ist $p.key \neq x$, dann kann man nicht sämtliche Intervalle ausgeben, die in der u- bzw. o-Liste von p vorkommen, denn diese Listen enthalten Intervalle, die zwar $p.key$, aber möglicherweise x nicht aufspießt. Ist jedoch $p.key > x$, so könnte x die Intervalle eines Anfangsstücks der u-Liste von p durchaus ebenfalls aufspießen. Entsprechend kann x durchaus einige Intervalle eines Anfangsstücks der o-Liste aufspießen, wenn $p.key < x$ ist. Diese Intervalle müssen natürlich sämtlich ausgegeben werden. Abbildung 7.17 illustriert die beiden Fälle.

Wir haben angenommen, daß genau einer der drei Fälle $x = p.key$ oder $x < p.key$ oder $x > p.key$ möglich ist. Daher können wir das Verfahren zum Berichten aller Intervalle eines Intervall-Baumes, die x aufspießt, wie folgt formulieren:

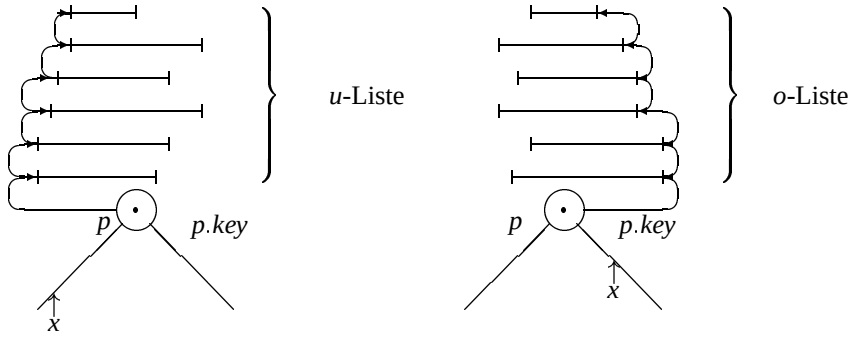


Abbildung 7.17

```
procedure report(p : Knoten; x : Punkt);
if x = p.key
  then
    gebe alle Intervalle der u-Liste (oder
      alle Intervalle der o-Liste) von p aus und fertig!
  else
    if x < p.key
      then
        gebe alle Intervalle I der u-Liste von p mit
          .I ≤ x aus  {das ist ein Anfangsstück dieser Liste!}
          report (p_λ, x)
      else {x > p.key}
        gebe alle Intervalle I der o-Liste von p mit
          I. ≥ x aus  {das ist ein Anfangsstück dieser Liste!}
          report (p_ρ, x)
```

Die Ausgabe eines Anfangsstücks einer sortierten Liste, die als balancierter Suchbaum implementiert ist, kann offensichtlich in einer Anzahl von Schritten erfolgen, die linear mit der Anzahl der ausgegebenen Elemente wächst.

Die *u*- und *o*-Listen eines Knotens des Skeletts eines Intervall-Baumes können maximal alle N Intervalle enthalten. Da jedoch jedes Intervall in der *u*- und *o*-Liste höchstens eines Knotens vorkommen kann, benötigt die Struktur insgesamt nur $O(N)$ Speicherplatz. Wir fassen unsere Überlegungen wie folgt zusammen:

Intervall-Bäume eignen sich zur Speicherung einer dynamisch veränderlichen Menge von höchstens N Intervallen mit Endpunkten im Bereich $\{1,\ldots,s\}$, $s \leq 2N$. Sie haben Speicherbedarf $O(N)$ und erlauben das Einfügen eines Intervalls in Zeit $O(\log N)$, das Entfernen eines Intervalls in Zeit $O(\log N)$, und das Beantworten von Aufspieß-Fragen in Zeit $O(\log N + k)$, wobei k die Größe der Antwort ist.

Der Aufbau eines Intervall-Baumes kann durch Bildung des zunächst leeren Skeletts in Zeit $O(N)$ geschehen, das dann durch iteriertes Einfügen gefüllt wird.

Auf Grund der bereits zum Ende des vorigen Abschnitts 7.4.2 angestellten Überlegungen erhält man ferner:

Das Rechteckschnittproblem kann nach dem Scan-line-Verfahren mit Hilfe von Intervall-Bäumen in Zeit $O(N \log N + k)$ und Platz $O(N)$ gelöst werden. Dabei ist N die Zahl der gegebenen Rechtecke und k die Anzahl sich schneidender Paare von Rechtecken.

Intervall-Bäume haben gegenüber Segment-Bäumen den Vorzug, weniger Speicherplatz zu beanspruchen. Ihr Nachteil ist, daß sie weniger flexibel sind. Denn im Unterschied zu Segment-Bäumen kann man die Knotenlisten in Intervall-Bäumen nicht beliebig anordnen.

Intervall-Bäume wurden unabhängig voneinander von Edelsbrunner [41] und McCreight [119] erfunden. McCreight kommt jedoch auf ganz anderem Wege zu dieser Struktur und nennt sie Kachelbaum-Struktur (tile tree): Er benutzt die Darstellung von Intervallen durch Punkte in der Ebene, wie wir sie im nächsten Abschnitt kennenlernen werden. Für Intervall-Bäume gilt übrigens wie für Segment-Bäume, daß sie vollkommen dynamisch gemacht werden können; d.h. ihre Größe paßt sich der Anzahl der jeweils vorhandenen Intervalle dynamisch an. Wir haben dagegen eine halbdynamische Struktur: Ein anfangs leeres Skelett kann dynamisch gefüllt werden.

7.4.4 Prioritäts-Suchbäume

Wir haben bereits in Abschnitt 7.4.1 gezeigt, daß es zur Implementation des Scan-line-Verfahrens zur Lösung des Rechteckschnittproblems genügt, eine Implementation für eine Menge L von Intervallen zu finden, auf der folgende Operationen ausgeführt werden: Einfügen eines Intervalls, Entfernen eines Intervalls und für ein gegebenes Intervall I alle Intervalle I' aus L finden, die sich mit I überlappen, für die also $I \cap I' \neq \emptyset$ ist. Nachdem wir in den Abschnitten 7.4.2 und 7.4.3 zwei Möglichkeiten angegeben haben, die sich durch eine weitere Reduktion des Überlappungsproblems für Intervalle auf das Beantworten von Bereichs- und Aufspieß-Fragen ergaben, wollen wir jetzt das Überlappungsproblem direkt betrachten.

Jedes Intervall (mit Endpunkten aus einer festen, beschränkten Menge möglicher Endpunkte) kann man repräsentieren durch einen Punkt im zweidimensionalen Raum: Repräsentiere das Intervall $[l,r]$ mit $l \leq r$ durch den Punkt (r,l). Dann bedeutet die Aufgabe, alle Intervalle $[x',y']$ zu bestimmen, die sich mit einem gegebenen Intervall $I = [x,y]$ überlappen, genau dasselbe wie die Aufgabe, alle Punkte (y',x') zu berichten, mit $x \leq y'$ und $x' \leq y$, d.h. alle Punkte, die rechts unterhalb des Frage-Punkts (x,y) liegen. Abbildung 7.18 erläutert dies genauer an einem Beispiel.

Es genügt also, eine Struktur zur Speicherung einer Menge von Punkten im zweidimensionalen Raum zu finden, derart, daß das Einfügen und Entfernen von Punkten möglichst effizient ausführbar ist und außerdem alle Punkte eines bestimmten Bereichs möglichst schnell berichtet werden können. Glücklicherweise sind die Bereiche, die wir zulassen müssen, von sehr spezieller Form. Ihre Grenzen sind parallel zu den gegebenen Koordinatenachsen, also iso-orientiert; mehr noch, sie sind stets *S-gegründet*

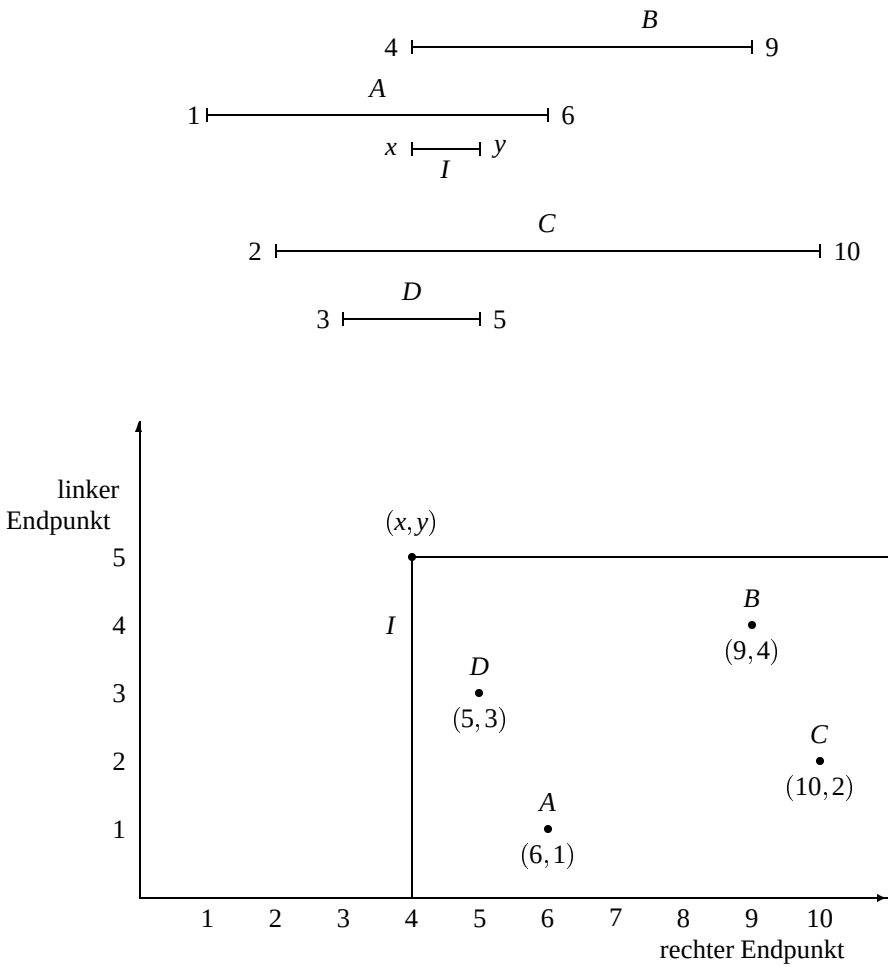


Abbildung 7.18

(*south-grounded*), d.h. die untere Bereichsgrenze fällt mit der x-Achse zusammen. Man kann einen solchen Bereich als *1.5-dimensional* ansehen. Denn er ist festgelegt durch einen eindimensionalen Bereich in x-Richtung (den linken und rechten Randwert) und durch eine Obergrenze in y-Richtung, vgl. Abbildung 7.19.

(Die zur Lösung des Überlappungsproblems für Intervalle benötigten Bereiche sind rechts offen, d.h. sie haben den maximal möglichen x-Wert als rechten Randwert.) Prioritäts-Suchbäume sind genau auf diese Situation zugeschnitten. Sie sind eine 1.5-dimensionale Struktur zur Speicherung von Punkten im zweidimensionalen Raum. Ein *Prioritäts-Suchbaum* ist ein Blattsuchbaum für die x-Werte und zugleich ein Heap für die y-Werte der Punkte. Genauer: Jeder Punkt (x,y) wird auf einem Suchpfad von der

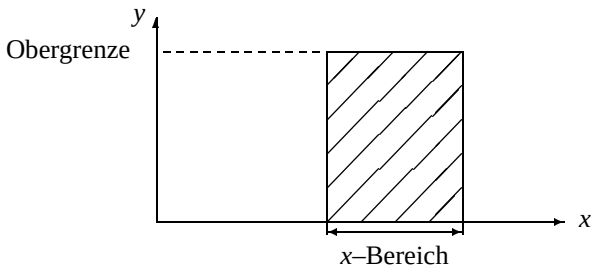


Abbildung 7.19

Wurzel zum Blatt x an einem inneren Knoten entsprechend seinem y-Wert abgelegt. D.h. die y-Werte nehmen auf jedem Suchpfad höchstens zu. Auch Prioritäts-Suchbäume kann man als volldynamische oder halbdynamische Skelettstrukturen über einem festen, beschränkten Universum entwickeln. Abbildung 7.20 zeigt einen Prioritätssuchbaum, der die Punkte *A, B, C, D* des ersten Beispiels über dem Universum $\{1,\ldots,10\}$ möglicher x-Koordinaten speichert.

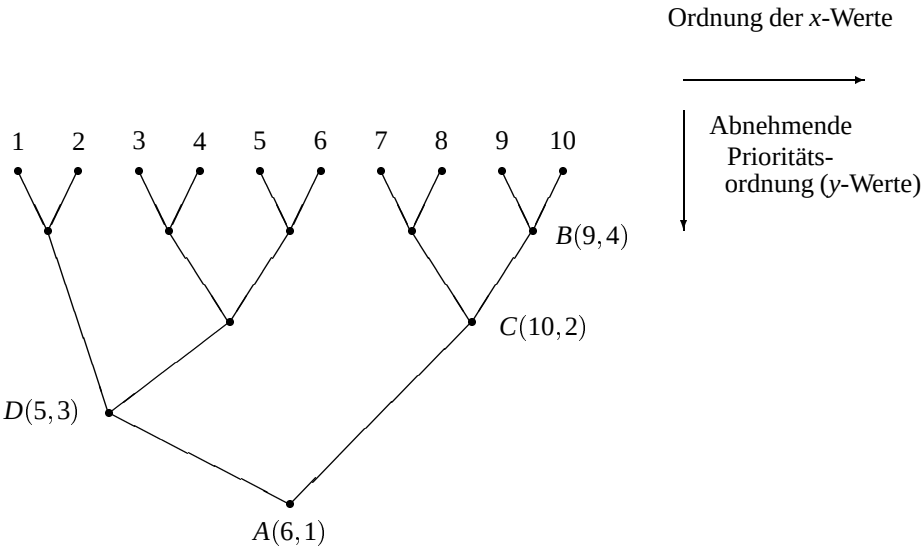


Abbildung 7.20

Wir haben die Punkte natürlich stets so nah wie möglich bei der Wurzel gespeichert. Wollen wir in diesen Prioritäts-Suchbaum als weiteren Punkt etwa den Punkt $E = (8,1)$ einfügen, können wir so vorgehen: Wir folgen dem Suchpfad von der Wurzel zum Blatt mit Wert 8. Auf diesem Pfad muß der Punkt E abgelegt werden und zwar so, daß die y-Koordinaten aller unterwegs angetroffenen Punkte höchstens zunehmen. Daher legen wir den Punkt E an der Stelle ab, an der zuvor der Knoten C stand. Nun fahren wir mit $C = (10,2)$ statt E fort und folgen dem Suchpfad zum Blatt 10. Dabei treffen wir auf den Knoten $B = (9,4)$ und sehen, daß wir C dort ablegen müssen. Schließlich wird B beim Blatt 9 abgelegt.

Wir haben in den zur Veranschaulichung benutzten Figuren die Suchstruktur von Prioritätsbäumen nicht explizit deutlich gemacht, sondern vielmehr stillschweigend angenommen, daß an den inneren Knoten eines Prioritäts-Suchbaumes stets geeignete Wegweiser stehen, die eine Suche nach einem mit einem bestimmten x-Wert bezeichneten Blatt dirigieren. Eine Möglichkeit ist, das Maximum der Werte im linken Teilbaum zu nehmen. Wir wollen diesen Wert den die Suche dirigierenden *Splitwert* eines Knotens p nennen und mit $p.sv$ bezeichnen. Zur Vereinfachung nehmen wir ferner an, daß kein x-Wert eines Punktes doppelt auftritt. (Ist diese Voraussetzung für eine gegebene Menge von Punkten nicht erfüllt, so betrachte man statt einer Menge von Punkten (x,y) die Menge der Punkte $((x,y),y)$, wobei die erste Koordinate lexikographisch, also zuerst nach x, dann nach y geordnet ist.)

Jeder Knoten p eines Prioritäts-Suchbaumes kann höchstens einen Punkt speichern, den wir mit $p.Punkt$ bezeichnen. $p.Punkt$ kann undefiniert sein. Ist $p.Punkt$ definiert, sind $p.Punkt.x$ und $p.Punkt.y$ die Koordinaten des am Knoten p gespeicherten Punktes $p.Punkt$.

Wir beschreiben jetzt zunächst das leere Skelett eines Prioritäts-Suchbaumes zur Speicherung einer Menge von N Punkten $\{(x_1,y_1),\ldots,(x_N,y_N)\}$: Es besteht aus einem vollständigen, binären Blattsuchbaum für die (nach Annahme paarweise verschiedenen) x-Werte $\{x_1,\ldots,x_N\}$ der Punkte. Diese x-Werte sind die Splitwerte der Blätter in aufsteigender Reihenfolge von links nach rechts; die Punkt-Komponenten der Blätter sind undefiniert. Jeder innere Knoten des leeren Skeletts hat als Splitwert das Maximum der Splitwerte im linken Teilbaum; die Punktkomponenten sind ebenfalls undefiniert.

Das Verfahren zum (iterierten) Einfügen eines Punktes A aus der gegebenen Menge mit den Koordinaten $A.x$ und $A.y$ kann nun wie folgt formuliert werden:

```
procedure Einfügen (p : Knoten; A : Punkt);
{anfangs ist p die Wurzel des Skeletts}
if p.Punkt ist undefiniert
   then {A ablegen} p.Punkt := A
   else
      if p.Punkt.y ≤ A.y
         then {Suchpfad nach A.x folgen}
            begin
               if p.sv ≥ A.x
                  then Einfügen(p_λ, A)
                  else Einfügen(p_ρ, A)
            end
         else {p.Punkt.y > A.y}
```

```
begin
   {A ablegen und mit p.Punkt weitermachen}
   hilf := p.Punkt;
   p.Punkt := A;
   Einfügen(p, hilf)
end
```

Betrachten wir als Beispiel eine Menge M von acht Punkten:

$$M = \{(1,3),(2,4),(3,7),(4,2),(5,1),(6,6),(7,5),(8,4)\}$$

Nach Einfügen der ersten drei Punkte $(1,3)$, $(2,4)$, $(3,7)$ in das anfänglich leere Skelett erhält man den Prioritäts-Suchbaum von Abbildung 7.21. Dabei sind die Splitwerte jeweils in der oberen und die Punkte in der unteren Hälfte der Knoten dargestellt.

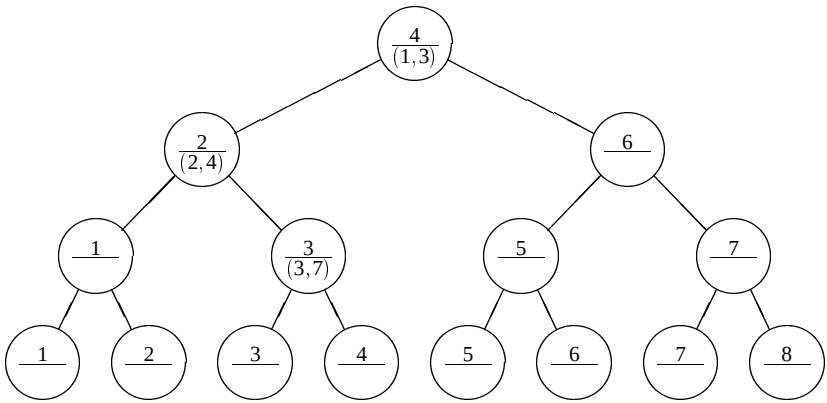


Abbildung 7.21

Einfügen des Punktes $(4,2)$ liefert den Baum von Abbildung 7.22.

Einfügen des Punktes $(5,1)$ liefert den Baum von Abbildung 7.23.

Einfügen der restlichen Punkte $(6,6)$, $(7,5)$, $(8,4)$ ergibt schließlich den Prioritäts-Suchbaum von Abbildung 7.24.

Wir hatten angenommen, daß nur Punkte aus der vorher bekannten Menge M mit paarweise verschiedenen x-Werten in das anfänglich leere Skelett eingefügt werden. Daher kann niemals der Fall eintreten, daß für einen dieser Punkte kein Platz auf dem Suchpfad von der Wurzel zu dem mit dem x-Wert des Punktes markierten Blatt ist: Spätestens in diesem Blatt findet der Punkt Platz.

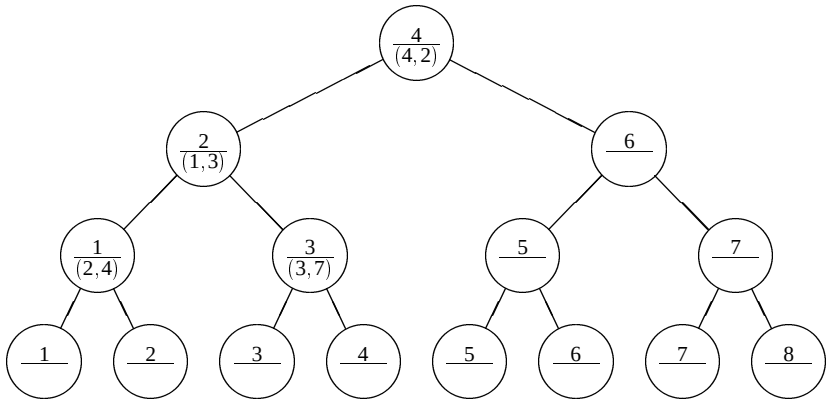


Abbildung 7.22

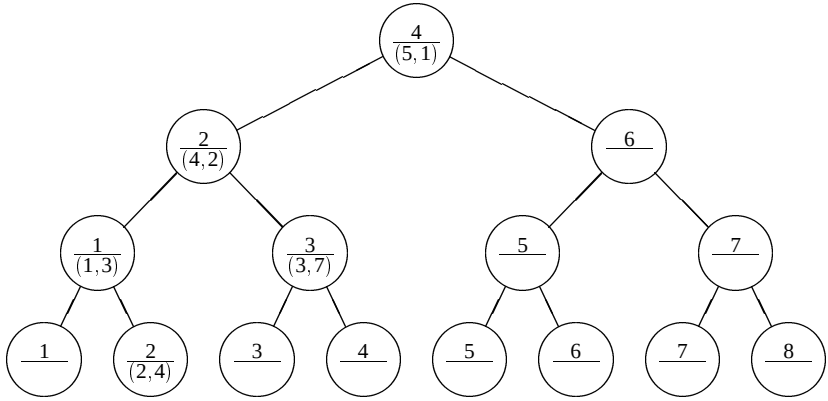


Abbildung 7.23

Um einen Punkt (x_0, y_0) aus dem Prioritäts-Suchbaum zu *entfernen*, sucht man zunächst den Knoten p mit $p.Punkt = (x_0, y_0)$. Die Suche wird allein durch den x-Wert des zu entfernenden Punktes und die Splitwerte der Knoten dirigiert. Hat höchstens einer der Söhne von p einen Punkt gespeichert, kann man diesen Punkt "hochziehen", d.h. zur Punktkomponente von p machen und mit dem Sohn von p ebenso fortfahren, bis man bei den Blättern oder bei einem Knoten angelangt ist, der nur Söhne ohne Punktkomponenten hat. Haben beide Söhne von p einen Punkt gespeichert, ersetzt man $p.Punkt$ durch den Punkt mit dem kleineren y-Wert. Durch dieses Hochziehen entsteht dort eine Lücke, die auf dieselbe Weise geschlossen wird. Mit anderen Worten: Die durch das Entfernen eines Punktes im Innern des Prioritäts-Suchbaumes entstehende Lücke wird

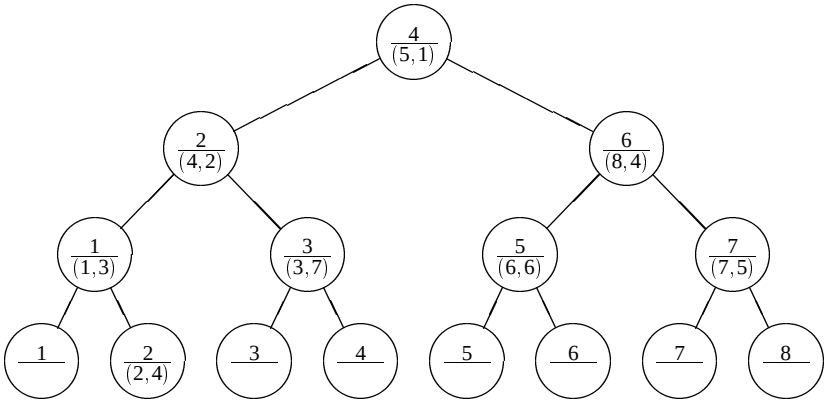


Abbildung 7.24

nach Art eines Ausscheidungskampfes unter den Punkten der Söhne geschlossen: Der Punkt mit dem jeweils kleineren y-Wert gewinnt und wird hochgezogen.

Das Verfahren zum Entfernen eines Punktes A kann damit wie folgt formuliert werden:

```
1. Schritt: {Suche nach einem Knoten p mit p.Punkt = A}
{anfangs ist p die Wurzel}
while (p.Punkt ist definiert) and (p.Punkt ≠ A) do
   if p.sv ≥ A.x
      then p := pλ
      else p := pρ;
if p.Punkt ist definiert
   then {p.Punkt = A} Schritt 2 ausführen
   else A kommt nicht vor; {fertig}

2. Schritt: {Entfernen und nachfolgende Punkte hochziehen}
procedure Entfernen (p : Knoten);
{anfangs ist p.Punkt = A}
entferne p.Punkt, d.h. setze p.Punkt := undefiniert;
Fall 1: [pλ.Punkt ist definiert, und pρ.Punkt ist definiert]
   if pλ.Punkt.y < pρ.Punkt.y
      then
         begin
            p.Punkt := pλ.Punkt;
            Entfernen(pλ)
         end
      else
         begin
            p.Punkt := pρ.Punkt;
```

Entfernen(p_ρ)
end;
Fall 2: [p_λ.*Punkt ist definiert, aber* p_ρ.*Punkt nicht*]
p.*Punkt* := p_λ.*Punkt*;
Entfernen(p_λ);
Fall 3: [p_ρ.*Punkt ist definiert, aber* p_λ.*Punkt nicht*]
p.*Punkt* := p_ρ.*Punkt*;
Entfernen(p_ρ);
Fall 4: [*weder* p_λ.*Punkt noch* p_ρ.*Punkt ist definiert*]
{*Hochziehen beendet*} *fertig!*

Entfernt man beispielsweise aus dem letzten Baum im angegebenen Beispiel den Punkt $(5,1)$, müssen nacheinander die Punkte $(4,2)$, $(1,3)$ und $(2,4)$ hochgezogen werden. Man erhält den Baum von Abbildung 7.25.

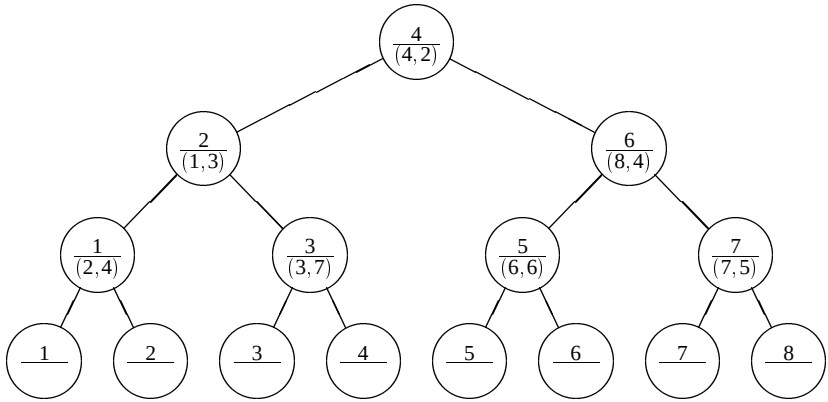


Abbildung 7.25

Es dürfte damit unmittelbar klar sein, daß das Einfügen und Entfernen von Punkten aus der ursprünglich gegebenen Menge von N Punkten stets in $O(\log N)$ Schritten möglich ist. Denn das Skelett des Prioritäts-Suchbaumes hat eine durch $\lceil \log_2 N \rceil$ beschränkte Höhe.

Wir überlegen uns nun, wie man alle in einem Prioritäts-Suchbaum gespeicherten Punkte (x,y) findet, deren x-Koordinaten in einem Bereich $[x_l, x_r]$ liegen und deren y-Koordinaten unterhalb eines Schwellenwertes y_0 bleiben.

Weil jeder Prioritäts-Suchbaum ein Suchbaum für die x-Werte ist, kann man den Bereich der Knoten mit zulässigen x-Werten von Punkten leicht eingrenzen. Unter diesen befinden sich die Knoten mit einem zulässigen y-Wert in einem Präfix des Baumes, d.h. sobald man auf einem Pfad von der Wurzel zu einem Blatt auf einen Punkt mit y-Wert $> y_0$ stößt, kann man die Ausgabe an dieser Stelle abbrechen. Grob vereinfacht

kann der Bereich der zulässigen Punkte wie in Abbildung 7.26 angegeben dargestellt werden.

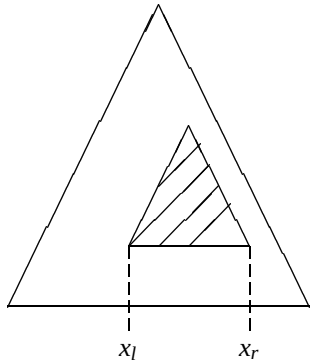


Abbildung 7.26

Jedem Knoten des Skeletts des Prioritäts-Suchbaumes kann man ein Intervall möglicher x-Werte eines an dem Knoten gespeicherten Punktes zuordnen. An der Wurzel ist das Intervall der gesamte zulässige x-Bereich, an den Blättern besteht er nur noch aus dem jeweiligen Splitwert. Um die Punkte zu bestimmen, deren x-Werte im vorgegebenen Bereich $[x_l, x_r]$ liegen, muß man höchstens die Knoten inspizieren, deren zugehörige Intervalle einen nichtleeren Durchschnitt mit dem Intervall $[x_l, x_r]$ haben. Das zeigt Abbildung 7.27.

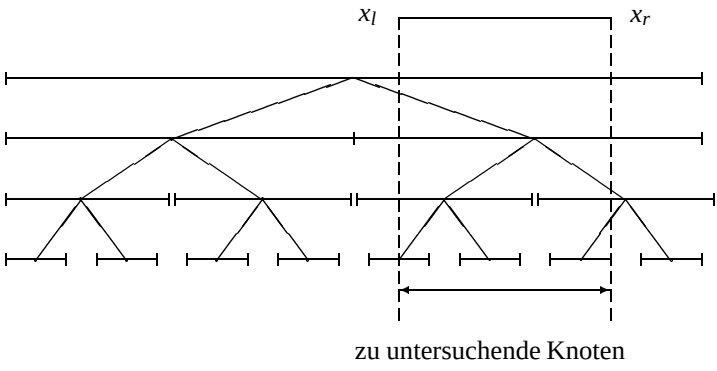


Abbildung 7.27

Den Bereich der höchstens in Frage kommenden Knoten kann man so abgrenzen: Man benutzt den Prioritäts-Suchbaum als Suchbaum für die Grenzen x_l und x_r des gegebenen Intervalls $[x_l, x_r]$. Alle Knoten auf den Suchpfaden von der Wurzel nach x_l bzw. x_r sowie sämtliche Knoten im Baum, die rechts vom Suchpfad nach x_l und links vom Suchpfad nach x_r liegen, können Punkte speichern, deren x-Wert in das gegebene Intervall fällt.

Unter diesen Knoten müssen diejenigen bestimmt werden, die einen Punkt mit y-Wert $\leq y_0$ gespeichert haben. Da die y-Werte von Punkten auf jedem Pfad von der Wurzel zu den Blättern zunehmen, kann man die gesuchten Punkte berichten in einer Anzahl von Schritten, die proportional zur Höhe des Skeletts und zur Anzahl k der berichteten Punkte ist, d.h. in $O(\log N + k)$ Schritten.

Kehren wir zurück zum Ausgangsproblem, die Menge aller Paare sich schneidender Rechtecke in einer Menge von N gegebenen Rechtecken in der Ebene zu bestimmen: Dieses Problem kann mit Hilfe des Scan-line-Verfahrens und Prioritäts-Suchbäumen zur Verwaltung der jeweils gerade aktiven Intervalle, d.h. der Schnitte der Scan-line mit den Rechtecken, in Zeit $O(N \log N + k)$ und Platz $O(N)$ gelöst werden. Dabei ist k die Anzahl der zu berichtenden Paare.

Für eine praktische Implementation mag es wünschenswert sein, anstelle einer Skelettstruktur der Größe $\Theta(N)$ während des Hinüberschwenkens der Scan-line über die Eingabe eine volldynamische Struktur zu verwenden, deren Größe sich der Anzahl der jeweils gerade aktiven Rechtecke anpaßt. Wie im Falle von Segment-Bäumen und Intervall-Bäumen kann man auch im Falle von Prioritätssuchbäumen eine auf einer geeigneten Variante von balancierten Bäumen gegründete, volldynamische Variante von Prioritätssuchbäumen entwerfen, die das Einfügen und Enfernen eines Intervalls in $O(\log n)$ Schritten erlaubt, wenn n die Anzahl der gerade gespeicherten Intervalle ist, und die es erlaubt, alle k Punkte in einem S-gegründeten Bereich in Zeit $O(\log n + k)$ zu berichten. Man vergleiche hierzu [120].

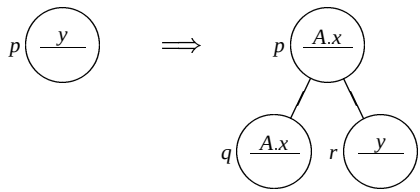
Wir skizzieren hier, wie man eine volldynamische Variante von Prioritäts-Suchbäumen erhält, die analog zu natürlichen Suchbäumen (random trees) zu einer gegebenen Folge von Punkten gebildet werden können und damit das Einfügen und Entfernen eines Punktes im Mittel in $O(\log n)$ Schritten erlauben. An Stelle des starren Skeletts verwenden wir als Suchstruktur einen natürlichen und damit von der Reihenfolge der Punkte abhängigen Blattsuchbaum. D.h. das Einfügen eines Punktes $A = (A.x, A.y)$ in den anfangs leeren Baum, der aus einem einzigen Knoten mit einem fiktiven Splitwert ∞ besteht, geschieht in zwei Phasen.

1. Phase: Suchbaum-Erweiterung

In dem bisher erzeugten Blattsuchbaum wird ein neues Blatt mit (Split-)Wert $A.x$ erzeugt und zwar so, daß im entstehenden Blattsuchbaum die x-Werte aller bisher eingefügten Punkte als Splitwerte der Blätter in aufsteigend sortierter Reihenfolge erscheinen und an jedem inneren Knoten als Splitwert stets das Maximum der Splitwerte im linken Teilbaum steht. Wir beginnen mit einer Suche im bisherigen Baum nach $A.x$:

$p := Wurzel$;
while (p *ist kein Blatt*) **do**
  if $A.x \leq p.sv$
    then $p := p_\lambda$
    else $p := p_\rho$

So findet man ein Blatt p mit Splitwert $p.sv$. Anfangs gilt trivialerweise $A.x \leq p.sv$. Wir sorgen dafür, daß diese Bedingung stets erhalten bleibt, indem wir p durch einen Knoten mit zwei Söhnen q und r ersetzen: Der linke Sohn q erhält als Splitwert $A.x$, der rechte als Splitwert $p.sv$ und p den neuen Splitwert $A.x$:



Ein eventuell bei p gespeicherter Punkt bleibt dort. Es gibt dann zu jedem Splitwert x genau ein Blatt mit Splitwert x und einen inneren Knoten auf dem Suchpfad zu diesem Blatt, der ebenfalls x als Splitwert hat.

2. Phase: Ablegen des Punktes A

Der Punkt A wird seinem y-Wert $A.y$ entsprechend auf dem Suchpfad von der Wurzel zum Blatt mit Splitwert $A.x$ so nah wie möglich an der Wurzel abgelegt. Diese Phase unterscheidet sich überhaupt nicht von dem für die Skelett-Variante von Prioritäts-Suchbäumen erklärten Einfügeverfahren.

Beispiel: Es sollen der Reihe nach die Punkte

$$(6,4),(7,3),(2,2),(4,6),(1,5),(3,9),(5,1)$$

in den anfangs leeren Baum eingefügt werden. Einfügen des ersten Punktes $(6,4)$ liefert den Baum von Abbildung 7.28.

Einfügen von $(7,3)$ liefert den Baum von Abbildung 7.29. Zum Einfügen des nächsten Punktes $(2,2)$ wird zunächst der unterliegende Suchbaum erweitert. Man erhält den Baum von Abbildung 7.30.

Ablegen des Punktes $(2,2)$ verdrängt den Punkt $(7,3)$ von der Wurzel und liefert den Baum von Abbildung 7.31.

Fügt man die restlichen Punkte auf dieselbe Weise ein, so erhält man schließlich den Prioritäts-Suchbaum von Abbildung 7.32.

Das *Entfernen* eines Punktes A verläuft umgekehrt zum Einfügen: Man sucht zunächst mit Hilfe des x-Wertes $A.x$ einen Knoten p, an dem A abgelegt wurde. Die durch das Entfernen dieses Punktes entstehende Lücke schließt man durch (iteriertes) Hochziehen von Punkten wie im Falle der Skelettstruktur. Man muß jetzt noch die unterliegende Suchbaumstruktur um ein Blatt mit Splitwert $A.x$ und einen inneren Knoten mit

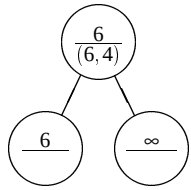


Abbildung 7.28

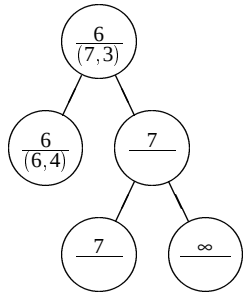


Abbildung 7.29

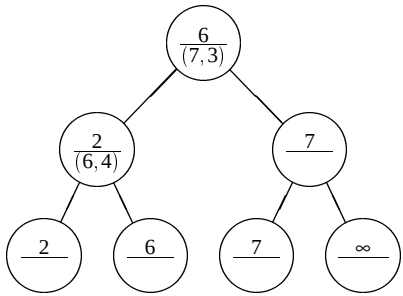


Abbildung 7.30

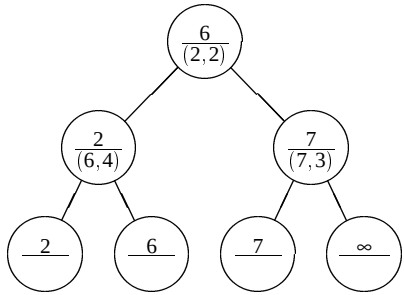


Abbildung 7.31

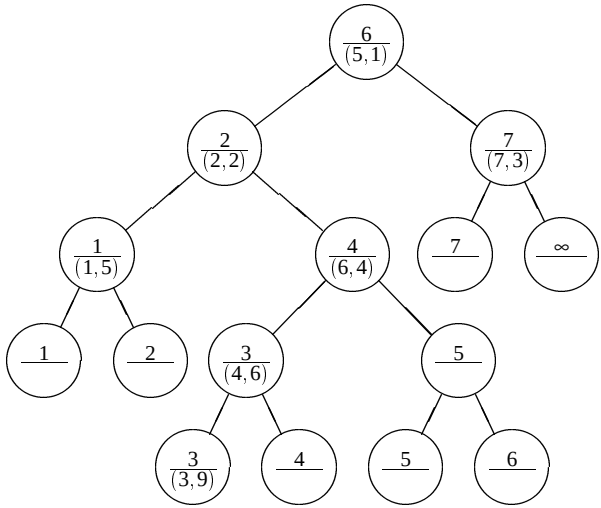


Abbildung 7.32

gleichem Splitwert verkleinern. Das geschieht wie folgt: Man sucht nach dem zu entfernenden Blatt p mit Splitwert $A.x$; unterwegs trifft man bei dieser Suche auch auf den inneren Knoten q mit Splitwert $A.x$. Zwei Fälle sind möglich:

Fall 1: [p ist rechter Sohn seines Vaters, vgl. Abbildung 7.33]

Dann muß der Splitwert des Vaters φp von p der symmetrische Vorgänger von $A.x$ sein. Man kann also φp durch den linken Teilbaum von φp ersetzen und den Splitwert $A.x$ von q durch den Splitwert y von φp ersetzen, ohne daß dadurch Suchpfade nach anderen x-Werten, die von $A.x$ verschieden sind, beeinflußt werden. Bei p kann höchstens der Punkt A abgelegt gewesen sein, den wir ja entfernt haben. Ein eventuell bei φp abgelegter Punkt B muß seinem y-Wert entsprechend in den linken Teilbaum von φp hinunterwandern. Dort ist Platz! Denn es gibt dort ein Blatt mit Splitwert $B.x$.

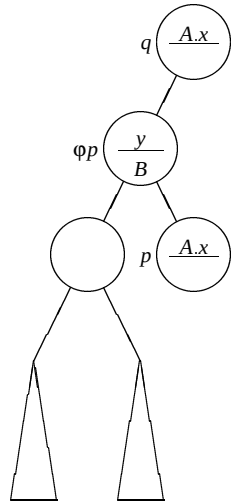


Abbildung 7.33

Fall 2: [p ist linker Sohn seines Vaters, vgl. Abbildung 7.34]

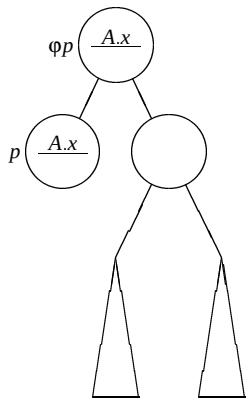


Abbildung 7.34

Dann muß der Vater φp von p ebenfalls $A.x$ als Splitwert haben; denn der Splitwert jedes inneren Knotens ist das jeweilige Maximum der Splitwerte im linken Teilbaum. Ersetzt man also (p und) φp durch den rechten Teilbaum von φp, wird jede Suche nach einem im rechten Teilbaum von φp stehenden x-Wert nach wie vor richtig gelenkt.

Einen eventuell bei φp abgelegten Punkt B muß man in den rechten Teilbaum von φp hinunter wandern lassen.

Verfolgen wir als Beispiel das Entfernen des Punktes $(5,1)$ aus dem zuletzt erhaltenen Baum auf den vorhergehenden Seiten: Der Punkt ist an der Wurzel abgelegt. Die nach dem Entfernen entstehende Lücke wird zunächst durch Hochziehen der Punkte $(2,2)$, $(6,4)$, $(4,6)$ und $(3,9)$ geschlossen. Dann werden das Blatt und der innere Knoten mit Splitwert 5 entfernt und man erhält den Baum von Abbildung 7.35. Entfernen des Punktes $(6,4)$ ergibt den Baum von Abbildung 7.36 (vgl. Fall 1).

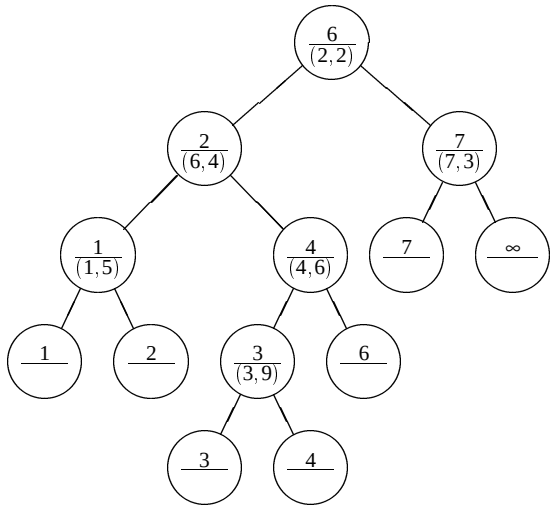


Abbildung 7.35

7.5 Das Zickzack-Paradigma

Wir haben bisher in erster Linie Probleme diskutiert, die Mengen iso-orientierter Objekte in der Ebene betrafen. In der Tat ist dieser Fall besonders gründlich untersucht worden mit dem Ergebnis, daß zahlreiche effiziente oder sogar optimale Algorithmen für diesen Fall gefunden wurden, vgl. [195]. In Wirklichkeit hat man es aber häufig mit Mengen beliebig orientierter Objekte im d-dimensionalen Raum, $d \geq 2$ zu tun. Beispiele sind Mengen beliebig orientierter Liniensegmente in der Ebene und Polygone oder polygonal begrenzte Flächen im dreidimensionalen Raum. Wir wollen in diesem Abschnitt der Frage nachgehen, ob und gegebenenfalls unter welchen Bedingungen sich

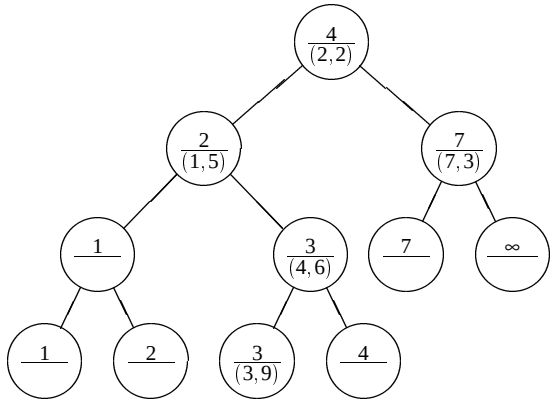


Abbildung 7.36

ein Verfahren zur Lösung eines algorithmischen Problems für Mengen iso-orientierter Objekte verallgemeinern läßt zu einem Verfahren zur Lösung des entsprechenden Problems für Mengen beliebig orientierter Objekte. Dabei möchte man natürlich möglichst wenig an Effizienz einbüßen. Wir werden zeigen, daß das für eine große Klasse von Verfahren möglich ist, genauer: für solche Verfahren, die dem Scan-line-Prinzip folgen und von halbdynamischen Skelettstrukturen Gebrauch machen, wie wir sie in Abschnitt 7.4 vorgestellt haben. Wir erläutern das Prinzip des Übertragens eines Verfahrens vom iso-orientierten auf den allgemeinen Fall am Beispiel des Schnittproblems für Polygone. Das ist folgendes Problem: Gegeben sei eine Menge von p Polygonen mit insgesamt N Kanten in der Ebene. Gesucht sind alle Paare sich schneidender Polygone. Zwei Polygone schneiden sich, wenn sich entweder zwei Polygonkanten dieser Polygone schneiden oder das eine Polygon das andere vollständig einschließt. Wir lassen nur einfach geschlossene Polygone zu. Die Polygone können, müssen aber natürlich nicht konvex sein. Im Falle, daß alle Polygone konvex sind, kann die Lösung des Polygonschnittproblems vereinfacht werden. Die im folgenden skizzierte Lösung des Polygonschnittproblems kann leicht auf den Fall ausgedehnt werden, daß die gegebenen Polygone nicht sämtlich einfach geschlossene Polygone sind, sondern von allgemeinerer Art sind, d.h. z.B. Löcher enthalten. Jedes einfach geschlossene Polygon kann man sich gegeben denken als Folge seiner in Umlaufrichtung angeordneten Eckpunkte. Durchläuft man die Eckpunkte in dieser Reihenfolge, kehrt man zum Ausgangspunkt zurück; das Innere des Polygons soll dabei stets rechts liegen. Wir wollen das Polygonschnittproblem ähnlich wie das Rechteckschnittproblem lösen, indem wir dem Scan-line-Prinzip folgen und eine horizontale Scan-line von oben nach unten über die Menge der gegebenen Polygone hinwegschwenken, vgl. Abbildung 7.37.

Dabei merken wir uns wie im Fall des Rechteckschnittproblems die Schnitte der Polygone mit der Scan-line als eindimensionale Intervalle in einer dynamisch veränderlichen Datenstruktur. Was sind die Unterschiede zwischen dem iso-orientierten und diesem allgemeineren Fall?

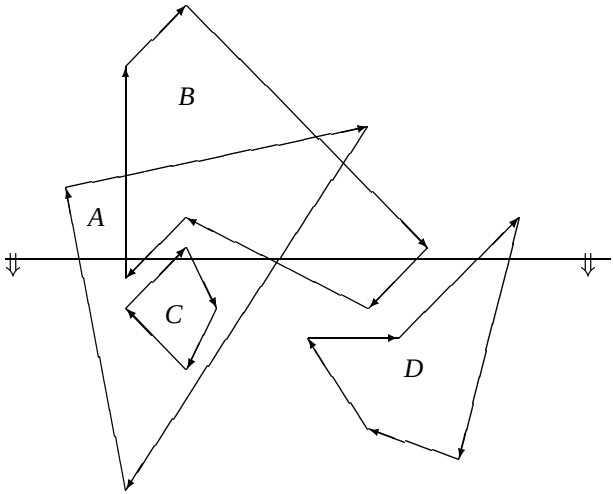


Abbildung 7.37

Zunächst bemerken wir, daß ein Polygon mehr Kantenschnitte mit der Scan-line haben kann, z.B. vier wie das Polygon *B* in Abbildung 7.37, und nicht nur zwei wie im Falle von Rechtecken. Die Anzahl der Schnitte kann die Größenordnung $\Omega(N)$ erreichen, wenn N die Gesamtzahl der Kanten ist. Allerdings kann die Scan-line höchstens zwei Kanten eines konvexen Polygons schneiden. In jedem Fall werden die Polygone durch wachsende und schrumpfende Intervalle auf der Scan-line repräsentiert. Das ist der zweite Unterschied zum iso-orientierten Fall. Trifft dort die Scan-line den oberen Rand eines Rechtecks, so wird das durch seinen linken und rechten Rand gegebene Intervall in die Menge der aktiven Intervalle aufgenommen und bleibt darin unverändert, bis die Scan-line den unteren Rechteckrand erreicht. Im Falle eines Polygons hingegen wachsen und schrumpfen diese Intervalle. Schließlich ist nicht zu sehen, wie man ein diskretes Raster finden könnte, das als Grundlage zum Bau einer halbdynamischen Skelettstruktur dienen könnte, das also Platz für alle beim Hinunterschwenken der Scan-line auftretenden Intervalle bietet. Dieser zuletzt genannte Unterschied ist der entscheidende. Denn das Wachsen und Schrumpfen von Intervallen, die die Schnitte der Scan-line mit den Polygonen bilden, kann man einfach ignorieren, solange man die jeweils korrekte, relative Anordnung der Intervallgrenzen aufrecht erhalten kann. Ferner kann man die Schnitte eines jeden Polygons mit der Scan-line in einem dem Polygon zugeordneten (balancierten) Suchbaum speichern (vgl. weiter unten). Es bleibt damit das Hauptproblem, einen Ersatz für das im iso-orientierten Fall offensichtlich vorhandene diskrete Raster zu finden, auf das man eine Skelettstruktur gründen kann. Im Falle des Rechteckschnittproblems wird nämlich das Raster von der Menge aller linken und rechten Rechteckseiten gebildet: Das ist eine angeordnete Menge von diskreten Punkten auf der x-Achse derart, daß jedes im Verlauf eines Scans von oben nach unten in der Vertikalstruktur abzuspeichernde Intervall ein Intervall über diesem Punktraster ist. Die Polygonkanten bilden dagegen eine Menge beliebig orientierter Liniensegmente

in der Ebene, die nicht in ähnlicher Weise eine Rasterung der x-Achse induzieren. Wie wir bereits bei der Lösung des allgemeinen Segmentschnittproblems im Abschnitt 7.2.3 gesehen haben, kann man auch nicht erwarten, daß die Polygonkanten in eine für den ganzen Scan feste Reihenfolge gebracht werden können, die für die jeweils von der Scan-line geschnittenen Kanten mit der Von-links-nach-rechts-Reihenfolge der Schnittpunkte längs der Scan-line übereinstimmt. Man wird höchstens eine lokal gültige Anordnung verlangen können, die an jedem Schnittpunkt zweier Kanten verändert werden muß.

Wir suchen also eine *Anfangsanordnung* der die Polygonkanten bildenden Menge von Liniensegmenten. Diese Anfangs-Anordnung liefert das zu Beginn des Scans *lokal gültige Raster*. Das lokal gültige Raster wird an jedem Schnittpunkt zweier Kanten dadurch verändert, daß die am Schnitt beteiligten Kanten ihre Plätze tauschen. Das lokal gültige Raster ist unser Ersatz für das im iso-orientierten Fall *global* gültige Raster der linken und rechten Rechteckseiten. Wir werden also jedes Polygon durch ein oder mehrere Intervalle über dem lokal gültigen Raster repräsentieren ebenso, wie wir im iso-orientierten Fall jedes Rechteck durch ein Intervall über dem globalen Raster der linken und rechten Rechteckseiten repräsentiert haben. Das einzige Problem besteht darin, eine geeignete Anfangsanordnung zu finden, mit der wir den Scan von oben nach unten beginnen können.

Wir können dieses Problem präziser formulieren, wenn wir den Begriff der für einen Scan von oben nach unten geeigneten Anfangsanordnung einer gegebenen Menge von Liniensegmenten in der Ebene wie folgt definieren:

Eine totale Ordnung "$<$" einer Menge von Liniensegmenten in der Ebene heißt für einen Scan von oben nach unten *geeignete Anfangsanordnung*, wenn folgender Algorithmus *hsweep* ausführbar und korrekt ist, d.h. die als Kommentare vermerkten Zusicherungen gelten an den angegebenen Stellen.

```
Algorithmus hsweep
S := Folge der Liniensegmente in Anordnung "<", alle
     Segmente als "nicht vorhanden" markiert;
Q := Folge der oberen Endpunkte, unteren Endpunkte und
     Schnittpunkte von Liniensegmenten in absteigender
     y-Reihenfolge;
while Q ist nicht leer do
  begin
    p := nächster Punkt von Q;
    case Art von p of
      p oberer Endpunkt eines Segments s :
        markiere s als "vorhanden" in S;
      p unterer Endpunkt eines Segments s :
        markiere s als "nicht vorhanden" in S;
      p Schnittpunkt zweier Segmente s und t :
        {s und t sind als "vorhanden" markiert und in S sind
         keine zwischen s und t stehenden Segmente
         als "vorhanden" markiert}
        ersetze S durch die Anordnung, die durch
          Vertauschen von s und t in S entsteht
```

```
        end {case}
        {die Anordnung der als "vorhanden" markierten Elemente
         in S stimmt überein mit der Links-nach-rechts-Anordnung
         dieser Segmente längs einer unmittelbar unterhalb von p
         verlaufenden horizontalen Geraden}
    end {while}
{alle Elemente von S sind als "nicht vorhanden" markiert}
end {Algorithmus hsweep}
```

Die Anordnung der Segmente in S zwischen je zwei aufeinanderfolgenden Schnittpunkten (aus Q) ist das zwischen diesen Punkten lokal gültige Raster: Es bietet Platz für alle zwischen diesen Punkten beginnenden Segmente durch Eintrag in das "Skelett" S in der richtigen Anordnung von links nach rechts. Das Finden einer für einen Scan von oben nach unten geeigneten Anfangsanordnung ist trivial, wenn die gegebene Menge nur aus vertikalen Liniensegmenten besteht: Die gesuchte Anordnung ist dann einfach die Anordnung der Segmente in aufsteigender x-Reihenfolge. Das ist der iso-orientierte Fall, in dem keine Schnittpunkte vorkommen und die Anfangsanordnung nicht mehr verändert wird.

Für eine beliebige, gegebene Menge von Liniensegmenten in der Ebene ist das Finden der für einen Scan von oben nach unten geeigneten Anfangsanordnung nicht so leicht. Betrachten wir folgendes Beispiel einer Menge aus vier Segmenten A, B, C, D wie in Abbildung 7.38.

Um zu prüfen, ob $A < B < D < C$ eine für einen Scan von oben nach unten geeignete Anfangsanordnung ist, verfolgen wir, wie sich S an den ersten sieben "Haltepunkten" aus Q verändert; dabei stellen wir fest, daß die Anordnung der in S als "vorhanden" markierten Elemente nicht mit der Von-links-nach-rechts-Reihenfolge übereinstimmt, sobald der obere Endpunkt von D angetroffen wurde. Die Ordnung $A < B < D < C$ ist also nicht die gesuchte Anfangsanordnung.

Um zu einer gegebenen Menge von Liniensegmenten in der Ebene die für einen Scan von oben nach unten geeignete Anfangsanordnung zu finden, genügt es allerdings, gewissermaßen die richtige Brille aufzusetzen. Wir fassen die Menge von sich möglicherweise schneidenden Segmenten auf als eine Menge von sich nicht schneidenden Zickzacks, die sich höchstens berühren können. Zu jedem Liniensegment s assoziieren wir ein *Zickzack* z_s wie folgt: z_s beginnt am obersten Punkt von s. Dann folgt man s in absteigender y-Richtung bis zum ersten Schnittpunkt mit einem Segment s'. Jetzt folgt man s' statt s bis zum nächsten Schnittpunkt usw., bis schließlich der unterste Punkt eines Segments erreicht ist. Dort endet das Zickzack z_s.

Abbildung 7.39 zeigt noch einmal die Menge von vier Segmenten A, B, C, D und die zugehörige Menge von vier Zickzacks z_A, z_B, z_C, z_D.

Zickzacks schneiden sich nicht. Sie berühren sich nur an den Schnittpunkten der Segmente. Zickzacks sind monoton in y-Richtung. Man kann sie sich als elastische, geknickte Bänder vorstellen: Zieht man sie an den Enden in y-Richtung straff, berühren sie sich schließlich gar nicht mehr und werden vertikal. Das veranschaulicht intuitiv, daß Zickzacks die Rolle spielen, die vertikale Segmente im iso-orientierten Fall spielten.

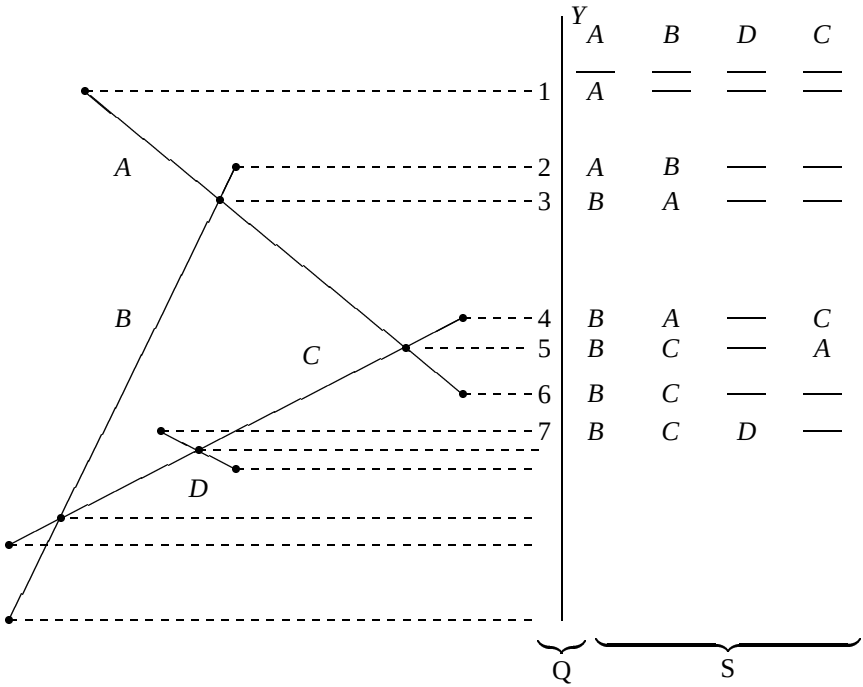


Abbildung 7.38

Zickzacks sind auf natürliche Art angeordnet durch eine vollständige Ordnung "left$^+$| above". Seien z und z' zwei Zickzacks; dann gilt: z *left*$^+$| *above* z' genau dann, wenn entweder eine horizontale Gerade l existiert, die z und z' schneidet und der Schnittpunkt $z \cap l$ links vom Schnittpunkt $z' \cap l$ liegt, oder z vollständig oberhalb von z' liegt, d.h. der untere Endpunkt von z oberhalb vom oberen Endpunkt von z'. Für das in Abbildung 7.39 angegebene Beispiel von vier Zickzacks ist die left$^+$| above-Ordnung z_A, z_D, z_B, z_C.

Die left$^+$| above-Ordnung der Zickzacks induziert eine Anordnung der ursprünglich gegebenen Menge von Liniensegmenten. Für das obige Beispiel ist es gerade die Anordnung *A,D,B,C*; ganz allgemein ist die Zickzack-Ordnung zugleich die auf der Menge der Anfangsstücke induzierte Anordnung der Segmente. Es ist nun nicht überraschend, daß diese über die left$^+$| above-Ordnung von Zickzacks induzierte Anordnung von Liniensegmenten genau die gesuchte, für einen Scan von oben nach unten geeignete Anfangsanordnung von Liniensegmenten ist. Wir überlassen den Beweis dem Leser.

Man kann zeigen, daß für eine gegebene Menge von N Liniensegmenten in der Ebene mit k Schnittpunkten die left$^+$| above-Anordnung der zugehörigen Menge von Zickzacks und damit eine für einen Scan von oben nach unten geeignete Anfangsanordnung der Liniensegmente in Zeit $O((N+k)\log N)$ und Platz $O(N)$ berechnet werden kann (vgl. [139]).

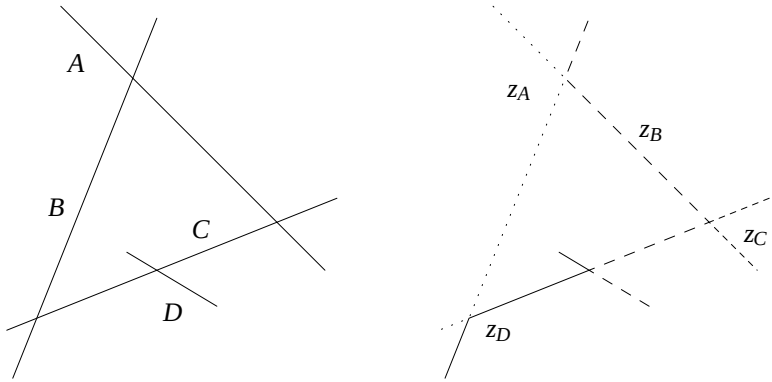


Abbildung 7.39

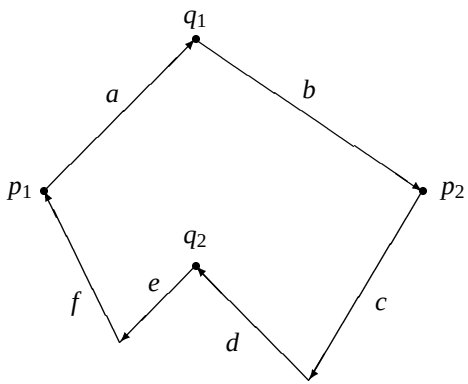
Wir kehren nun zum Ausgangsproblem zurück und zeigen, wie man das Polygonschnittproblem nach dem Scan-line-Prinzip löst.

Das Verfahren kann ganz grob wie folgt beschrieben werden: Zunächst wird die gegebene Menge von Polygonen in eine (möglichst kleine) Menge von Zickzacks zerlegt. Für diese Zickzacks bestimmt man die left$^+$| above-Ordnung; das liefert zugleich die für einen Scan von oben nach unten geeignete Anfangsanordnung der Polygonkanten. Man nimmt diese Anfangsanordnung als anfangs lokal gültiges Raster und baut darauf eine Skelettstruktur. In dieser Skelettstruktur speichert man die jeweils gerade aktiven Polygone als Intervalle über dem lokal gültigen Raster wie im iso-orientierten Fall. Im Unterschied zum iso-orientierten Fall muß man allerdings das lokal gültige Raster und damit die Skelettstruktur an allen Schittpunkten von Polygonkanten verändern.

Wir beschreiben nun die einzelnen Schritte genauer: Zuerst muß die gegebene Menge von Polygonen in Zickzacks zerlegt werden. Das könnte man natürlich so machen, daß man die Menge von Polygonen als Menge von Liniensegmenten auffaßt, die durch die Menge der Polygonkanten gegeben ist. Haben die Polygone insgesamt N Kanten, würde man also auch N Zickzacks bekommen. Man kommt jedoch im allgemeinen mit wesentlich weniger Zickzacks aus, da es nicht nötig ist, an einem "Knick", d.h. an einem Punkt, an dem zwei Polygonkanten zusammenstoßen und dessen y-Wert nicht ein lokales Maximum ist, ein neues Zickzack zu beginnen. Es genügt, an jedem Punkt p mit lokal maximalem y-Wert ein neues Paar von Zickzacks zu beginnen. Wir wollen die an einem solchen Punkt zusammenstoßenden Kanten *Top-Segmente* des jeweiligen Polygons nennen. Abbildung 7.40 zeigt ein Beispiel.

Das Polygon aus Abbildung 7.40 kann in vier Zickzacks zerlegt werden: *a–f, e, d, b–c*. Je ein Paar beginnt an den Punkten q_1 und q_2. Die von der left$^+$| above-Ordnung der Zickzacks induzierte Ordnung der Top-Segmente ist *a, e, d, b*.

Da jedes konvexe Polygon genau eine Ecke mit maximalem y-Wert hat, zerfällt eine Menge von p konvexen Polygonen in genau $2p$ Zickzacks. Das Beispiel aus Abbildung 7.41 zeigt eine Menge von vier konvexen Polygonen A, B, C, D. In dieser Abbil-



a, b und e, d
sind Top-Segmente,
p_1 und p_2 sind
Knickpunkte

Abbildung 7.40

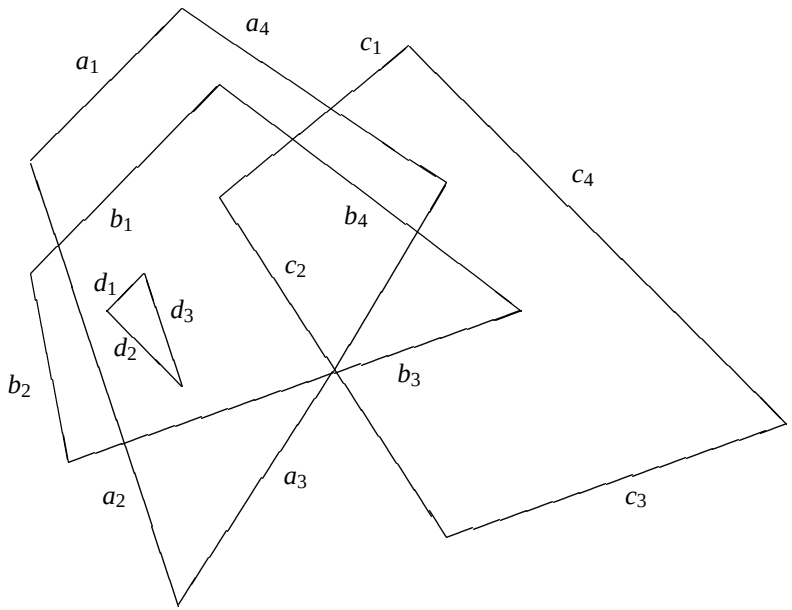


Abbildung 7.41

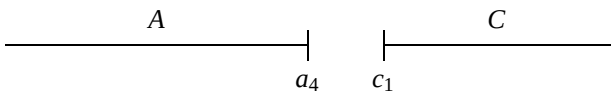
dung hat Polygon A die Kanten a_1, a_2, a_3, a_4; Polygon B hat die Kanten b_1, b_2, b_3, b_4; Polygon C hat die Kanten c_1, c_2, c_3, c_4; Polygon D hat die Kanten d_1, d_2, d_3.

Die Menge dieser Polygone zerfällt in acht Zickzacks; die Anordnung dieser Zickzacks in left$^+$| above-Ordnung induziert die folgende, für einen Scan von oben nach unten geeignete Anfangsanordnung.

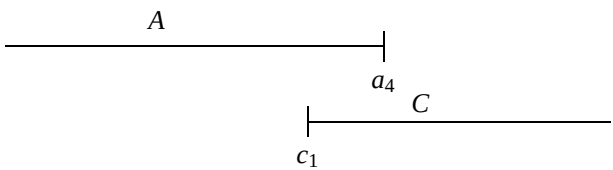
$$a_1,\ b_1,\ d_1,\ d_3,\ b_4,\ a_4,\ c_1,\ c_4.$$

Diese Anordnung liefert das zu Beginn des Scans von oben nach unten lokal gültige Raster: Die Polygone können als Intervalle über diesem Raster dargestellt werden.

Schwenkt man die Scan-line von oben nach unten über die Menge dieser Polygone, wird zuerst der oberste Punkt des Polygons A angetroffen. Das Polygon A wird als Intervall $[a_1, a_4]$ über dem Raster $a_1\ b_1\ d_1\ d_3\ b_4\ a_4\ c_1\ c_4$ repräsentiert; d.h. das Intervall $[a_1, a_4]$ wird in die (anfangs leere) Menge der gerade aktiven Intervalle aufgenommen. Dann wird am zweiten Haltepunkt der Scan-line der oberste Punkt des Polygons C angetroffen; C kann als Intervall $[c_1, c_4]$ über demselben Raster repräsentiert werden. Der nächste Haltepunkt ist der oberste Punkt des Polygons B, das ebenfalls als Intervall $[b_1, b_4]$ über diesem Raster repräsentiert werden kann. Das Raster bleibt gültig, bis der erste Kantenschnittpunkt angetroffen wird. Das ist im obigen Beispiel der Schnittpunkt zwischen a_4 und c_1. Die Situation unmittelbar oberhalb des Schnittpunktes



muß ersetzt werden durch die unmittelbar unterhalb des Schnittpunktes bis zum nächsten Haltepunkt gültige Situation:



Diese Änderung kann man so erreichen: Das das Polygon A repräsentierende Intervall mit rechtem Endpunkt a_4 und das das Polygon C repräsentierende Intervall mit linkem Endpunkt c_1 werden aus der Menge der aktiven Intervalle entfernt. Dann wird das lokal gültige Raster verändert, indem a_4 und c_1 ihre Plätze tauschen. Anschließend werden die A und C repräsentierenden Intervalle wieder in die Menge der aktiven Intervalle eingefügt. Für die ersten sieben Haltepunkte der Scan-line geben wir in Tabelle 7.1 das jeweils lokal gültige Raster und die jeweils aktive Menge von Intervallen zwischen den Haltepunkten an.

Jetzt besteht praktisch kein Unterschied mehr zwischen dem iso-orientierten Fall, also dem Rechteckschnittproblem, und dem allgemeinen Fall, also dem Polygonschnittproblem. Zu jedem Zeitpunkt wird die Menge der gerade aktiven, d.h. von der Scan-line geschnittenen Objekte (Rechtecke bzw. Polygone) repräsentiert durch eine dynamisch

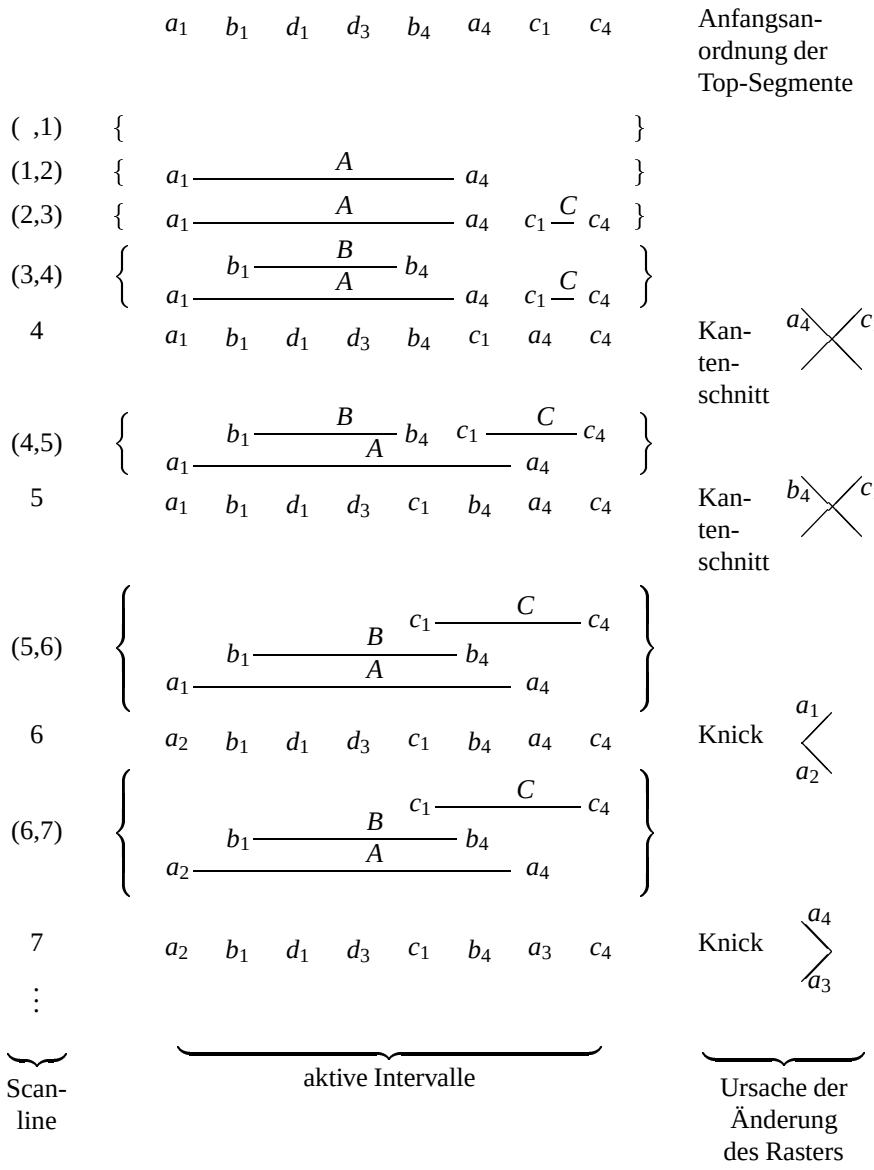

a_1 b_1 d_1 d_3 b_4 a_4 c_1 c_4
Anfangsanordnung der Top-Segmente
(,1) { }
(1,2) { a_1 A a_4 }
(2,3) { a_1 A a_4 c_1 C c_4 }
(3,4) { b_1 B b_4 / a_1 A a_4 c_1 C c_4 }
4 a_1 b_1 d_1 d_3 b_4 c_1 a_4 c_4
Kantenschnitt a_4 c_1
(4,5) { b_1 B b_4 c_1 C c_4 / a_1 A a_4 }
5 a_1 b_1 d_1 d_3 c_1 b_4 a_4 c_4
Kantenschnitt b_4 c_1
(5,6) { c_1 C c_4 / b_1 B b_4 / a_1 A a_4 }
6 a_2 b_1 d_1 d_3 c_1 b_4 a_4 c_4
Knick a_1 a_2
(6,7) { c_1 C c_4 / b_1 B b_4 / a_2 A a_4 }
7 a_2 b_1 d_1 d_3 c_1 b_4 a_3 c_4
Knick a_4 a_3
⋮
Scanline
aktive Intervalle
Ursache der Änderung des Rasters

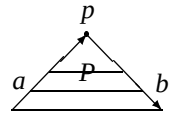
Tabelle 7.1

veränderliche Menge von Intervallen über einem jeweils lokal gültigen Raster. Die Überlappungsverhältnisse der jeweils aktiven Intervalle spiegeln genau die Schnittverhältnisse der von den Intervallen repräsentierten Polygone wieder. Die Intervalle kann man auch im nicht iso-orientierten Fall in ein anfangs leeres Skelett über dem jeweils gültigen Raster eintragen. Als Skelettstruktur kann man z.B. Segment- und Intervall-Bäume nehmen. Wir formulieren nun das Verfahren zur Bestimmung aller Schnitte in einer Menge von gegebenen Polygonen, indem wir den oben angegebenen Algorithmus *hsweep* um die für dieses Problem spezifischen Details ergänzen; wir lassen aber immer noch zahlreiche Implementationsdetails offen.

Wir nehmen an, daß die Kantenschnitte schon berechnet, aber noch nicht berichtet sind. Die Kantenschnitte werden nämlich ohnehin benötigt, um die Zickzack-Zerlegung zu bestimmen. Man zerlegt die Menge der Polygone in Zickzacks, bestimmt die Anfangsanordnung der Top-Segmente und baut eine anfangs leere Skelettstruktur S über diesem (lokalen) Raster.

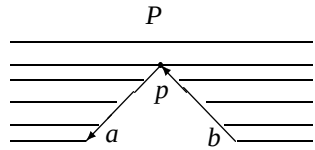
Algorithmus *Polygonschnitt*
{*berechnet zu einer Menge von Polygonen mit insgesamt N Kanten und k Kantenschnitten in der Ebene die Menge aller Paare von sich schneidenden Polygonen*}
$Q :=$ *Menge der oberen Endpunkte, unteren Endpunkte und Schittpunkte von Polygonkanten in abnehmender y-Reihenfolge*;
while *Q ist nicht leer* **do**
begin
$p :=$ *nächster Punkt von Q*;
case *Art von p* **of**

(1.) {*p ist konvexe Ecke eines Polygons P*}



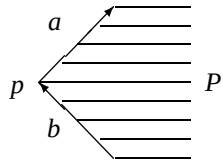
füge das P repräsentierende Intervall $[a,b]$ *in S ein*;
bestimme jedes Intervall $[a',b']$ *aus S, das ein Polygon Q repräsentiert und den Punkt p (bzw. eine der Kanten a oder b) enthält und berichte das Paar* (P,Q); {*dies ist eine* Aufspieß-*Anfrage*}

(2.) {*p ist konkave Ecke eines Polygons P*}

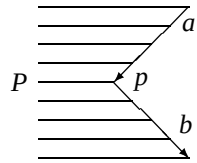


bestimme die a unmittelbar vorangehende Kante a' *und die b unmittelbar nachfolgende Kante* b' *von P in x-Richtung*;
{*oberhalb von p wird P durch das Intervall* $[a',b']$ *repräsentiert*}
entferne $[a',b']$ *aus S und füge* $[a',a]$ *und* $[b,b']$ *in S ein*;

(3.) {*p ist Knick*}

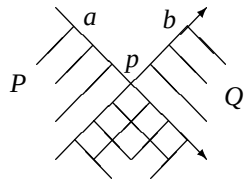


oder



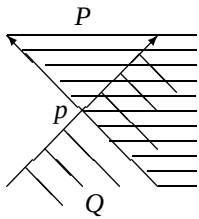
ersetze im lokalen Raster, also im Skelett von S, a durch b und ersetze alle Intervalle mit rechtem bzw. linkem Rand a durch solche mit rechtem bzw. linkem Rand b;

(4.1) {*p ist Schnittpunkt zweier Kanten a und b*}

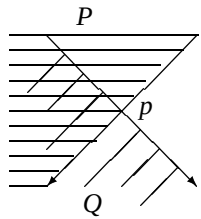


bestimme die a unmittelbar vorangehende Kante a' *von P und die b unmittelbar nachfolgende Kante* b' *von Q in x-Richtung*;
{*oberhalb von p wird P durch* $[a', a]$ *und Q durch* $[b, b']$ *repräsentiert*}
entferne $[a', a]$ *und* $[b, b']$ *aus S*;
vertausche im lokalen Raster, also im Skelett von S, a und b;
füge $[a', a]$ *und* $[b, b']$ *wieder ein in S*;
berichte das Paar (P, Q);

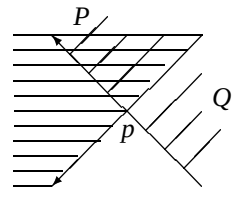
{*Fälle* (4.2) – (4.4) *werden analog zu Fall* (4.1) *behandelt*}



(4.2)



(4.3)



(4.4)

end {*case*}
end {*while*}
end {*Algorithmus Polygonschnitt*}

Wie kann man die an einer bestimmten Halteposition einer Kante eines Polygons P unmittelbar vorangehende bzw. unmittelbar nachfolgende Kante in x-Richtung bestimmen? Dazu merkt man sich zu jedem Polygon die jeweils gerade aktiven Kanten in Von-links-nach-rechts-Reihenfolge längs der Scan-line in einem P zugeordneten, balancierten Suchbaum. Da wir insgesamt nur N Kanten haben, können alle Bäume zusammen niemals mehr als $O(N)$ Platz beanspruchen. Das Einfügen und Entfernen von Kanten und das Bestimmen von unmittelbaren Vorgängern und Nachfolgern ist stets in $O(\log N)$ Schritten ausführbar.

Die Komplexität des Verfahrens zur Lösung des Polygonschnittproblems hängt jetzt davon ab, wie die Skelettstruktur S implementiert wird. Es ist offensichtlich, daß wir für S Analoga zu Segment- und Intervall-Bäumen bauen können. Der einzige Unterschied zu den entsprechenden Strukturen im iso-orientierten Fall besteht darin, daß wir von Zeit zu Zeit lokale Änderungen im Skelett vornehmen müssen. Die Größe bleibt dabei allerdings stets unverändert. Das Ersetzen eines Rasterpunktes durch einen neuen wie im Fall (3.) und das Vertauschen zweier Rasterpunkte, wie im Fall (4.), des oben angegebenen Algorithmus ist aber in jedem Fall in $O(\log N)$ Schritten möglich, da die Größe des Skeletts stets durch $O(N)$ beschränkt bleibt; im Falle konvexer Polygone sogar durch die Anzahl dieser Polygone. Die übrigen Operationen, nämlich das Einfügen und Entfernen von Intervallen und das Beantworten von Aufspieß-Anfragen, benötigen dieselbe Schrittzahl wie für gewöhnliche Segment- und Intervall-Bäume. Zählt man noch die Anzahl der Schritte hinzu, die für die Bestimmung der für den Scan von oben nach unten geeigneten Anfangsanordnung der Top-Segmente der gegebenen Polygone erforderlich ist, erhält man:

Für eine gegebene Menge von Polygonen mit insgesamt N Kanten und k Kantenschnitten kann man alle r Paare sich schneidender Polygone berichten in Zeit $O((N+k+r)\log N)$. Der benötigte Speicherplatz ist von der Größenordnung $O(N\log N)$, falls Analoga zu Segment-Bäumen verwendet werden, und $O(N)$, falls Analoga zu Intervall-Bäumen verwendet werden. In beiden Fällen dürfen allerdings die k Schnittpunkte nicht explizit gespeichert werden, wie wir es bei der Formulierung des oben angegebenen Verfahrens angenommen haben; vielmehr muß man sie im Verlaufe des Verfahrens noch einmal mitberechnen. Will man das nicht, ist der Speicherbedarf $O(N\log N+k)$ bzw. $O(N+k)$.

Das Verfahren zur Lösung des Polygonschnittproblems läßt sich verhältnismäßig leicht ausbauen, um ein Grundproblem der graphischen Datenverarbeitung zu lösen, das sogenannte *Hidden-Line-Eliminationsproblem*. Nehmen wir an, eine Menge polygonal begrenzter, ebener Flächen im dreidimensionalen Raum sei gegeben. Wir möchten wissen, welche Kanten sichtbar sind, wenn man aus dem Unendlichen von oben auf diese Flächen blickt. Das ist eine anschauliche Formulierung des Problems, die verdeckten Kanten einer dreidimensionalen Szene bei orthographischer Parallelprojektion zu bestimmen.

Wir nehmen natürlich an, daß die polygonal begrenzten, ebenen Flächen sich nicht gegenseitig durchdringen können und nicht durchsichtig sind. Ist die Papierebene die Projektionsebene, könnte ein Betrachter beispielsweise die in Abbildung 7.42 dargestellte Szene sehen.

Zur Lösung dieses Problems kann man so vorgehen: Wir schwenken eine horizontale Scan-line über die in die Betrachtungsebene projizierte zweidimensionale Szene. D.h. wir haben eine Menge von Polygonen in der Ebene wie im Falle des Polygonschnittpro-

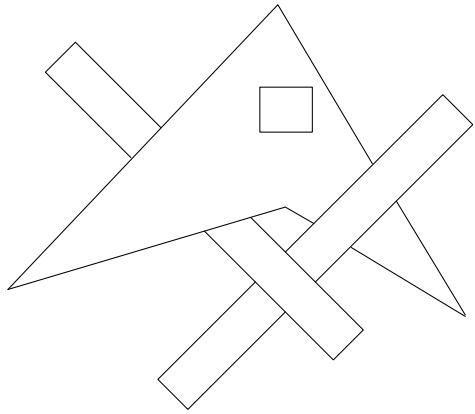


Abbildung 7.42

blems. Anders als beim Polygonschnittproblem merken wir uns jetzt aber zu jedem ein Polygon repräsentierenden, gerade aktiven Intervall dessen relative Distanz zum Betrachter. Für jede Position der Scan-line gilt: Eine Kante ist sichtbar genau dann, wenn sie ein Intervall begrenzt, das unter allen gerade aktiven Intervallen, die diese Kante enthalten, die geringste Distanz zum Betrachter hat. Anstelle von Aufspieß-Anfragen, die beim Polygonschnittproblem ausgeführt werden, um Inklusionen zu entdecken, müssen jetzt also Sichtbarkeitstests durchgeführt werden an allen Stellen, an denen sich die Sichtbarkeitsverhältnisse von Kanten ändern können. Das sind die Anfänge und Enden von Kanten und die Schnittpunkte zwischen je zwei Kanten. Verwendet man Segment-Bäume zur Speicherung der jeweils gerade aktiven Intervalle, kann man die an den Kanten des Skeletts stehenden Intervalle als nach Distanz zum Betrachter sortierte Listen organisieren. Dann ist das Einfügen und Entfernen von Intervallen in $O(\log^2 N)$ Schritten möglich. Ein Sichtbarkeitstest kann in $O(\log N)$ Schritten ausgeführt werden. Man durchläuft wie bei Aufspieß-Anfragen einen Suchpfad im Skelett des Segment-Baumes von der Wurzel zu dem Blatt, das der auf Sichtbarkeit zu prüfenden Kante entspricht, und inspiziert auf diesem Suchpfad jeweils nur ein Element der nach Distanz geordneten Intervall-Listen: Das Element mit der jeweils geringsten Distanz zum Betrachter. Am Ende weiß man dann, ob die auf Sichtbarkeit zu prüfende Kante ein Intervall begrenzt, das unter allen die Kante enthaltenden Intervallen, die gerade aktiv sind, die geringste Distanz zum Betrachter hat, also sichtbar ist, oder nicht.

Es ist zwar möglich, auch Intervall-Bäume zur Speicherung der jeweils gerade aktiven Intervalle zu nehmen; jedoch sind Sichtbarkeitstests dann nicht so einfach durchzuführen wie im Falle von Segment-Bäumen mit nach (relativer) Distanz zum Betracher geordneten Intervall-Listen. Für weitere Einzelheiten verweisen wir auf [140].

7.6 Anwendungen geometrischer Datenstrukturen

Segment-Bäume und Intervall-Bäume sind Strukturen zur Speicherung von eindimensionalen Intervallen; Prioritäts-Suchbäume dienen zur Speicherung von Punkten in der Ebene. Wir haben diese Strukturen im Abschnitt 7.4 als *halbdynamische* Skelettstrukturen eingeführt: Man kann Objekte, d.h. Intervalle oder Punkte, eines festen Universums einfügen und entfernen und kann darüberhinaus bestimmte geometrische Anfragen effizient beantworten. Alle drei Strukturen lassen sich zur Lösung des Rechteckschnittproblems nach dem Scan-line-Prinzip benutzen. Wir wollen in diesem Abschnitt einige weitere Beispiele für die vielfältigen Anwendungsmöglichkeiten dieser Strukturen angeben. Im Abschnitt 7.6.1 lösen wir einen sehr einfachen Spezialfall des Hidden-Line-Eliminationsproblems (HLE). Dieser Spezialfall ist dadurch charakterisiert, daß alle Flächen in der gegebenen Szene iso-orientiert und parallel zur Projektionsebene sind. Man erhält für diesen Spezialfall des HLE-Problems eine Lösung, deren Komplexität von der Größe der Eingabe und der Größe der Ausgabe, d.h. der Anzahl der sichtbaren Kanten, nicht aber von der Anzahl der Kantenschnitte in der Projektion abhängt.

Im Abschnitt 7.6.2 diskutieren wir ein Suchproblem für Punktmengen in der Ebene mit Fenstern fester Größe. Als Fenster erlauben wir ein beliebiges Rechteck, das in der Ebene verschoben werden kann. Wir zeigen, daß Varianten von Prioritäts-Suchbäumen eine zur Speicherung der Punkte geeignete Struktur sind, die folgende Operationen unterstützt: Das Einfügen und Entfernen von Punkten und das Aufzählen aller Punkte, die in das Fenster bei einer gegebenen Lage fallen.

7.6.1 Ein Spezialfall des HLE-Problems

Eine dreidimensionale Szene kann man sich gegeben denken durch eine Menge undurchsichtiger, sich gegenseitig nicht durchdringender, polygonal begrenzter ebener Flächen im Raum. Wir wollen eine solche dreidimensionale Szene auf eine zweidimensionale Betrachtungsebene projizieren und die in der Projektion sichtbaren Kanten berechnen. Dazu setzen wir die orthographische Projektion voraus, d.h. wir setzen parallele, etwa senkrecht von oben kommende Projektionsstrahlen (Licht) voraus. Dies ist eine durchaus übliche Annahme. Wir machen jedoch eine weitere, sehr spezielle und in der Praxis wohl nur selten realisierte Annahme: Alle Flächen sollen rechteckig, iso-orientiert und parallel zur Projektionsebene sein. Ein aus $z = \infty$ auf die x-y-Ebene schauender Betrachter könnte also zum Beispiel das in Abbildung 7.43 gezeigte Bild sehen, wenn die x-y-Projektionsebene die Papierebene ist.

In diesem Fall kann man die sichtbaren Kanten der als undurchsichtig vorausgesetzten Flächen wie folgt bestimmen [72]. Man baut die sichtbare Kontur der Flächen von vorn nach hinten auf: Begonnen wird mit der Fläche mit größtem z-Wert, da diese dem Betrachter am nächsten liegt. Von ihr sind alle Kanten sichtbar. Dann geht man die Flächen in der Reihenfolge wachsender Distanz zum Betrachter, also mit abnehmenden z-Werten, der Reihe nach durch. Jedesmal, wenn man dabei auf eine neue Fläche

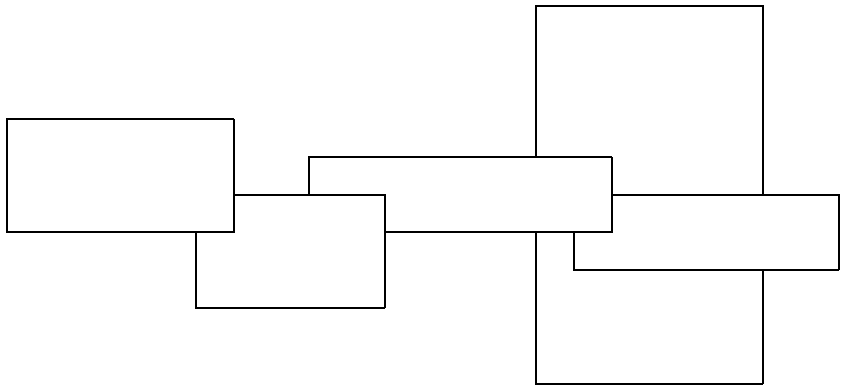


Abbildung 7.43

trifft, wird die Kontur des nunmehr sichtbaren Gebietes entsprechend aktualisiert, vgl. Abbildung 7.44.

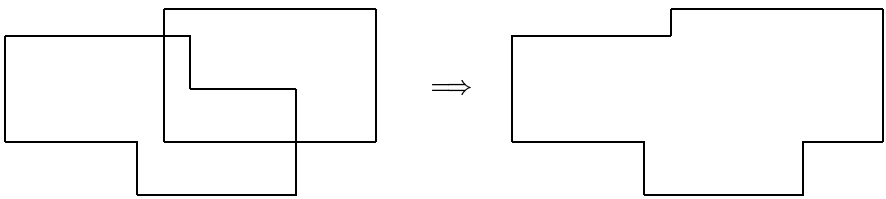


Abbildung 7.44

Es kommt also darauf an, die Menge der Rechtecke und ihre (sichtbare) Kontur so zu speichern, daß die oben angegebene Veränderung der Kontur effizient berechnet werden kann.

Wir verwenden dazu zwei Mengen E und F: E ist die Menge der Kanten der Kontur des bis zum jeweiligen z-Wert sichtbaren Gebietes; F ist eine Menge von Rechtecken, deren Vereinigung E als Kontur hat.

Man initialisiert E und F zunächst als leere Menge, sortiert die gegebene Menge R iso-orienterter und zur Projektionsebene paralleler Rechtecke nach abnehmenden z-Werten, also nach wachsender Distanz zum Betrachter, und geht dann wie folgt vor:

while *noch nicht alle Rechtecke betrachtet* **do**
  begin
    nimm nächstes Rechteck $r \in R$;

(1) *bestimme alle Schnitte zwischen Seiten von r und Kanten der Kontur*;
(1*a*) *für jede Kante e ∈ E, die von einer Seite von r geschnitten wird, berechne die außerhalb von r liegenden* {*sichtbaren*!} *Teile der neuen Kontur, füge sie in E ein und entferne e aus E*;
(1*b*) *für jede Kante e′ von r, die eine Kante der Kontur schneidet, berechne die außerhalb der Kontur liegenden Teile von e′, berichte diese Teile als sichtbar und füge sie in E ein*;
(2) *für jede Kante e′ von r, die keine Kante der Kontur schneidet, stelle* (*mit Hilfe von F*) *fest, ob sie ganz innerhalb von E liegt* (*also unsichtbar ist*) *oder nicht*;
if *e′ ist nicht innerhalb E*
then *berichte e′ als sichtbar und füge sie in E ein*;
(3) *bestimme alle Kanten von E, die ganz innerhalb r liegen und entferne sie aus E*;
(4) *füge r in F ein*
end {*while*}

Falls das nächste Rechteck *r* ganz innerhalb der aktuellen Kontur *E* liegt, bleibt *E* also unverändert, und es wird nichts berichtet. Falls das nächste Rechteck *r* das Gebiet mit Kontur *E* ganz einschließt, so wird *r* zur Kontur des neuen sichtbaren Gebietes. Die Kanten von *r* werden im Schritt (2) des Algorithmus als sichtbar berichtet und als Ergebnis von Schritt (2) und (3) wird die bisherige Kontur *E* durch die Kanten von *r* als neuer Kontur ersetzt.

Im allgemeinen wird das nächste Rechteck *r* einige Kanten der (alten) Kontur *E* schneiden, wie im in Abbildung 7.44 gezeigten Beispiel. In Abbildung 7.45 haben wir die Kanten mit den Nummern der Schritte markiert, in denen sie nach dem oben angegebenen Algorithmus betrachtet werden.

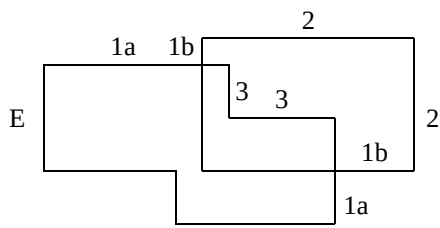


Abbildung 7.45

Die Frage, ob Kanten von *E* innerhalb von *r* liegen, wird gestellt, nachdem eventuelle Kantenschnitte zwischen *E* und *r* bereits behandelt wurden. Daher kann der Test, ob eine Kante von *E* innerhalb von *r* liegt, ersetzt werden durch einen Test, ob je ein Punkt einer Kante von *E* innerhalb *r* liegt. Aus demselben Grunde läßt sich auch die Frage,

ob eine Kante von r innerhalb oder außerhalb von E liegt, auf die entsprechende Frage für einen (eine Kante repräsentierenden) Punkt reduzieren.

Insgesamt folgt, daß es für eine Implementation des oben angegebenen Verfahrens genügt, folgende drei Teilprobleme zu lösen:

1. Ein *Segmentschnitt-Suchproblem*: Für eine gegebene Menge S horizontaler (vertikaler) Segmente und ein gegebenes vertikales (horizontales) Segment l, finde alle Segmente in S, die l schneidet.

2. Ein *zweidimensionales Aufspieß-Problem* (oder: eine zweidimensionale inverse Bereichsanfrage): Für eine gegebene Menge R von Rechtecken und einen gegebenen Punkt p, stelle fest, ob p in $\bigcup R$ liegt.

3. Eine *zweidimensionale Bereichsanfrage*: Für eine gegebene Menge P von Punkten und ein gegebenes Rechteck r, finde alle Punkte von P, die innerhalb r liegen.

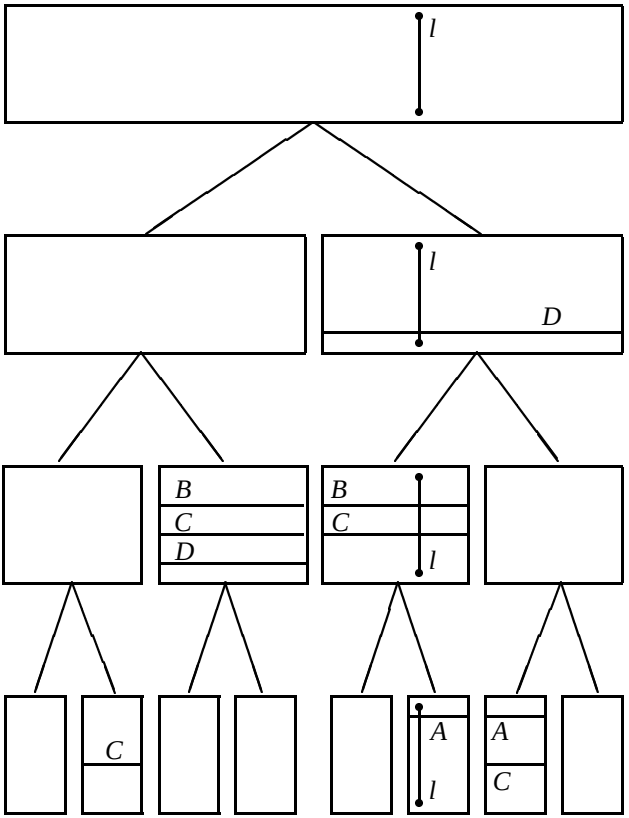
Die Teilprobleme 2 und 3 treten im Schritt (2) und (3) des Algorithmus auf, nachdem das Teilproblem 1 im Schritt (1) behandelt wurde. In jedem Fall werden dynamische Lösungen für die drei Teilprobleme benötigt, weil im Verlaufe des Verfahrens Objekte in die jeweiligen Mengen eingefügt oder aus ihnen entfernt werden.

Wir skizzieren mögliche Lösungen für die drei Teilprobleme:

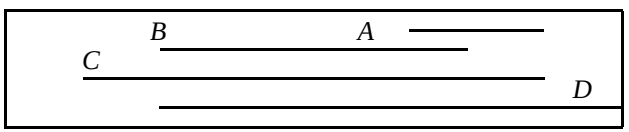
Zur Lösung des Segmentschnitt-Suchproblems für eine Menge horizontaler Segmente kann man *Segment-range-Bäume* verwenden. Das sind Segment-Bäume, deren Knotenlisten als Bereichs-Suchbäume organisiert sind und damit Bereichsanfragen unterstützen. Genauer: Man baut einen Segment-Baum als halbdynamische Skelettstruktur, die allen im Verlauf des Verfahrens angetroffenen horizontalen Segmenten Platz bietet. Die an den Knoten des Skeletts stehenden Listen von Intervallnamen werden als Bereichs-Suchbäume, d.h. z.B. als balancierte Blattsuchbäume mit doppelt verketteten Blättern organisiert, so daß Bereichsanfragen für vertikale Intervalle beantwortet werden können in einer Anzahl von Schritten, die proportional zum Logarithmus der Anzahl der Intervalle in der jeweiligen Liste und zur Anzahl der zu berichtenden Intervalle ist. In Abbildung 7.46 haben wir diese Struktur an Hand eines einfachen Beispiels veranschaulicht, indem wir die zweistufige, hierarchische Struktur in der Ebene ausgebreitet haben und an Stelle von Bereichs-Suchbäumen einfach vertikal angeordnete Intervall-Listen dargestellt haben.

Werden Segment-range-Bäume wie oben angegeben implementiert, so können die benötigten Operationen wie folgt ausgeführt werden:

Zum Einfügen eines neuen horizontalen Segments H bestimmt man die $\log N$ Knoten des Skeletts, in deren Knotenliste H eingefügt werden muß. Jede Knotenliste ist ein vertikal geordneter, balancierter Blattsuchbaum mit höchstens N Elementen. Daher kann H in eine einzelne Knotenliste in $\log N$ Schritten und insgesamt in $O(\log^2 N)$ Schritten in einen Segment-range-Baum eingefügt werden. Das Entfernen eines horizontalen Segments verläuft genau umgekehrt und kann ebenfalls in $O(\log^2 N)$ Schritten ausgeführt werden. Da ein Segment-Baum zur Speicherung von N horizontalen Segmenten in sämtlichen Knotenlisten höchstens insgesamt $N \log N$ Elemente hat, hat natürlich auch ein Segment-range-Baum einen Speicherbedarf der Größe $O(N \log N)$. Um für ein gegebenes vertikales Segment l alle horizontalen Segmente zu finden, die l schneiden, benutzt man den x-Wert von l als Suchschlüssel für eine Suche im Segment-Baum und



Segment-range-Baum zur Speicherung von S: Jeder Knoten enthält eine vertikal angeordnete Liste von Intervallen



Menge $S = \{A, B, C, D\}$ horizontaler Intervalle, vertikales Segment l.

Abbildung 7.46

berichtet für jeden Knoten auf dem Suchpfad nach x alle im Intervall l liegenden Segmente durch eine Bereichsanfrage im jeweiligen Bereichs-Suchbaum. Offensichtlich können auf diese Weise alle k horizontalen Segmente, die l schneiden, in $O(\log^2 N + k)$ Schritten bestimmt werden.

Zur Lösung des zweidimensionalen Aufspieß-Problems benutzen wir *Segment-Segment-Bäume*: Das ist wiederum eine hierarchische Struktur, die aus einem Segment-Baum besteht, dessen Knotenlisten ebenfalls als Segment-Bäume organisiert sind. Genauer: Die horizontalen Projektionen der Rechtecke (auf die x-Achse) werden in einem Segment-Baum gespeichert. Enthält die Liste der Projektionen an einem Knoten dieses Segment-Baumes die (Namen der) Rechtecke $R_1, \ldots, R_t$, so werden die vertikalen Projektionen dieser Rechtecke (auf die y-Achse) ebenfalls in einem Segment-Baum gespeichert, der diesem Knoten zugeordnet ist. Dann kann man durch eine Suche im Segment-Baum für die horizontalen Rechteckprojektionen nach dem x-Wert eines gegebenen Punktes $p \in (x_0, y_0)$ die höchstens $\log N$ Knoten mit daranhängenden Segment-Bäumen bestimmen, die die vertikalen Projektionen sämtlicher Rechtecke enthalten, deren horizontale Projektion von x_0 aufgespießt wird. Unter diesen findet man in $O(\log N + k_i)$ Schritten je Segment-Baum S_i alle in S_i enthaltenen Rechtecke, deren vertikale Projektion von y_0 aufgespießt wird.

Insgesamt lassen sich also alle k Rechtecke, die p aufspießt, in Zeit $O(\log^2 N + k)$ finden. Es ist nicht nötig und aus Speicherplatzgründen auch nicht sinnvoll, die Segment-Bäume zur Speicherung der vertikalen Projektionen über dem Raster aller möglichen y-Werte von Rechtecken zu bauen. Vielmehr genügt es, zu jedem Knoten im Segment-Baum für die horizontalen Projektionen vorab alle die Rechtecke zu bestimmen, die jemals in die Knotenliste dieses Knotens aufgenommen werden müssen; dann genügt es, den Segment-Baum für die vertikalen Projektionen, der an diesem Knoten hängt, über dem von den vorab bestimmten Rechtecken induzierten Raster zu bauen. Dann bleibt der gesamte Speicherbedarf des Segment-Segment-Baumes in der Größenordnung $O(N \log^2 N)$ und der Zeitbedarf zum Aufbau des leeren Skeletts bei $O(N \log N)$.

Das letzte Teilproblem, nämlich das Beantworten zweidimensionaler Bereichsanfragen für eine durch Einfüge- und Entferne-Operationen veränderliche Menge von Punkten, ist auf vielfältige Weise lösbar. Es gehört zu den am gründlichsten untersuchten zweidimensionalen Suchproblemen überhaupt. Entsprechend vielfältig ist das Spektrum der zur Speicherung der Punkte geeigneten Datenstrukturen. Wir skizzieren hier kurz eine mögliche Lösung mit Hilfe von *Range-range-Bäumen*: Ein Range-range-Baum für eine dynamisch veränderliche Menge von Punkten über einem festen Universum von N möglichen Punkten hat große Ähnlichkeit mit einem Segment-Segment-Baum: Man baut zunächst einen halbdynamischen Bereichs-Suchbaum, der eindimensionale Bereichsanfragen, etwa für x-Bereiche unterstützt. Das Skelett eines halbdynamischen Bereichs-Suchbaums unterscheidet sich nicht wesentlich vom Skelett eines Segment-Baumes. Das Universum der möglichen x-Werte wird in elementare Fragmente eingeteilt und über dieser Menge wird ein vollständiger Binärbaum gebaut. Jeder (innere) Knoten repräsentiert dann ein Intervall auf der x-Achse, das genau aus der Folge der elementaren Fragmente besteht, die durch die Blätter des Teilbaumes des Knotens repräsentiert werden. Jeder Knoten enthält eine Liste von Punkten: In die Liste des Knotens p kommen genau die Punkte, die in das von p repräsentierte Intervall fallen. Man sieht leicht, daß jeder Punkt in höchstens $\log N$ Knotenlisten vorkommen kann. Die Liste der Wurzel enthält alle aktuell vorhandenen Punkte und die Blätter enthalten

jeweils höchstens einen Punkt. Nehmen wir an, es sollen alle Punkte bestimmt werden, die in einen gegebenen Bereich fallen. Dabei nehmen wir ohne Einschränkung an, daß der Bereich aus einer zusammenhängenden Folge von Elementarfragmenten besteht. Dann kann man in $\log N$ Schritten die Knoten finden, die den gegebenen Bereich im Skelett repräsentieren, d.h. die am nächsten bei der Wurzel liegen und ein Intervall repräsentieren, das ganz im gegebenen Bereich liegt. Die Punkte in den zu diesen Knoten gehörenden Punktlisten sind genau die gesuchten. Abbildung 7.47 zeigt ein Beispiel einer Menge von neun Punkten $\{A,\ldots,I\}$ über einem Universum von 16 möglichen x-Werten.

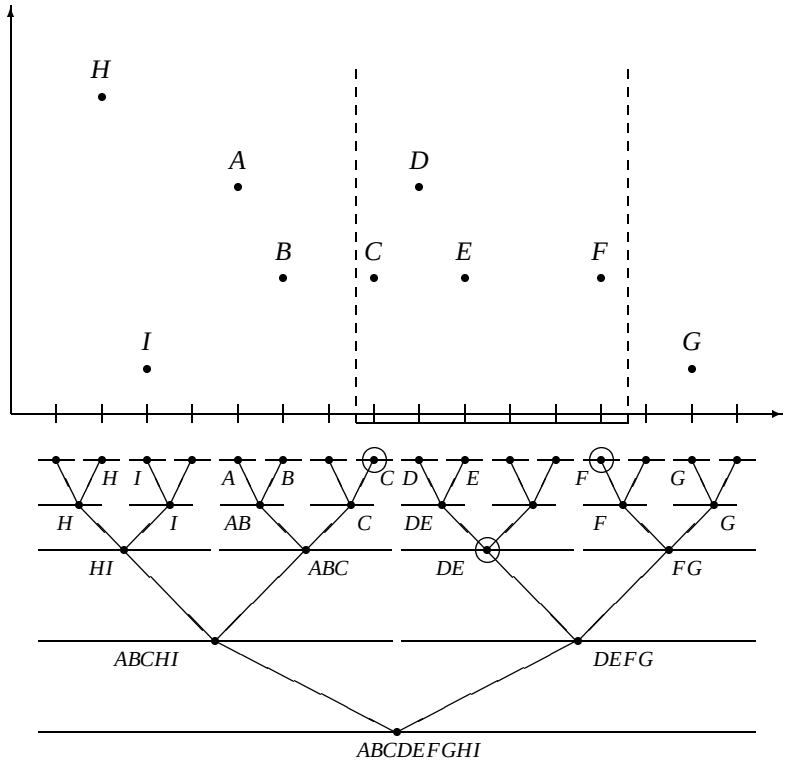


Abbildung 7.47

Der gegebene Bereich $[x_l, x_r]$ wird im Baum von Abbildung 7.47 durch die drei eingekreisten Knoten repräsentiert. Dort stehen genau die Punkte, deren x-Wert in den Bereich $[x_l, x_r]$ fällt. Zum Einfügen eines Punktes P sucht man im Baum nach P und fügt P in die Listen aller Knoten auf dem Suchpfad ein. Zum Entfernen eines Punktes P geht man umgekehrt vor, hat aber natürlich (wie bei Segment-Bäumen) das Problem, die Stellen innerhalb der Punktlisten zu finden, an denen P auftritt. Dieses Problem läßt

sich, wie bei Segment-Bäumen, mit Hilfe eines (globalen) Wörterbuches lösen. Insgesamt erhält man so eine Struktur mit folgenden Charakteristika:

Das Einfügen und Entfernen eines Punktes kann in $O(\log N)$ Schritten ausgeführt werden; für einen gegebenen eindimensionalen Bereich kann man alle k in den Bereich fallenden Punkte in Zeit $O(\log N + k)$ finden; der Platzbedarf ist von der Ordnung $O(N \log N)$.

Natürlich haben wir diese Struktur nicht entwickelt, nur um damit eindimensionale Bereichsanfragen beantworten zu können. (Dafür hätten wir auch balancierte Blattsuchbäume als volldynamische Struktur nehmen können.) Die soeben vorgestellten, analog zu Segment-Bäumen gebildeten halbdynamischen Bereichs-Suchbäume sind vielmehr geeignete Bausteine für hierarchisch aufgebaute Strukturen. Man kann auf ihrer Basis insbesondere Range-range-Bäume bauen, die zweidimensionale Bereichsanfragen unterstützen: Man organisiert die Punktlisten eines halbdynamischen Bereichs-Suchbaums, der Bereichsanfragen für x-Bereiche unterstützt, als halb- oder volldynamische Bereichs-Suchbäume, die Bereichsanfragen für y-Bereiche unterstützen.

In einer solchen Struktur lassen sich Punkte in $O(\log^2 N)$ Schritten einfügen und entfernen und alle k Punkte eines gegebenen zweidimensionalen Bereichs in $O(\log^2 N + k)$ Schritten aufzählen. Der Platzbedarf eines Range-range-Baums ist $O(N \log N)$, wenn die Bereichs-Suchbäume zur Unterstützung von Bereichsanfragen für y-Bereiche als volldynamische Bereichs-Suchbäume implementiert wurden.

Fassen wir noch einmal kurz zusammen, wie wir den zu Eingang dieses Abschnitts angegebenen Spezialfall des HLE-Problems lösen können: Wir gehen die Menge der gegebenen Rechtecke der Reihe nach mit wachsender Distanz vom Betrachter durch. Dabei merken wir uns die Kontur des jeweils sichtbaren Bereichs in einer Menge E horizontaler und vertikaler Liniensegmente, d.h. E wird als Paar von Segment-range-Bäumen, je ein Baum für die horizontalen und ein Baum für die vertikalen Kanten, repräsentiert. Weiter wird jede Kante von E durch einen Punkt repräsentiert und die Menge dieser Punkte in einem Range-range-Baum gespeichert. Schließlich wird eine Menge F von Rechtecken, deren Vereinigung die Kontur E hat, als Segment-Segment-Baum gespeichert. Wird dann ein neues Rechteck r angetroffen, so verändert man diese Strukturen wie im Algorithmus oben angegeben und gibt gegebenenfalls sichtbare Teile von Kanten von r aus. Es ist nicht schwer zu sehen, daß der insgesamt erforderliche Zeitaufwand von der Ordnung $O(N \log^2 N + q \cdot \log^2 N)$ ist, wenn q die Anzahl der sichtbaren Kanten und N die Anzahl der ursprünglich gegebenen Rechtecke ist.

7.6.2 Dynamische Bereichssuche mit einem festen Fenster

In diesem Abschnitt behandeln wir das Problem, für eine gegebene Menge von Punkten in der Ebene und einen gegebenen Bereich alle Punkte zu bestimmen, die in den Bereich fallen. Dieses Problem hat viele Varianten: Wir können annehmen, daß die Punktmenge fest, aber die Bereiche variabel sind. Die Bereiche können rechteckig, durch ein (konvexes) Polygon begrenzt oder kreisförmig sein. Man kann aber auch einen Bereich fester Größe und Gestalt annehmen, der wie ein Fenster über die Punktmenge verschoben werden kann. Man denke etwa an einen Bildschirm als Fenster, mit dem man auf eine Menge von Punkten blickt. Wir interessieren uns für diese Variante des Problems und

nehmen aber zusätzlich an, daß die Menge der Punkte nicht ein für allemal fest gegeben ist, sondern durch Einfügen und Entfernen von Punkten dynamisch verändert werden kann.

Wir setzen ein kartesisches x-y-Koordinatensystem in der Ebene voraus und bezeichnen die x- und y-Koordinaten eines Punktes a mit a_x und a_y, also $a = (a_x, a_y)$. Für zwei Punkte a und b sei $a + b = (a_x + b_x, a_y + b_y)$ und für eine Menge A von Punkten und einen Punkt q sei

$$A_q = A + q = \{(a_x + q_x, a_y + q_y) \mid a \in A\}.$$

Jetzt können wir das in diesem Abschnitt behandelte Problem präziser wie folgt formulieren: Sei P eine Menge von Punkten und sei W ein festes Fenster (z.B. ein Rechteck, Dreieck, konvexes Polygon, Kreis); für einen gegebenen Punkt q sollen folgende Operationen ausgeführt werden:

Einfügen(P,q): Fügt den Punkt q in die Menge P ein.

Entfernen(P,q): Entfernt den Punkt q aus der Menge P.

Window$_W(P,q)$: Liefert alle Punkte in $P \cap W_q$.

Dann nennen wir eine Repräsentation von P zusammen mit Algorithmen zum Ausführen der Operationen *Einfügen*, *Entfernen*, *Window*$_W$ eine Lösung des dynamischen Bereichssuchproblems mit Fenster W.

Wir behandeln den Fall, daß W ein Rechteck ist, das durch seinen linken, rechten, unteren und oberen Rand gegeben ist, also $W = (x_l, x_r, y_b, y_t)$. Das dynamische Bereichssuchproblem für ein rechteckiges Fenster W nennen wir auch kurz *DRW-Problem*. Wir zeigen, wie man das DRW-Problem mit (volldynamischen) Prioritäts-Suchbäumen löst.

Zur Lösung des DRW-Problems zerschneiden wir die euklidische Ebene in Gedanken in horizontale Streifen der Höhe $Y = \text{Höhe}(W) = y_t - y_b$. Wir nennen

$$s_i = \{p \mid iY \leq p_y < (i+1)Y\}$$

den i-ten Streifen. Wenn $p \in s_i$ ist, heißt i die Streifennummer von p; sie wird mit $s(p)$ bezeichnet. Es ist klar, daß man für jeden Punkt p die Streifennummer $s(p)$ in konstanter Zeit berechnen kann.

Die Zerlegung der Ebene in Streifen der Höhe Y hat folgende wichtige Konsequenzen:

1. Für jede durch einen Punkt q gegebene Verschiebung W_q des Rechtecks W gilt: W_q schneidet höchstens zwei Streifen.

2. Für jeden Streifen s und jede durch einen Punkt q gegebene Verschiebung gilt entweder

   (a) $W_q \cap s = \emptyset$ oder

   (b) $W_q \cap s \neq \emptyset$ und ($W_q \cap s$ ist S-gegründet in s oder $W_q \cap s$ ist N-gegründet in s).

Hier benutzen wir die bereits im Abschnitt 7.4.4 eingeführten Begriffe *S*-gegründet (für: Süd-gegründet) und *N*-gegründet (für: Nord-gegründet). Für einen Bereich *R* (ein Fenster) und einen Streifen *s* heißt *R S-gegründet* (bzw. *N-gegründet*) in *s*, wenn $R \cap s$ mit der orthogonalen Projektion von *R* auf die untere, also südliche (bzw. auf die obere, also nördliche) Begenzung von *s* zusammenfällt. In dem in Abbildung 7.48 gezeigten Beispiel ist W_{q_2} *S*-gegründet in s_{i+1} und *N*-gegründet in s_i. Falls eine Verschiebung W_q von *W* sowohl *S*- als auch *N*-gegründet in einem Streifen *s* ist, muß offenbar $W_q \cap s = W_q$ sein. Abbildung 7.48 zeigt auch dafür ein Beispiel.

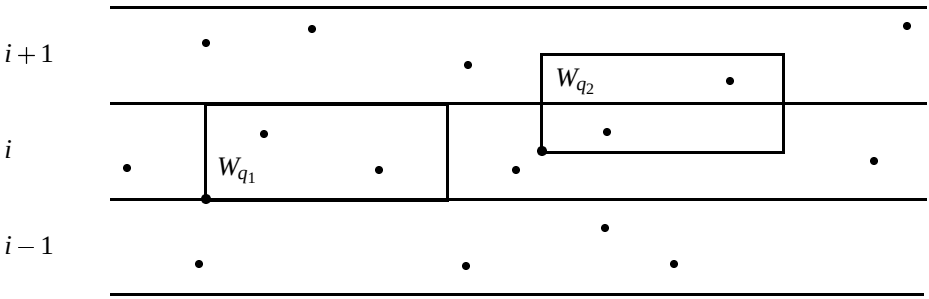


Abbildung 7.48

Die Idee zur Lösung des DRW-Problems ist nun, jedem Streifen *s* ein Paar von Prioritäts-Suchbäumen zuzuordnen, die die Punkte in *s* speichern. Ein Prioritäts-Suchbaum unterstützt *S*-gegründete Anfragen in *s* und der zweite Prioritäts-Suchbaum *N*-gegründete Anfragen in *s*. Natürlich können wir nicht annehmen, daß das Universum der in *s* fallenden Punkte im vorhinein bekannt und fest ist. Wir müssen also volldynamische Prioritäts-Suchbäume verwenden. Es kommen dafür nicht die in Abschnitt 7.4.4 als halbdynamische Skelettstruktur implementierten Prioritäts-Suchbäume, wohl aber die dort ebenfalls angegebene, analog zu natürlichen Suchbäumen entwickelte volldynamische Struktur in Frage. Sie erlaubt das Einfügen und Entfernen von Punkten im Mittel in logarithmischer Zeit und auch das Berichten aller *k* Punkte in einem *S*- (bzw. *N*-) gegründeten Bereich im Mittel in Zeit $O(\log N + k)$. Es ist bekannt, vgl. [120], daß Prioritäts-Suchbäume auch als balancierte Bäume gebaut werden können mit dem Ergebnis, daß die Operationen *Einfügen*, *Entfernen* und $Window_W$ auch im schlechtesten Fall jeweils in $O(\log N)$ bzw. $O(\log N + k)$ Schritten ausführbar sind. Weil Prioritäts-Suchbäume (in jedem Fall) Blattsuchbäume für die *x*-Werte von Punkten in der Ebene sind, unterstützen sie Bereichsanfragen für *x*-Intervalle; weil Prioritäts-Suchbäume Heaps bzgl. der *y*-Werte von Punkten sind, erlauben sie es, alle Punkte zu berichten, deren *y*-Wert unterhalb eines gegebenen Schwellenwertes liegt. Beide Eigenschaften zusammen liefern gerade das, was wir brauchen: Um *S*-gegründete Anfragen beantworten zu können, speichern wir die Punkte eines Streifens *s* in einem *s* zugeordneten Prioritäts-Suchbaum derart, daß die Punkte mit kleinerem *y*-Wert näher bei der Wurzel stehen, d.h. die Prioritätsordnung ist die *y*-Ordnung. Um *N*-gegründete Anfragen beant-

worten zu können, speichern wir die Punkte mit größerem y-Wert näher bei der Wurzel, d.h. die Prioritätsordnung ist die negative y-Ordnung.

Ordnet man also jedem Streifen s ein Paar von volldynamischen Prioritäts-Suchbäumen zu, so kann man Punkte (im Streifen s) einfügen und entfernen, indem man diese Operationen in beiden s zugeordneten Prioritäts-Suchbäumen ausführt. Zur Beantwortung von S- bzw. N-gegründeten Bereichsanfragen konsultiert man jeweils nur einen Prioritäts-Suchbaum. Abbildung 7.49 veranschaulicht dies noch einmal.

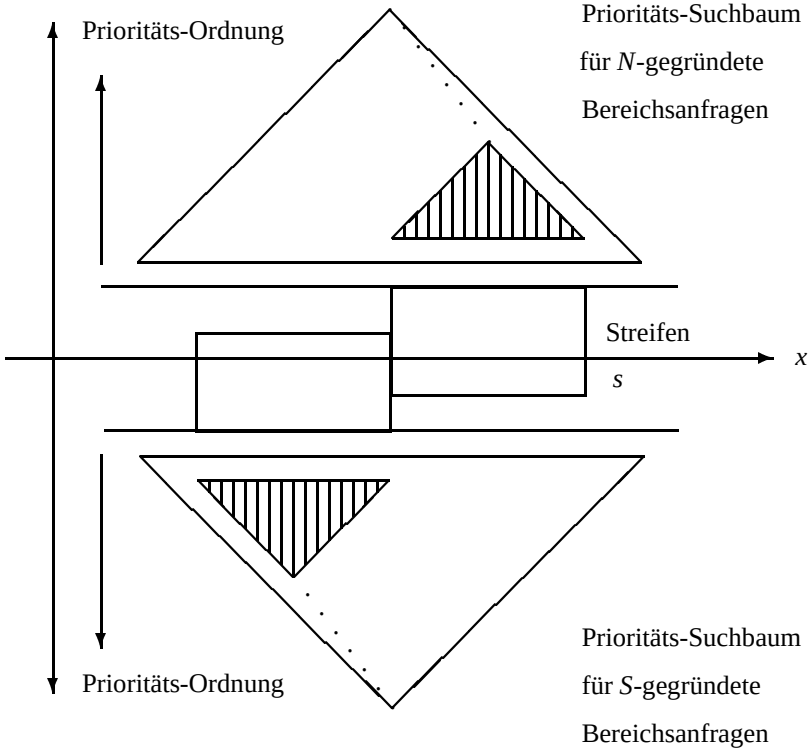


Abbildung 7.49

Um linearen Speicherplatz und einen Zeitbedarf von $O(\log N)$ bzw. $O(\log N + k)$ im Mittel bzw. im schlechtesten Fall in der Anzahl N der Punkte in P und der Anzahl k der bei einer $Window_W$-Operation zu berichtenden Punkte zu erhalten, darf man allerdings leere Streifen nicht explizit repräsentieren. Deshalb speichert man die Streifennummern genau der nichtleeren Streifen in einem balancierten Suchbaum T_S: Jeder in T_S gespeicherten Streifennummer ordnen wir ein Paar von Prioritäts-Suchbäumen zu, die genau die Punkte, die im Streifen mit dieser Streifennummer liegen, enthalten. Damit kön-

nen die zur Lösung des DRW-Problems benötigten Operationen wie folgt ausgeführt werden.

Einfügen(P,q): Bestimme die Streifennummer $s(q)$ von q; suche in T_S nach $s(q)$; wenn $s(q)$ in T_S bereits vorkommt, füge q in die beiden $s(q)$ zugeordneten Prioritäts-Suchbäume ein; andernfalls, d.h. wenn $s(q)$ nicht in T_S vorkommt, füge $s(q)$ in T_S ein, schaffe ein neues Paar von Prioritäts-Suchbäumen, die beide genau q speichern, und ordne dies Paar $s(q)$ zu.

Entfernen(P,q): Analog.

Window$_W(P,q)$: Bestimme die Nummern der höchstens zwei nichtleeren Streifen, die W_q schneidet; suche in T_S nach diesen Nummern und benutze die den Nummern zugeordneten Prioritäts-Suchbäume, um die Punkte in den S- bzw. N-gegründeten Teilen von W_q zu berichten.

Mit demselben Zeitbedarf wie das DRW-Problem kann man auch das dynamische Bereichssuchproblem mit einem festen, dreieckigen Fenster lösen [88]. Diese Lösung kann leicht auf den Fall ausgedehnt werden, daß das Fenster ein durch ein beliebiges, einfach geschlossenes Polygon begrenzter Bereich ist: Man zerlegt den Bereich in eine (feste, endliche) Anzahl von Dreiecken. Damit kann man die dynamische Bereichssuche mit polygonalem Fenster reduzieren auf eine feste Anzahl von dynamischen Bereichssuchen mit dreieckigem Fenster.

7.7 Distanzprobleme und ihre Lösung

Bei keinem der bisher betrachteten geometrischen Probleme hat die *Distanz* von Objekten eine Rolle gespielt. In diesem Abschnitt werden wir einige wichtige Probleme und Lösungen näher betrachten, bei denen die Distanz ein entscheidendes Kriterium ist. Wir beschränken uns auf die euklidische Ebene, also den $\mathbb{R}^2$ mit der (üblichen) Distanzfunktion $d(p_1,p_2)$, wobei $p_1=(x_1,y_1)$ und $p_2=(x_2,y_2)$ Punkte der Ebene sind:

$$d(p_1,p_2) := \sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$$

Anders ausgedrückt: Die Distanz zweier Punkte ist die Länge der geradlinigen Verbindung zwischen diesen Punkten. Man kann sich leicht davon überzeugen, daß dies tatsächlich eine Distanzfunktion ist, indem man die drei charakterisierenden Bedingungen überprüft:

(1) Für alle $p_1,p_2 \in \mathbb{R}^2$ ist $d(p_1,p_2)=0$ genau dann, wenn $p_1=p_2$.

(2) Für alle $p_1,p_2 \in \mathbb{R}^2$ ist $d(p_1,p_2)=d(p_2,p_1)$ (Symmetrie).

(3) Für alle $p_1,p_2,p_3 \in \mathbb{R}^2$ ist $d(p_1,p_2)+d(p_2,p_3) \geq d(p_1,p_3)$ (Dreiecksungleichung).

Sehen wir uns nun einige Distanzprobleme näher an.

7.7.1 Distanzprobleme

Wir wollen im folgenden einige der bestuntersuchten Distanzprobleme betrachten; andere findet man bei [149], [105] und [122]. Für jedes Problem geben wir einen naiven Lösungsalgorithmus sowie eine untere Schranke für die Zeitkomplexität der Lösung an.

Problem: *Dichtestes Punktepaar* (closest pair)
gegeben: Eine Menge P von N Punkten in der Ebene.
gesucht: Ein Paar p_1, p_2 von Punkten aus P mit minimaler Distanz.

Dieses Problem wird als eines der fundamentalen Probleme der algorithmischen Geometrie angesehen (vgl. [149]), weil es so einfach formuliert werden kann, zu einer effizienten Lösung aber bereits wichtige Prinzipien und Erkenntnisse erforderlich sind. Außerdem hat es auch allerlei praktische Anwendungen. Sind etwa die Punkte (Projektionen der) Flugzeuge in der Nähe eines Flugplatzes, so sind die am nächsten benachbarten Punkte die Flugzeuge mit der größten Kollisionsgefahr (allerdings bewegen sich in diesem Fall die Punkte [vgl. [142]).

Ein naives Verfahren, das dichteste Punktepaar zu bestimmen, besteht offenbar darin, für jedes Punktepaar die Distanz zu berechnen und dann das Minimum der Distanzen ausfindig zu machen. Da es bei N Punkten $N(N-1)/2$ Punktepaare gibt, kostet dieses Verfahren $\Theta(N^2)$ Schritte.

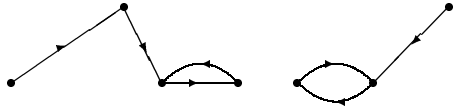
Die entscheidende Frage ist jetzt, ob man die geometrische Information über die Lage der Punkte ausnutzen kann, um das Problem effizienter zu lösen. Im eindimensionalen Fall ist das ganz leicht. Für eine Menge eindimensionaler Punkte $p_1 = (x_1), p_2 = (x_2), \ldots, p_N = (x_N)$ genügt es ja, die Punkte nach ihrem Koordinatenwert zu sortieren und sie dann in sortierter Reihenfolge zu betrachten. Die am dichtesten beieinanderliegenden Punkte sind offenbar Nachbarn in der Sortierreihenfolge. Damit ist das Problem im eindimensionalen Fall mit $O(N \log N)$ Rechenschritten lösbar.

Das ist gleichzeitig auch ein asymptotisch schnellstes Verfahren, wie folgende Überlegung zeigt. Das Problem, für eine gegebene Folge von Zahlen festzustellen, ob eine Zahl mehrmals in der Folge auftritt (element uniqueness), benötigt zur Lösung $\Omega(N \log N)$ Schritte für N Zahlen (vgl. [149] oder [105]). Dieses Problem läßt sich lösen, indem man nach dem dichtesten Zahlenpaar fragt. Ist die zugehörige Distanz 0, so gibt es eine Zahl, die mehr als einmal auftritt, sonst nicht. Folglich muß das Problem, das dichteste Zahlenpaar zu finden, ebenfalls mindestens $\Omega(N \log N)$ Schritte benötigen. Für Punkte in der Ebene (statt Zahlen, also eindimensionale Punkte) gilt diese untere Schranke erst recht.

Betrachten wir zunächst noch einige Distanzprobleme, bevor wir der Frage nach einer bestmöglichen Lösung nachgehen.

Problem: *Alle nächsten Nachbarn* (all nearest neighbors)
gegeben: Eine Menge P von N Punkten in der Ebene.
gesucht: Für jeden Punkt $p_1 \in P$ ein nächster Nachbar $p_2 \in P$, d.h. ein Punkt $p_2 \neq p_1$ mit $d(p_1, p_2) = \min_{p \in P - \{p_1\}} \{d(p_1, p)\}$.

Die Antwort für dieses Problem besteht also aus N Punktepaaren. Man beachte, daß die Relation "nächster Nachbar" nicht symmetrisch ist: Wenn p_2 nächster Nachbar von p_1 ist, so muß noch nicht p_1 nächster Nachbar von p_2 sein. Das folgende Bild zeigt eine Menge von Punkten und die Relation "nächster Nachbar": "p_2 ist nächster Nachbar von p_1" wird dargestellt durch einen Pfeil $p_1 \longrightarrow p_2$.



Dieses Problem läßt sich auf naive Weise lösen, indem man für jeden Punkt $p_1 \in P$ die Distanz zu allen anderen Punkten in P berechnet und einen Punkt p_2 mit minimaler Distanz auswählt; p_2 ist ein nächster Nachbar von p_1. Dieses Verfahren benötigt $\Theta(N^2)$ Schritte für eine Menge von N Punkten.

Man stellt leicht fest, daß das Problem für eindimensionale Punkte (wie auch schon das "closest pair"-Problem) wegen der Existenz einer totalen Ordnung auf den Punkten effizienter lösbar ist. So genügt es hier, die Punkte zu sortieren und anschließend für jeden Punkt seine beiden Nachbarn in der Sortierreihenfolge zu betrachten, denn nur sie kommen als nächste Nachbarn in Frage. Also kann auch dieses Problem für N eindimensionale Punkte mit $O(N \log N)$ Rechenschritten gelöst werden.

Das Problem, ein "dichtestes Punktepaar" zu finden, kann man lösen, indem man zuerst alle nächsten Nachbarn bestimmt, und dann ein Paar mit minimaler Distanz auswählt. Deshalb ist eine untere Schranke für die Laufzeit des "dichtestes Punktepaar"-Problems auch eine untere Schranke für das "alle nächsten Nachbarn"-Problem. Damit ist klar, daß dieses Problem mindestens $\Omega(N \log N)$ Schritte für eine Menge von N Punkten benötigt.

Betrachten wir nun das Problem, zu einer gegebenen Punktmenge ein kürzestes verbindendes Netzwerk zu finden. Die verschiedenen Versionen solcher Netzwerke, je nach Anforderungen an die Lösung, definieren kürzeste Verbindungen (etwa bei höchstintegrierten Schaltkreisen) oder einfach Ähnlichkeitsmaße für Punktmengen (etwa im Bereich der Mustererkennung). Wir interessieren uns dafür, einen minimalen spannenden Baum zu einer gegebenen Punktmenge zu finden.

Problem: *Minimaler spannender Baum* (minimum spanning tree)
gegeben: Eine Menge P von N Punkten in der Ebene.
gesucht: Ein minimaler spannender Baum für P, d.h. ein Baum, dessen Knoten gerade die Punkte aus P sind, dessen Kanten Verbindungen zwischen den Punkten sind, und der unter allen solchen Bäumen minimale Länge hat. Die Länge eines Baumes ist dabei die Summe der Längen seiner Kanten; die Länge einer Kante ist die (euklidische) Distanz der beiden Endpunkte.

Die Abbildung 7.50 zeigt eine Menge von sieben Punkten und einen minimalen spannenden Baum für diese Punktmenge.

Eine naive Lösung des Problems könnte darin bestehen, alle möglichen spannenden Bäume — das sind Bäume, deren Knoten gerade die Punkte aus P sind — zu berechnen, und einen mit minimaler Länge auszuwählen. Dazu müssen jedenfalls Kanten für alle Punktepaare betrachtet werden — das sind bereits $\Theta(N^2)$ Kanten. Jedes so operierende Lösungsverfahren muß also mindestens $\Theta(N^2)$ Schritte zur Lösung im schlimmsten Fall benötigen; womöglich benötigt es beträchtlich mehr.

Im eindimensionalen Fall läßt sich das Problem wieder ganz leicht durch Sortieren mit anschließendem Verbinden aller in der Sortierreihenfolge benachbarten Punkte lösen — also in $O(N \log N)$ Schritten für N Punkte.

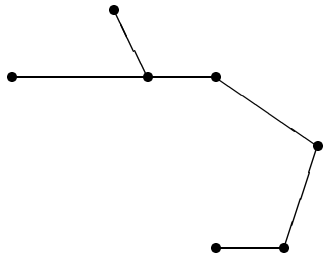


Abbildung 7.50

Es ist leicht einzusehen, daß jeder minimale spannende Baum für Punktmenge P eine verbindende Kante zwischen zwei dichtesten Punkten in P enthält. Betrachten wir zum Beweis einen spannenden Baum B, der für kein dichtestes Punktepaar eine verbindende Kante enthält. Nun verändern wir diesen Baum, indem wir eine solche Kante hinzufügen; das Resultat B' ist kein Baum mehr, weil es jetzt einen Zyklus gibt, wie die Abbildung 7.51 zeigt.

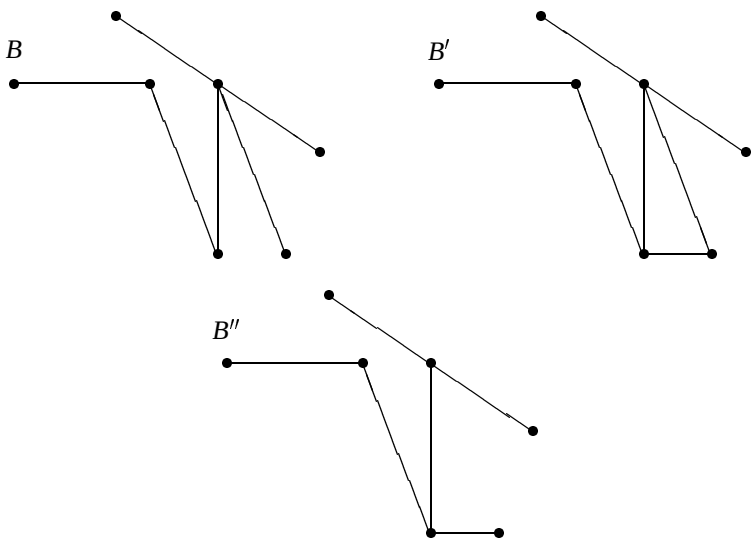


Abbildung 7.51

Aus B' machen wir durch Entfernen einer Kante des Zyklus (etwa der längsten Kante) wieder einen Baum B''. Dann ist klar, daß die Länge von B'' geringer ist als die von B, und damit kann B kein minimaler spannender Baum sein.

Jetzt läßt sich das "dichteste Paar"-Problem lösen, indem man zunächst einen minimalen spannenden Baum berechnet, und dann nur noch unter allen $N-1$ durch eine Kante verbundenen Punktepaaren das dichteste ausfindig macht. Deshalb benötigt die Berechnung eines minimalen spannenden Baumes im schlimmsten Fall mindestens soviel Zeit wie das Finden eines dichtesten Paares, nämlich $\Omega(N \log N)$.

Wieder stellt sich die Frage, ob man die Lage der Punkte in der Ebene nutzen kann, um einen schnellen — vielleicht sogar optimalen — Algorithmus zur Berechnung eines minimalen spannenden Baumes zu finden.

Die bisher genannten sind Probleme, bei denen man für eine gegebene Punktemenge einmal eine Frage beantworten will. Im Gegensatz dazu geht es bei einer großen Klasse von Problemen darum, wiederholt auf einer Grundmenge von Elementen gewisse Operationen auszuführen, wie etwa beim Speichern und Wiederfinden von Informationen. In solchen Fällen ist es oft nützlich, die Grundmenge, unter Umständen mit einigem Rechenaufwand, so vorzubehandeln (preprocessing), daß nachfolgende Anfragen schnell ausgeführt werden können. Stellen wir uns etwa vor, ein Kunde im Reisebüro sucht nach einem Urlaubsangebot eines gewissen Typs (Badeurlaub), zu einer grob festgelegten Zeit. Er hat Idealvorstellungen in vielerlei Hinsicht (Ort, Dauer, Preis, Verpflegung, Lage des Hotels, etc.), die er insgesamt so gut es eben geht realisieren möchte. Das Reisebüro wird versuchen, aus der Grundmenge aller verfügbaren Urlaubsreisen eine möglichst passende herauszusuchen (best match). Faßt man die Attribute einer Urlaubsreise als Koordinaten in einem mehrdimensionalen Koordinatensystem auf und bringt man die Gewichtung der Attribute in einer Distanzfunktion zum Ausdruck, so sucht unser Urlaubswilliger in der Menge der angebotenen Urlaubsreisen vielleicht gerade nach einer Reise mit geringster Distanz zu seiner Idealvorstellung. Für den Fall von Punkten in der Ebene läßt sich das "best match"-Problem wie folgt formulieren.

Problem: *Suche nächsten Nachbarn* (nearest neighbor search, best match)
gegeben: Eine Menge P von N Punkten in der Ebene.
gesucht: Eine Datenstruktur und Algorithmen, die

1. P in der durch die Datenstruktur vorgeschriebenen Form speichern (preprocessing),
2. zu einem gegebenen, neuen Punkt q (Anfragepunkt, query point) einen Punkt aus P finden, der nächster Nachbar von q ist.

Ganz ohne Vorbehandlung läßt sich eine Anfrage nach einem nächsten Nachbarn von q in Zeit $\Theta(N)$ beantworten, indem die Distanz von q zu jedem Punkt in P berechnet und das Minimum ausgesucht wird.

Im eindimensionalen Fall kann man wieder durch Sortieren von P, also mit Vorbehandlungsaufwand $O(N \log N)$, eine schnelle Beantwortung dieser Anfrage erreichen, nämlich mittels binärer Suche. Endet die binäre Suche erfolgreich, so hat man genau den gesuchten Punkt gefunden; andernfalls ist ein nächster Nachbar einer der (höchstens zwei) Nachbarn der Stelle, an der die Suche endet. Wegen der Optimalität der binären Suche ist auch diese Nachbarschaftssuche optimal: man kann sie zur Suche nach einem Punkt verwenden. Damit ist $\Omega(\log N)$ eine untere Schranke für den Aufwand zur Suche eines nächsten Nachbarn im schlimmsten Fall, und zwar für jede Dimension.

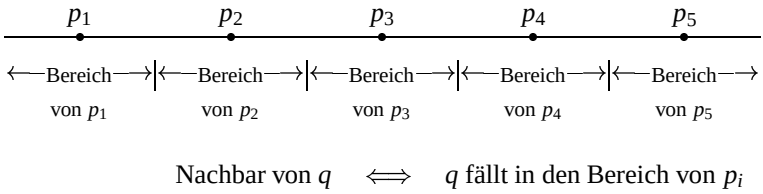


Abbildung 7.52

Erinnern wir uns: Alle gestellten Probleme können im eindimensionalen Fall leicht optimal gelöst werden, weil wir uns die Sortierung der Punktmenge zunutze machen können. Da es aber für zweidimensionale Punkte keine Sortierung gibt, läßt sich dieser Ansatz nicht auf höhere Dimensionen verallgemeinern. Bei näherem Hinsehen stellen wir aber fest, daß sich eine Eigenschaft der sortierten Punktmenge verallgemeinern läßt, die wir zur Lösung der Probleme genutzt haben: Wir haben (implizit) für jeden gegebenen Punkt p die Menge aller Punkte vorausberechnet, die näher bei p als bei irgendeinem anderen Punkt der Menge liegen. Bei der Suche nach dem nächsten Nachbarn beispielsweise haben wir dann nur noch feststellen müssen, zu welcher der vorausberechneten Punktmengen ein Anfragepunkt gehört; die Abbildung 7.52 illustriert diesen Sachverhalt.

Dasselbe Prinzip wollen wir nun auf zweidimensionale Punkte in der Ebene verallgemeinern, um eine schnellere Lösung für alle genannten Probleme zu erhalten.

7.7.2 Das Voronoi-Diagramm

Das Voronoi-Diagramm für eine Menge von Punkten in der Ebene teilt die Ebene ein in *Gebiete gleicher nächster Nachbarn*. Besteht die Menge lediglich aus zwei Punkten, so wird die Einteilung gerade durch die Mittelsenkrechte auf der Verbindungsstrecke der beiden Punkte realisiert (Abbildung 7.53).

Der geometrische Ort aller Punkte, die näher bei p_1 liegen als bei p_2, ist die Halbebene $H(p_1|p_2)$; das entsprechende gilt für p_2 und $H(p_2|p_1)$. Allgemein nennen wir für eine gegebene Menge P von Punkten und einen Punkt $p \in P$ den geometrischen Ort aller Punkte der Ebene, die näher bei p liegen als bei irgendeinem anderen Punkt aus P, die *Voronoi-Region* $VR(p)$ von p. Sie ist stets der Durchschnitt aller Halbebenen von p, gebildet mit allen anderen Punkten aus P:

$$VR(p) = \bigcap_{p' \in P \setminus \{p\}} H(p|p')$$

Die Abbildung 7.54 zeigt eine Menge von sechs Punkten und die Voronoi-Region für einen der Punkte p_1.

Das Studium dieser Regionen geht zurück auf den Mathematiker G. Voronoi (vgl. [189]). Man nennt sie manchmal auch Dirichlet-Gebiete oder Thiessen-Polygone (vgl. [149]). Die Menge aller Voronoi-Regionen für eine Menge von Punkten ist das *Voronoi-Diagramm*. Abbildung 7.55 zeigt ein Beispiel.

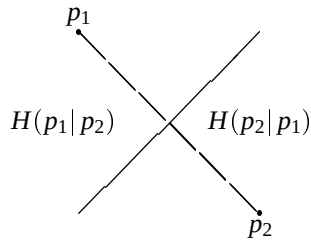
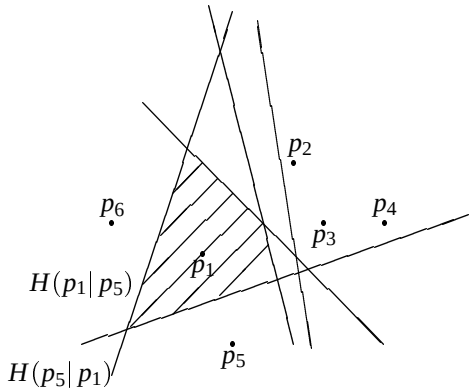


Abbildung 7.53



schraffiert: $VR(p_1)$

Abbildung 7.54

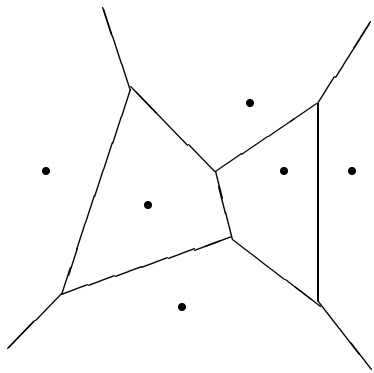
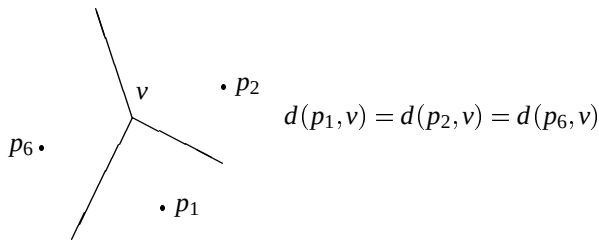


Abbildung 7.55

Wir betrachten das Voronoi-Diagramm als ebenes Netzwerk; die Knoten des Netzwerkes heißen *Voronoi-Knoten*, die Kanten *Voronoi-Kanten*.

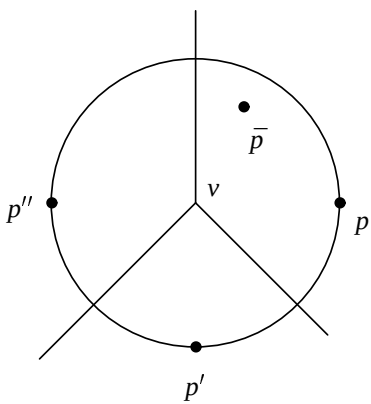
Das Voronoi-Diagramm hat eine Reihe von Eigenschaften, die es erlauben, die eingangs gestellten Probleme effizient zu lösen. Für die Suche nach einem nächsten Nachbarn eines Anfragepunktes q genügt es, die Voronoi-Region $VR(p)$ des Punktes p zu bestimmen, in der q liegt; p ist dann ein nächster Nachbar von q. Bevor wir die Lösung der Probleme mit Hilfe des Voronoi-Diagramms beschreiben, wollen wir die Eigenschaften des Voronoi-Diagramms etwas genauer betrachten. Nehmen wir (zur Vermeidung einer umständlichen Sonderfallbetrachtung) an, daß keine vier Punkte der gegebenen Punktmenge auf einem gemeinsamen Kreis liegen.

Da jeder Punkt auf der Mittelsenkrechten der Verbindungsstrecke zwischen p und p' zu p und p' den gleichen Abstand hat, liegt auch jeder Voronoi-Knoten v gleich weit von allen Punkten aus P entfernt, deren Voronoi-Regionen an v grenzen:



Weil keine vier Punkte aus P auf einem Kreis liegen, und weil zwei Punkte aus P keinen Voronoi-Knoten definieren, muß jeder Voronoi-Knoten genau drei Kanten begrenzen und auf dem Rand von genau drei Voronoi-Regionen liegen. Jeder Knoten des Voronoi-Diagramms hat also genau den Grad drei.

Ist p ein Punkt aus P einer an Voronoi-Knoten v angrenzenden Voronoi-Region, so liegen folglich gerade drei Punkte aus P, sagen wir p, p' und p'', auf dem Kreis um v mit Radius $d(p,v)$. In diesem Kreis kann kein Punkt $\overline{p}$ aus P liegen. Dann wäre nämlich $d(\overline{p},v) < d(p,v)$, und damit müßte $v \in VR(\overline{p})$ gelten, im Widerspruch zur Voraussetzung $v \in VR(p)$.



Man macht sich leicht klar, daß jeder nächste Nachbar eines Punktes $p \in P$ eine Kante der Voronoi-Region $VR(p)$ definiert; nächste Nachbarn haben also sich berührende Voronoi-Regionen.

Manche der Voronoi-Regionen sind beschränkt, andere sind unbeschränkt. Die unbeschränkten Regionen gehören genau zu denjenigen Punkten, die auf der konvexen Hülle von P liegen (Abbildung 7.56).

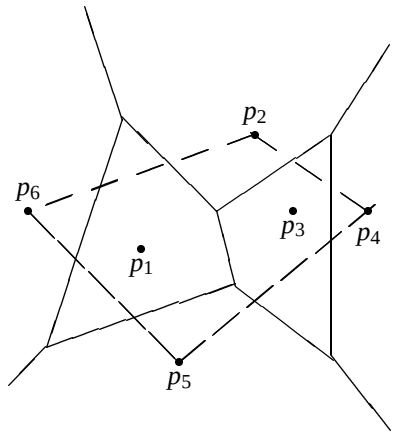


Abbildung 7.56

Diesen Sachverhalt kann man sich wie folgt klar machen. Betrachten wir zunächst eine beschränkte Voronoi-Region $VR(p)$ eines Punktes $p \in P$, und die reihum angrenzenden Voronoi-Regionen $VR(p'_1)$, $VR(p'_2), \ldots,$ $VR(p'_k)$. In unserem Beispiel grenzen $VR(p_2)$, $VR(p_3)$, $VR(p_5)$ und $VR(p_6)$ an die beschränkte Region $VR(p_1)$. Dann muß p im Polygon mit den Eckpunkten $p'_1, p'_2, \ldots, p'_k$ liegen, also nicht auf der konvexen Hülle. In unserem Beispiel liegt p_1 im Polygon mit Eckpunkten p_2, p_3, p_5 und p_6.

Wir überlegen uns noch, daß der Schluß auch in der anderen Richtung gilt, d.h. daß für $p \in P$ nicht auf der konvexen Hülle $VR(p)$ beschränkt ist. Liegt p nicht auf der konvexen Hülle, so liegt p in Innern eines Dreiecks, dessen drei Eckpunkte p'_1, p'_2, p'_3 aus P stammen. In unserem Beispiel liegt p_1 im Innern des Dreiecks p_4, p_5, p_6. Betrachten wir die drei Kreise, die durch p und jeweils zwei der Punkte p'_1, p'_2, p'_3 gehen (Abbildung 7.57). Jeder Punkt auf dem Rand der Vereinigung der drei Kreise K_{12}, K_{23} und K_{13} liegt näher an einem der Punkte p'_1, p'_2, p'_3 als an p. Dasselbe gilt ebenfalls für alle Punkte außerhalb $K_{12} \cup K_{23} \cup K_{13}$. Also muß $VR(p)$ ganz in der Vereinigung der drei Kreise enthalten sein; damit ist $VR(p)$ beschränkt.

Nun versuchen wir, die im Voronoi-Diagramm implizit repräsentierten Nachbarschaften explizit darzustellen. Dazu betrachten wir den dualen Graphen (das duale Netzwerk): Jeder (gegebene) Punkt $p \in P$ ist ein Knoten, und zwischen zwei Knoten p und p' gibt es genau dann eine (ungerichtete) Kante, wenn $VR(p)$ und $VR(p')$ sich berühren, also eine gemeinsame Voronoi-Kante haben. Die Kanten des dualen Graphen haben eine Länge, die gerade der Distanz der beiden Endknoten entspricht. Die Abbildung 7.58 zeigt den zum Voronoi-Diagramm dualen Graphen für unser Beispiel.

Der duale Graph trianguliert die Menge P, d.h., er definiert eine Zerlegung der konvexen Hülle von P in Dreiecke mit Punkten aus P als Eckpunkte. Dies kann man einsehen,

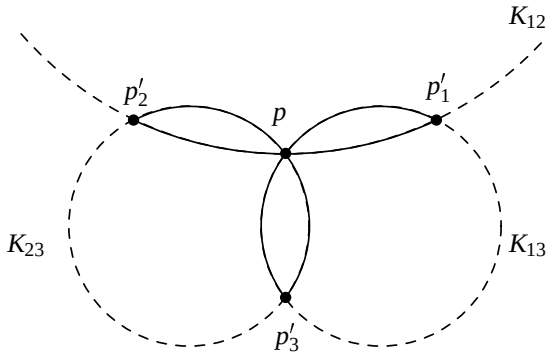


Abbildung 7.57

indem man jedem Voronoi-Knoten v das Dreieck des dualen Graphen mit Eckpunkten p'_1, p'_2, p'_3 zuordnet, wobei v auf dem Rand der Voronoi-Regionen $VR(p'_1)$, $VR(p'_2)$, $VR(p'_3)$ liegt. Dann zeigt man, daß sich diese Dreiecke nicht überlappen (sondern sich allenfalls berühren), und daß jeder Punkt der konvexen Hülle von P in einem solchen Dreieck liegt. Man beachte, daß v selbst nicht im zugehörigen Dreieck p'_1, p'_2, p'_3 liegen muß. Delaunay hat bereits 1934 gezeigt, daß der zum Voronoi-Diagramm duale Graph P trianguliert ([34, 149]); die so definierte Zerlegung heißt daher auch *Delaunay-Triangulierung*.

Damit ergibt sich direkt eine Aussage über die Anzahl der Voronoi-Knoten und Voronoi-Kanten eines Voronoi-Diagramms für eine Menge von N Punkten: ein solches Diagramm hat höchstens $2N-4$ Knoten und höchstens $3N-6$ Kanten. Weil die Delaunay-Triangulier[illegible]lanarer Graph ist, besteht sie nach Euler aus höchstens $3N-6$ Kanten und höchstens $2N-4$ Dreiecken (Flächen). Jedem Dreieck entspricht ein Voronoi-Knoten, jeder Kante der Triangulierung entspricht eine gemeinsame Kante der beiden betreffenden Voronoi-Regionen.

Das Voronoi-Diagramm erlaubt also — sobald es erst einmal berechnet ist — eine sehr kompakte und trotzdem explizite Darstellung der Nachbarschaftsverhältnisse von Punkten. Überlegen wir uns zunächst genauer, wie das Voronoi-Diagramm gespeichert werden soll, und anschließend, wie wir es denn berechnen können.

7.7.3 Die Speicherung des Voronoi-Diagramms

Wir speichern das Voronoi-Diagramm als einen in die Ebene eingebetteten planaren Graphen. [126] schlagen vor, eine doppelt verkettete Liste der Kanten (englisch: doubly connected edge list) abzuspeichern. Jede Kante wird durch ihre beiden Endpunkte (Knoten) angegeben; außerdem wird bei jeder Kante vermerkt, welche beiden Flächen sich auf beiden Seiten der Kante anschließen. Jeder Knoten v wird durch die beiden Koordinatenwerte (x_v, y_v) repräsentiert. Wir legen das übliche kartesische Koordinaten-

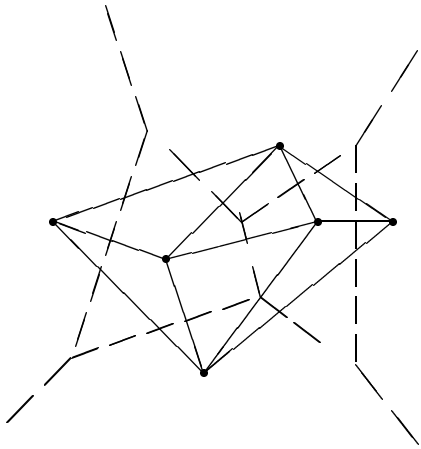
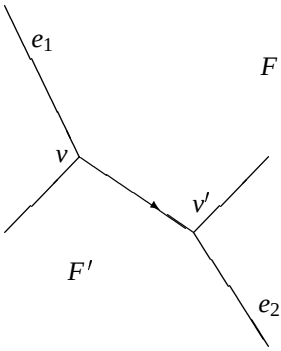


Abbildung 7.58

system zugrunde. Um die zu einer Fläche gehörenden Kanten nacheinander betrachten zu können, werden mit jeder Kante zwei Verweise auf die an den beiden Endknoten der Kante weiterführenden Kanten gespeichert. Genauer hat die doppelt verkettete Kantenliste die in Abbildung 7.59 gezeigte Gestalt.



Richtung der Kante $v\,v'$: implizit, willkürlich durch Abspeicherung festgelegt

Kante

Anfangs-knoten: v	End-knoten v'
Fläche "links": F	Fläche "rechts": F'
nächste Kante von F bei v:	nächste Kante von F' bei v':

legt implizit, willkürlich eine Richtung fest

Kante e_1 Kante e_2

Abbildung 7.59

Verwenden wir jetzt die Definition

```
type
  kantenzeiger = ↑kante;
  kante =  record
             anfangsknoten, endknoten: knoten;
             linkeflaeche, rechteflaeche: flaeche;
             anfangskante, endkante: kantenzeiger
           end
```

mit geeigneten Definitionen für *knoten* und *flaeche*, so können wir beispielsweise alle Kanten der Fläche F im Uhrzeigersinn direkt nacheinander besuchen, wenn wir schon eine Kante der Fläche F kennen:

```
var
  z1, z2 : kantenzeiger;
.
.
.
{sei z1 ein Zeiger auf eine zur Fläche F gehörende Kante;
 also entweder z1 ↑.linkeflaeche = F oder z1 ↑.rechteflaeche = F}
z2 := z1;  {starte das Umrunden der Fläche bei z1}
repeat
  {die aktuell betrachtete Kante ist z2 ↑}
  {fahre fort mit der nächsten zu F gehörenden Kante:}
  if z2 ↑.linkeflaeche = F
    then z2 := z2 ↑.anfangskante
    else z2 := z2 ↑.endkante
until z2 = z1  {Umrundung ist vollendet}
```

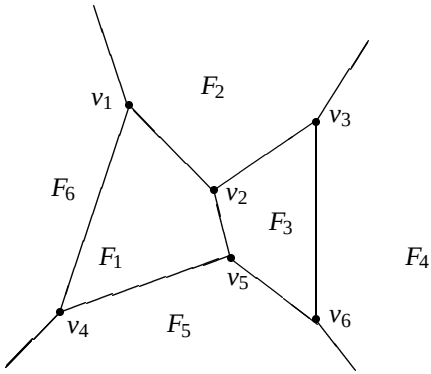
Entsprechend läßt sich leicht angeben, wie man alle mit einem Knoten inzidenten Kanten entgegen dem Uhrzeigersinn besuchen kann. Wichtig ist hier nur, daß die Laufzeit beider Operationen (Kanten einer Fläche besuchen; Kanten eines Knotens besuchen) lediglich proportional zur Anzahl der besuchten Kanten ist, wenn man bereits Zugriff auf eine der zu besuchenden Kanten hat.

Versehen wir die doppelt verkettete Kantenliste nun noch mit einem Anfangszeiger auf eine beliebige Kante, so läßt sich eine Startkante für eine Fläche oder einen Knoten in Zeit proportional zur Anzahl aller gespeicherten Kanten finden.

Für unser Voronoi-Diagramm-Beispiel ergibt sich beispielsweise die in Abbildung 7.60 gezeigte Situation.

7.7.4 Die Konstruktion des Voronoi-Diagramms

Preparata und Shamos [149] weisen darauf hin, daß in einigen Anwendungsgebieten das Berechnen des Voronoi-Diagramms nicht ein Zwischenschritt beim Lösen eines Problems ist, sondern bereits die Lösung — Beispiele findet man in der Archäologie,



Voronoi-Diagramm

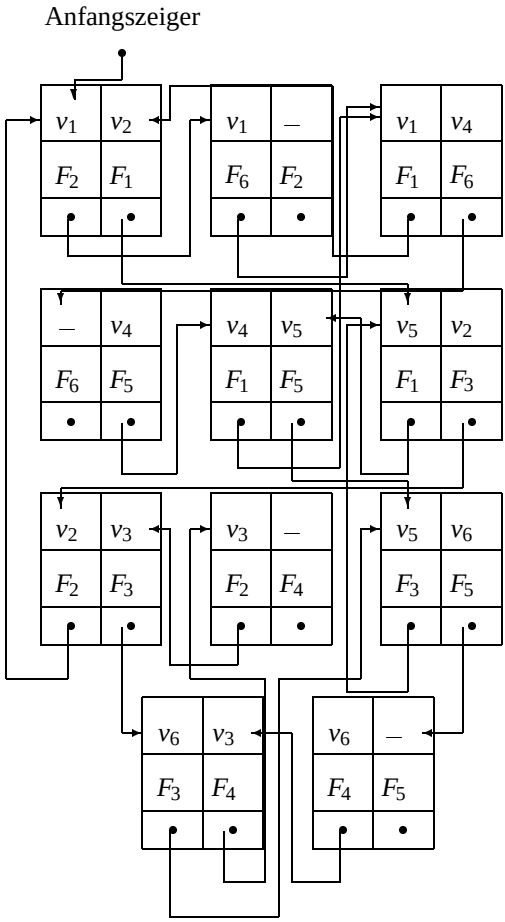


Abbildung 7.60

der Ökologie, der Chemie und der Physik. Wir wollen das Voronoi-Diagramm berechnen, um damit die eingangs beschriebenen Probleme effizienter zu lösen. Formulieren wir zunächst das Problem.

Problem: *Voronoi-Diagramm*
gegeben: Eine Menge P von N Punkten in der Ebene.
gesucht: Das Voronoi-Diagramm für P, als doppelt verkettete Kantenliste.

Ein naives Verfahren zur Berechnung des Voronoi-Diagramms könnte damit beginnen, daß für jeden Punkt $p \in P$ durch Betrachten aller anderen Punkte $p' \in P \setminus \{p\}$ die p betreffenden Halbebenen berechnet und ihr Durchschnitt gebildet werden. Damit erhält man die Voronoi-Region für jeden Punkt $p \in P$ in Zeit $\Omega(N)$ pro Punkt, also insgesamt in Zeit $\Omega(N^2)$. Solch ein Verfahren kann aber nicht als Grundlage für schnellere Algorithmen für unsere Ausgangsprobleme dienen. Fragen wir uns zunächst, wie lange denn die Berechnung des Voronoi-Diagramms mindestens dauern muß.

Im eindimensionalen Fall besteht das Voronoi-Diagramm gerade aus den "Trennstellen" für Gebiete gleicher nächster Nachbarn, wie am Ende des Abschnitts 7.7.1 angegeben. Die Voronoi-Region eines Punktes aus P ist also hier ein Intervall, das den Punkt enthält. Wenn man wieder fordert, daß aus einer Voronoi-"Kante" (das ist hier eine "Trennstelle") in konstanter Zeit auf die angrenzenden Gebiete geschlossen werden kann und umgekehrt, so kann man das Voronoi-Diagramm für eine Menge von Zahlen zum Sortieren benutzen: Man beginnt bei der kleinsten Zahl und durchläuft alle Zahlen gemäß dem Voronoi-Diagramm. Da dieses Durchlaufen lediglich lineare Zeit, Sortieren aber $\Omega(N \log N)$ Zeit benötigt, muß das Berechnen des Voronoi-Diagramms $\Omega(N \log N)$ Zeit benötigen. Im eindimensionalen Fall ist das ja mittels Sortieren auch tatsächlich leicht erreichbar.

Wir werden jetzt zeigen, wie das Voronoi-Diagramm auch im zweidimensionalen Fall, also für Punkte in der Ebene, effizient berechnet werden kann. Dazu verwenden wir ein dem Divide-and-Conquer-Prinzip folgendes Verfahren:

1. Teile P in zwei etwa gleich große Teilmengen P_1 und P_2.
2. Berechne die Voronoi-Diagramme für P_1 und P_2 rekursiv.
3. Verschmelze die beiden Voronoi-Diagramme für P_1 und P_2 zum Voronoi-Diagramm für P.

Das Ende der Rekursion ist erreicht, wenn das Voronoi-Diagramm für einen einzigen Punkt berechnet werden soll: das ist gerade die ganze Ebene. Wichtig ist, daß wir P so teilen, daß Schritt 3, das Verschmelzen der Teil-Diagramme, möglichst effizient ausführbar ist. Dabei hilft eine wichtige Beobachtung: Wenn P durch eine vertikale Linie in zwei Teile P_1 und P_2 geteilt wird, so bilden diejenigen Kanten des Voronoi-Diagramms, die sowohl an Voronoi-Regionen für Punkte aus P_1 als auch an Voronoi-Regionen für Punkte aus P_2 angrenzen, einen in vertikaler Richtung monotonen, zusammenhängenden Linienzug. Dieser Linienzug besteht am oberen und unteren Ende aus je einer Halbgeraden, mit Geradenstücken dazwischen. Die Abbildung 7.61 illustriert diese Aussage für unser Beispiel.

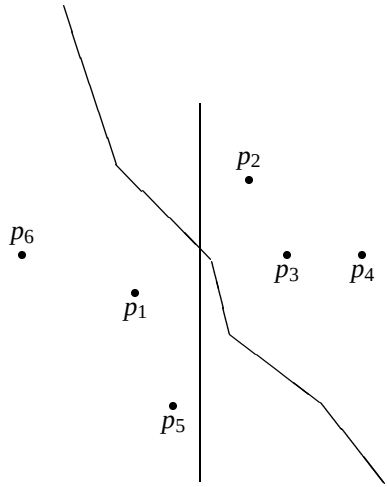


Abbildung 7.61

Das Voronoi-Diagramm für P setzt sich dann zusammen aus dem links dieses Kantenzugs liegenden Teil des Voronoi-Diagramms von P_1, dem rechts des Kantenzugs liegenden Teil des Voronoi-Diagramms von P_2 und dem Kantenzug selbst (Abbildung 7.62).

Wir präzisieren jetzt das Verfahren zur Berechnung des Voronoi-Diagramms entsprechend.

Algorithmus *Voronoi-Diagramm*
{*liefert zu einer Menge P von N Punkten in der Ebene das Voronoi-Diagramm VD(P) in Form einer doppelt verketteten Kantenliste*}

1. {*Divide*:}
   Teile P durch eine vertikale Trennlinie T in zwei etwa gleich große Teilmengen P_1 (links von T) und P_2 (rechts von T), falls $|P| > 1$ ist; sonst ist VD(P) die gesamte Ebene.
2. {*Conquer*:}
   Berechne $VD(P_1)$ und $VD(P_2)$ rekursiv.
3. {*Merge*:}
   (*a*) *Berechne den P_1 und P_2 trennenden Kantenzug K, der Teil von VD(P) ist.*
   (*b*) *Schneide den rechts von K liegenden Teil von $VD(P_1)$ ab, und schneide den links von K liegenden Teil von $VD(P_2)$ ab.*
   (*c*) *Vereinige $VD(P_1)$, $VD(P_2)$ und K; das ist VD(P).*

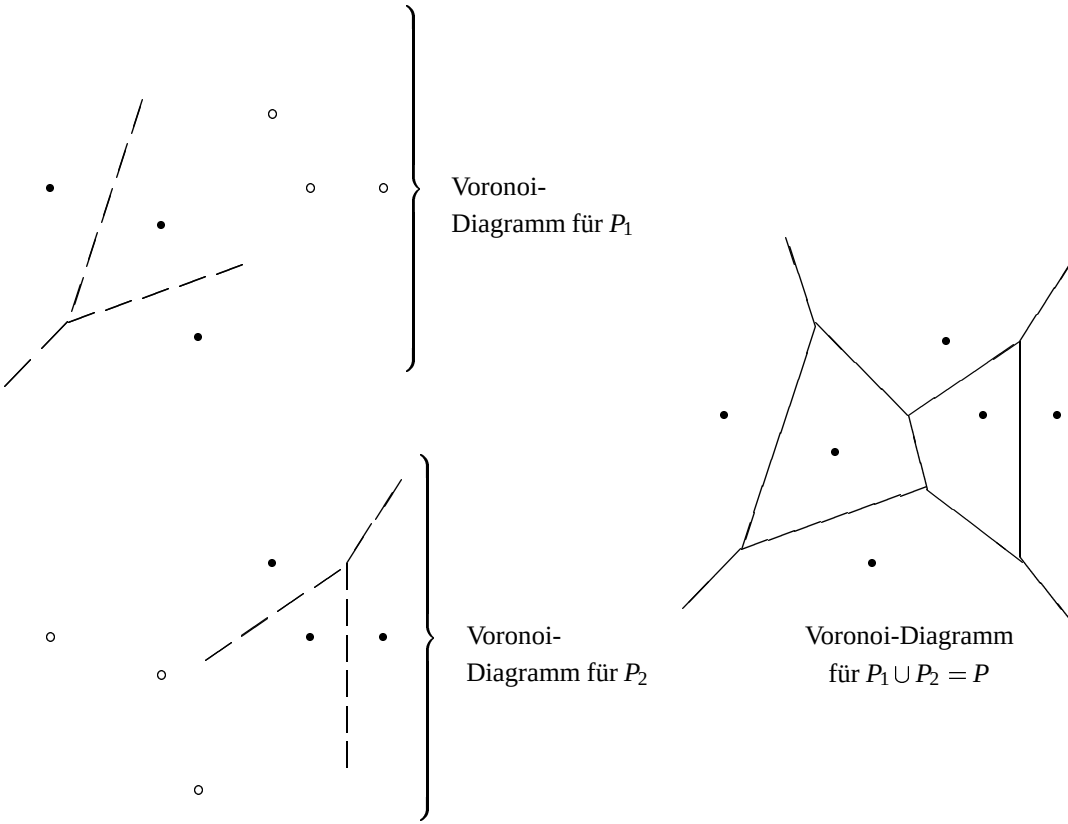


Abbildung 7.62

Schritt 1, das Aufteilen von P, ist gerade das Finden des Medians der x-Koordinaten aller Punkte, und das Verteilen der Punkte auf die beiden Teilmengen. Beides kann in linearer Zeit ausgeführt werden. Der kritische Schritt ist das Berechnen von K; das anschließende Abschneiden der überstehenden Kanten von $VD(P_1)$ und $VD(P_2)$ kann durch das Durchlaufen der jeweiligen doppelt verketteten Kantenliste in linearer Zeit geschehen. Wir wollen uns jetzt überlegen, wie auch der trennende Kantenzug K in linearer Zeit berechnet werden kann. Dann ergibt sich für die Laufzeit $T(N)$ des Verfahrens zur Berechnung des Voronoi-Diagramms für N Punkte

$$\begin{aligned} T(N) &= 2 \cdot T(N/2) + O(N) \\ T(1) &= O(1) \end{aligned}$$

und damit

$$T(N) = O(N \log N);$$

das Verfahren ist also optimal.

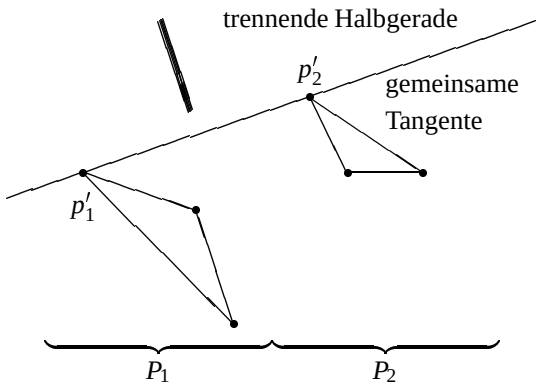


Abbildung 7.63

Wir berechnen den trennenden Kantenzug schrittweise, ein Geradenstück nach dem anderen ([167]). Dabei beginnen wir mit der oberen Halbgeraden des Kantenzugs. Diese Halbgerade muß Teil der Mittelsenkrechten zweier Punkte sein, von denen einer zu P_1 und einer zu P_2 gehört. Da beide angrenzenden Voronoi-Regionen unbeschränkt sind, müssen beide Punkte auf der konvexen Hülle der Punktmenge P liegen. Wir können diese beiden Punkte also bestimmen, indem wir eine gemeinsame Tangente von "oben" an die beiden konvexen Hüllen der Punktmengen P_1 und P_2 legen, wie in Abbildung 7.63 gezeigt.

Erinnern uns: Die beiden Voronoi-Diagramme $VD(P_1)$ und $VD(P_2)$ für die Teilmengen P_1 und P_2 von P sind bereits (rekursiv) berechnet worden. Der $VD(P_1)$ und $VD(P_2)$ trennende Kantenzug K muß in $VD(P)$ so verlaufen, daß alle Punkte der Ebene links von K näher bei einem Punkt aus P_1 als bei einem Punkt aus P_2 liegen (das gilt symmetrisch für die Punkte rechts von K). Also sind die Geradenstücke, aus denen K besteht, Teile von Mittelsenkrechten mit einem Punkt aus P_1 und einem Punkt aus P_2. Lassen wir nun einen Punkt k auf K von oben nach unten wandern, beginnend mit k auf der Mittelsenkrechten der beiden Tangentialpunkte $p'_1 \in P_1$ und $p'_2 \in P_2$. An der Stelle k_1, an der k die Grenze einer der beiden Voronoi-Regionen $VR(p'_1)$ oder $VR(p'_2)$ erreicht, muß K von dieser Mittelsenkrechten abweichen, weil sich der nächstliegende Punkt in P_1 oder P_2 für K geändert hat. Nehmen wir ohne Beschränkung der Allgemeinheit an, daß K die Grenze von $VR(p'_1)$ erreicht, und daß $VR(p''_1)$ mit $VR(p'_1)$ diese Grenze bildet, wie in Abbildung 7.64 gezeigt.

Da K in vertikaler Richtung monoton fällt, wird nun p''_1 zum K nächstliegenden Punkt in P_1. Also ergibt sich das nächste Geradenstück für K aus der Mittelsenkrechten der Verbindungsstrecke von p''_1 und p'_2. Dieser Geraden folgt K solange, bis wieder die Grenze einer Voronoi-Region erreicht ist. Im Beispiel wird die Grenze von $VR(p'_2)$ erreicht; damit folgt K nunmehr der Mittelsenkrechten der Verbindungsstrecke von p''_1 und p''_2. Dieser Prozeß wird solange fortgesetzt, bis K der Mittelsenkrechten der unte-

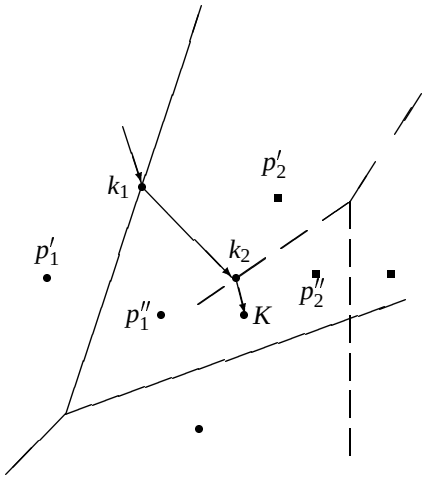


Abbildung 7.64

ren Tangentialstrecke an die beiden konvexen Hüllen von P_1 und P_2 folgt; dann ist K komplett beschrieben.

Die Berechnung des Kantenzugs K bei gegebenen Voronoi-Diagrammen $VD(P_1)$ und $VD(P_2)$ läßt sich also wie folgt beschreiben:

{*Berechnung des trennenden Kantenzugs K bei gegebenen Voronoi-Diagrammen* $VD(P_1)$, $VD(P_2)$*; wird nur ausgeführt für* $P_1 \neq \emptyset$, $P_2 \neq \emptyset$}

1. *Ermittle die beiden oberen Tangentialpunkte* $p'_1 \in P_1$ *und* $p'_2 \in P_2$ *und die beiden unteren Tangentialpunkte* $\underline{p}_1 \in P_1$ *und* $\underline{p}_2 \in P_2$. *Bestimme die Mittelsenkrechte m der Verbindungsstrecke zwischen* p'_1 *und* p'_2. *Wähle* $k = (x_k, y_k)$ *mit* $y_k = \infty$ *so, daß k auf m liegt. Setze* $K := \emptyset$.

2. **while** $(p'_1 \neq \underline{p}_1)$ **or** $(p'_2 \neq \underline{p}_2)$ **do**
   begin {*Berechnung von K fortsetzen*}
   ermittle Schnittpunkt s_1 *von m mit* $VR(p'_1)$ *unterhalb k*
   und Schnittpunkt s_2 *von m mit* $VR(p'_2)$ *unterhalb k*;
   {*nicht beide Schnittpunkte müssen existieren, aber mindestens einer*}
   if (s_1 *liegt oberhalb von* s_2) **or** (s_2 *existiert nicht*)
   then $i := 1$
   else $i := 2$;
   füge Geradenstück m von k bis s_i *zu K hinzu*;
   setze $k := s_i$;
   sei p''_i *der Punkt aus* P_i, *dessen Voronoi-Region* $VR(p''_i)$

in s_i an $VR(p_i')$ angrenzt;
setze $p_i' := p_i''$
end {*while*}

3. *Füge m von k bis $k' = (x_{k'}, y_{k'})$ mit $y_{k'} = -\infty$ und k' auf m liegend zu K hinzu.*

Die Zeit zur Berechnung von K darf $O(|P_1| + |P_2|)$ nicht übersteigen, wenn zur Berechnung des Voronoi-Diagramms für P nicht mehr als $O(N \log N)$ Zeit zur Verfügung steht, für $|P| = N$. Nehmen wir (induktiv) an, daß die konvexe Hülle für P_1 und P_2 bei der Berechnung von K bekannt ist, so können alle vier Tangentialpunkte in sublinearer Zeit berechnet werden. Mit Hilfe der berechneten gemeinsamen Tangenten läßt sich ebenso die konvexe Hülle von $P_1 \cup P_2$ in höchstens linearer Zeit angeben; die rekursive Konstruktion der konvexen Hülle ist also genügend effizient sichergestellt.

Alle Operationen im Innern der while-Schleife (Schritt 2) außer dem Ermitteln des nächsten Schnittpunktes benötigen lediglich konstante Schrittzahl, da mit Hilfe der doppelt verketteten Kantenlisten für $VD(P_1)$ und $VD(P_2)$ direkt auf benachbarte Voronoi-Regionen zugegriffen werden kann, wenn die gemeinsame Voronoi-Kante bekannt ist. Weil K aus $\Theta(|P_1| + |P_2|)$ Geradenstücken bestehen kann, benötigt dieser Teil der Operationen der Schleife also insgesamt höchstens $O(|P_1| + |P_2|)$ viele Schritte.

Das Finden aller nächsten Schnittpunkte von Mittelsenkrechten mit Voronoi-Regionen entlang K darf insgesamt ebenfalls höchstens $O(|P_1| + |P_2|)$ Schritte kosten. Daß diese Schrittzahl tatsächlich genügt, ist nicht so offensichtlich, wenn man bedenkt, daß K $\Theta(|P_1| + |P_2|)$ Voronoi-Regionen passieren kann und daß eine Voronoi-Region $\Theta(|P_1| + |P_2|)$ Kanten haben kann. Alle Voronoi-Regionen zusammen haben aber auch nur $O(|P_1| + |P_2|)$ Kanten. Da wir im Innern der **while**-Schleife aber jeweils zwei Schnittpunkte, s_1 und s_2, berechnen, aber nur einen davon (den weiter oben liegenden) verwenden, müssen wir sicherstellen, daß die Kanten der Voronoi-Region des nicht verwendeten Schnittpunktes bei späteren Schnittpunktberechnungen nicht wieder inspiziert werden müssen. Es ist also nicht effizient genug, zur Schnittpunktberechnung für p_1' (bzw. p_2') alle Kanten von $VR(p_1')$ (bzw. $VR(p_2')$) zu besuchen, und für jede Kante eine Schnittpunktprüfung vorzunehmen.

Eine effiziente Realisierung der Schnittpunktberechnung findet man mit folgender Überlegung. Nehmen wir (ohne Beschränkung der Allgemeinheit) an, für p_1'' sei bereits eine Schnittpunktberechnung für die Mittelsenkrechte m der Verbindungsstrecke von p_1'' und p_2' durchgeführt worden, aber der errechnete Schnittpunkt s_1 sei nicht gewählt worden. Dann wird p_2' von p_2'' abgelöst, die neue Mittelsenkrechte sei m'. Diese Situation ist in Abbildung 7.65 gezeigt.

Für p_1'' muß erneut eine Schnittpunktberechnung von $VR(p_1'')$, diesmal mit m', durchgeführt werden. Der Übergang von p_2' zu p_2'' kann aber in K (von oben nach unten betrachtet) nur einen Knick nach rechts zur Folge haben. Also schneidet m' $VR(p_1'')$ im Uhrzeigersinn nach s_1 (das ist links von s_1). Das Entsprechende gilt auch, wenn K mehrere Male hintereinander Voronoi-Regionen von Punkten aus P_2 passiert, bevor K $VR(p_1'')$ verläßt. Daher genügt es bei der wiederholten Schnittpunktberechnung für $VR(p_1'')$, nur die im Uhrzeigersinn auf den zuletzt berechneten Schnittpunkt folgenden Kanten (inklusive dieser Kanten selbst) zu inspizieren. Sobald ein (nächster) Schnittpunkt gefunden ist, müssen wegen der Konvexität von $VR(p_1'')$ für die gegebe-

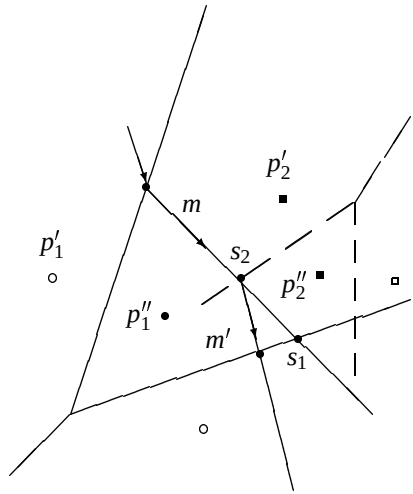


Abbildung 7.65

ne Mittelsenkrechte keine weiteren Kanten mehr betrachtet werden. Insgesamt werden so höchstens alle Kanten von $VR(p_1'')$ einmal betrachtet, zuzüglich der wiederholten Betrachtung je einer Kante für das Fortschreiten von K in P_2.

Für wiederholtes Finden von Schnittpunkten für $VR(p_2')$ gelten diese Betrachtungen entsprechend, wobei die Kanten von $VR(p_2')$ entgegen dem Uhrzeigersinn besucht werden müssen.

Das Besuchen der Kanten einer Voronoi-Region im Gegenuhrzeigersinn ist (ebenso wie im Uhrzeigersinn, vgl. Abschnitt 7.7.3) in linearer Zeit möglich, weil alle Voronoi-Knoten nur mit drei Kanten inzidieren. Der Schnittpunkt einer Voronoi-Kante mit einer Geraden kann in konstanter Zeit berechnet werden; also können alle Schnittpunkte während der Konstruktion von K in linearer Zeit berechnet werden. Man kann sich leicht überlegen, wie man mit Hilfe der doppelt verketteten Kantenlisten der Voronoi-Diagramme für P_1 und P_2 und des Kantenzugs K eine doppelt verkettete Kantenliste des Voronoi-Diagramms für P erzeugt; wir überlassen es dem Leser, die Details auszufüllen.

Damit ist gezeigt, daß (rekursiv) aus $VD(P_1)$ und $VD(P_2)$ in linearer Zeit $VD(P)$ berechnet werden kann, daß also insgesamt das Voronoi-Diagramm $VD(P)$ für eine Menge P von N Punkten in $O(N \log N)$ Zeit bestimmt werden kann. Weil sich mit Hilfe des Voronoi-Diagramms sortieren läßt, ist diese Laufzeit optimal. Das Voronoi-Diagramm für N Punkte kann mit $O(N)$ Speicherplatzbedarf in Form einer doppelt verketteten Kantenliste abgespeichert werden.

7.7.5 Lösungen für Distanzprobleme

Wir wollen jetzt zeigen, wie das Voronoi-Diagramm zur Lösung der im Abschnitt 7.7.1 angegebenen Distanzprobleme eingesetzt werden kann.

Für das Problem, ein *dichtestes Punktepaar* (closest pair) in einer Menge P von N Punkten zu finden, ist eine optimale Lösung jetzt offensichtlich, da für jeden Punkt $p \in P$ jeder nächste Nachbar $p' \in P$ von p eine an $VR(p)$ angrenzende Voronoi-Region $VR(p')$ hat. Das Problem kann also wie folgt gelöst werden:

Algorithmus *Dichtestes Punktepaar*
{*liefert zu einer Menge P von N Punkten in der Ebene ein Paar von Punkten mit minimaler Distanz unter allen Punktepaaren in P*}

1. *Konstruiere das Voronoi-Diagramm $VD(P)$ für P.*
2. *Durchlaufe die doppelt verkettete Kantenliste für $VD(P)$ und ermittle dabei das Minimum der Distanz benachbarter Punkte sowie ein Punktepaar, das diese Distanz realisiert.*

Schritt 1 kann gemäß Abschnitt 7.7.4 in $O(N \log N)$ Zeit ausgeführt werden. Da die Anzahl der Knoten und Kanten des Voronoi-Diagramms mit $O(N)$ beschränkt ist und da zu jeder Voronoi-Kante ein Paar benachbarter Punkte gehört, ist Schritt 2 sogar in $O(N)$ Zeit ausführbar. Insgesamt ergibt sich also eine Laufzeit von $O(N \log N)$; diese Laufzeit ist optimal (vgl. Abschnitt 7.7.1).

Das Problem, *alle nächsten Nachbarn* (all nearest neighbors) zu finden, löst man völlig analog.

Algorithmus *Alle nächsten Nachbarn*
{*liefert zu einer Menge P von N Punkten in der Ebene zu jedem Punkt in P einen nächsten Nachbarn in P, also eine Menge von N Punktepaaren*}

1. *Konstruiere das Voronoi-Diagramm $VD(P)$ für P.*
2. *Durchlaufe die doppelt verkettete Kantenliste für $VD(P)$ so, daß der Reihe nach für jeden Punkt p alle Voronoi-Kanten von $VR(p)$ betrachtet werden. Dabei wird für jeden Punkt ein nächster Nachbar unter allen Punkten mit benachbarter Voronoi-Region ermittelt.*

Schritt 1 kann wiederum in $O(N \log N)$ Zeit ausgeführt werden und Schritt 2 benötigt sogar nur $O(N)$ Zeit, weil das zu einer Voronoi-Kante gehörige Punktepaar p, p' höchstens zweimal, nämlich bei der Bestimmung eines nächsten Nachbarn für p und für p', betrachtet wird. Damit ist die gesamte Laufzeit $O(N \log N)$; das ist gemäß Abschnitt 7.7.1 optimal.

Das Problem, einen *minimalen spannenden Baum* (minimum spanning tree) für einen Graphen mit Kantenbewertungen zu finden, wird im Kapitel 8 ausführlich behandelt. Wir wollen hier ein Verfahren auf den Fall einer Menge von Punkten in der Ebene spezialisieren.

Algorithmus: *Minimaler spannender Baum*
{*liefert zu einer Menge P von N Punkten in der Ebene einen minimalen spannenden Baum für P in Gestalt einer Menge von Kanten*}

1. *Beginne mit einer Menge von Bäumen, wobei jeder Baum ein Punkt der Menge ist.*
2. *Solange noch mehr als ein Baum vorhanden ist, führe aus:*
   - 2.1. *Wähle einen Baum T aus.*
   - 2.2. *Finde ein Punktepaar $p, p' \in P$ so, daß p zu T gehört, p' nicht zu T gehört und $d(p,p')$ minimal ist unter allen solchen Punktepaaren.*
   - 2.3. *Sei T' der Baum, zu dem p' gehört. Verbinde T und T' durch die Kante zwischen p und p'; T und T' werden aus der Menge der Bäume gelöscht, und der neu entstandene Baum wird dort eingetragen.*

Der entscheidende Schritt ist das Finden eines Paars dichtester Punkte, Schritt 2.2. Alle anderen Teile können effizient implementiert werden, wie in Kapitel 8 beschrieben. Es ist natürlich ineffizient, jedes Punktepaar in Schritt 2.2 zu überprüfen. Hier ist das Voronoi-Diagramm die entscheidende Hilfe, denn die Voronoi-Regionen eines in Schritt 2.2 gewählten Punktepaars müssen aneinander angrenzen.

Allgemein gilt für eine beliebige Aufteilung der Punktmenge P in disjunkte Teilmengen P_1 und P_2, daß die kürzeste Verbindung zweier Punkte, von denen einer zu P_1 und einer zu P_2 gehört, zwischen Punkten mit angrenzenden Voronoi-Regionen realisiert wird. Um dies einzusehen, nehmen wir an, p'_1 und p'_2 seien zwei Punkte, die minimale Distanz zwischen P_1 und P_2 realisieren, mit $p'_1 \in P_1$ und $p'_2 \in P_2$. Wenn nun die Voronoi-Region $VR(p'_2)$ nicht an $VR(p'_1)$ angrenzt, so liegt der Mittelpunkt p_m der Verbindungsstrecke zwischen p'_1 und p'_2 außerhalb von $VR(p'_1)$. Damit schneidet der Rand von $VR(p'_1)$ die Verbindungsstrecke $\overline{p'_1 p'_2}$ in einem Punkt p''_1, der näher bei p'_1 liegt als p_m. Die Voronoi-Kante von $VR(p'_1)$ durch p''_1 trennt $VR(p'_1)$ und $VR(p')$, für einen Punkt $p' \in P$. Dieser Punkt p' liegt auf dem Kreis mit Radius $d(p''_1, p'_1)$ um den Punkt p''_1, also jedenfalls innerhalb des Kreises mit Radius $d(p_m, p'_1)$ um Punkt p''_1. Diese Situation ist in Abbildung 7.66 illustriert.

Damit ist $d(p', p'_1) < d(p'_2, p'_1)$ und auch $d(p', p'_2) < d(p'_2, p'_1)$. Ob nun p' zu P_1 oder P_2 gehört, stets ist die Folge, daß p'_1 und p'_2 kein Punktepaar mit minimaler Distanz zwischen P_1 und P_2 gewesen sein kann. Also grenzen die Voronoi-Regionen der Punkte p'_1 und p'_2 aneinander.

Damit genügt es, bei der Suche nach einem Punktepaar mit minimalem Abstand in Schritt 2.2 nur Punktepaare mit angrenzender Voronoi-Region zu betrachten. Der minimale spannende Baum ist also als Teil des zum Voronoi-Diagramm dualen Graphen mit geradlinigen Kanten, der Delaunay-Triangulierung, konstruierbar. Einen minimalen spannenden Baum für unser Beispiel zeigt die Abbildung 7.67.

Die Berechnung eines minimalen spannenden Baumes für einen Graphen mit N Knoten und Kanten kann in Zeit $O(N \log N)$ ausgeführt werden, wie wir in Kapitel 8 zeigen werden; für planare Graphen genügt sogar Zeit $O(N)$. Damit kann ein minimaler spannender Baum für eine Menge von Punkten in Zeit $O(N \log N)$ berechnet werden: Man berechnet das Voronoi-Diagramm in Zeit $O(N \log N)$, bildet den dualen Graphen, die Delaunay-Triangulierung, in Zeit $O(N)$ durch Durchlaufen der doppelt verketteten

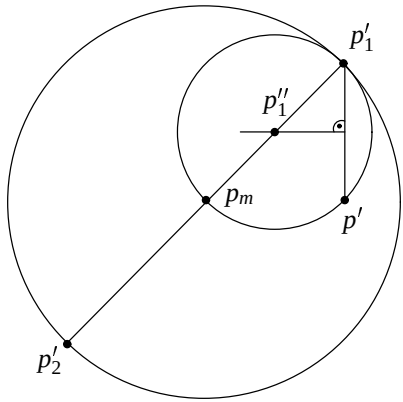


Abbildung 7.66

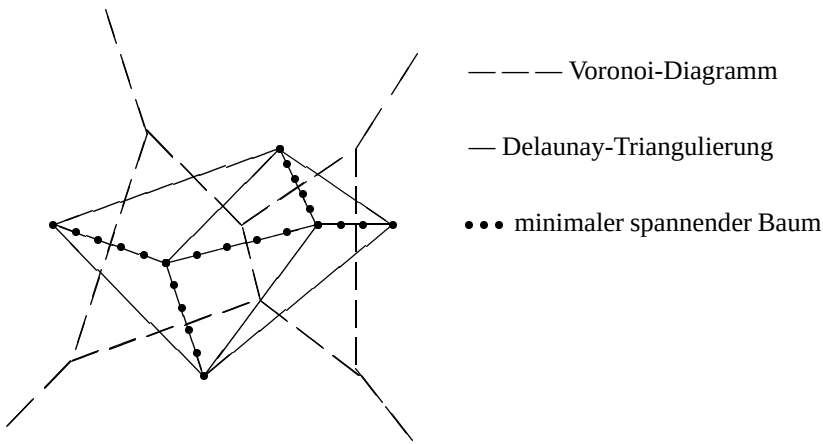


Abbildung 7.67

Kantenliste für das Voronoi-Diagramm und berechnet anschließend einen minimalen spannenden Baum der Delaunay-Triangulierung. Daß dies optimal ist, wurde bereits in Abschnitt 7.7.1 gezeigt.

Die bisher gelösten Probleme waren allesamt Probleme mit fest gegebener Objektmenge und vorgegebener Frage. Betrachten wir nun eine Lösung zum Problem der Anfrage nach einem nächsten Nachbarn bei gegebener, fester Punktmenge P für beliebige, zunächst unbekannte Anfragepunkte. Da viele dieser Anfragen beantwortet werden sollen, wollen wir mit einigem Vorbereitungsaufwand P so präparieren, daß Anfragen effizient beantwortet werden können. Die *Suche nach einem nächsten Nachbarn* für einen Anfragepunkt (nearest neighbor search, best match), wird dann in zwei Schritten erledigt:

Algorithmus 1 *Vorbereitung für "Suche nächsten Nachbarn"*
{*liefert zu einer Menge P von N Punkten in der Ebene eine Datenstruktur für P mit einer effizienten Unterstützung der Suchanfrage*}

Algorithmus 2 *Suche nächsten Nachbarn*
{*liefert zu einem Anfragepunkt q der Ebene einen nächsten Nachbarn* $p \in P$}
Verwende die angebotene Suchanfrageoperation für P und q.

Wir müssen also lediglich noch den ersten Teil, den Vorbereitungsschritt, präzisieren. Dabei hilft wieder das Voronoi-Diagramm. Für einen Anfragepunkt q ist ein solcher Punkt $p \in P$ nächster Nachbar unter allen Punkten aus P, in dessen Voronoi-Region q fällt; $VR(p)$ war ja gerade entsprechend definiert (vgl. Abschnitt 7.7.2). Die zu unterstützende Operation für beliebiges q ist also das Finden der (einer) Voronoi-Region $VR(p)$, die q enthält (ein *point location problem*). Auch wenn das Voronoi-Diagramm als bekannt vorausgesetzt wird, ist diese Operation nicht ganz einfach effizient ausführbar. Unter den verschiedenen Methoden hierfür wollen wir eine näher betrachten, die Methode der *hierarchischen Triangulierung* [87].

Zunächst wird das zu betrachtende Gebiet trianguliert, also in Dreiecke zerlegt, deren Ecken aus der vorgegebenen Punktmenge stammen. In unserem Fall ist dies die Menge der Voronoi-Punkte, also *nicht* die Menge P. Da Voronoi-Regionen im allgemeinen mehr als drei Kanten besitzen, müssen sie in Dreiecke unterteilt werden; unbeschränkte Voronoi-Regionen werden hier zunächst ignoriert. Die entstehende Triangulierung der beschränkten Voronoi-Regionen umgeben wir mit einem Dreieck; die Differenzregion wird ebenfalls trianguliert. Die Abbildung [illegible] zeigt eine solche Triangulierung für unser Beispiel.

Die Anzahl der Dreiecke einer solchen Triangulierung ist proportional zur Anzahl der Voronoi-Knoten, also linear in der Anzahl N der Punkte in P. Die Triangulierung läßt sich in $O(N \log N)$ Schritten ermitteln, etwa mit Hilfe eines Scan-Line-Verfahrens.

In dieser Triangulierung des Voronoi-Diagramms von P suchen wir nun nach einem Dreieck, das q enthält. Ist das gefundene Dreieck Teil einer beschränkten Voronoi-Region $VR(p)$, so ist p nächster Nachbar von q; andernfalls ist das gefundene Dreieck Teil einer unbeschränkten Voronoi-Region oder q liegt ganz außerhalb des umschließenden Dreiecks. Dann führen wir eine binäre Suche auf den zyklisch geordneten unbeschränkten Voronoi-Regionen (repräsentiert durch die trennenden Halbgeraden) aus, um einen Punkt zu finden, in dessen Voronoi-Region q liegt. Diese Suche kann in

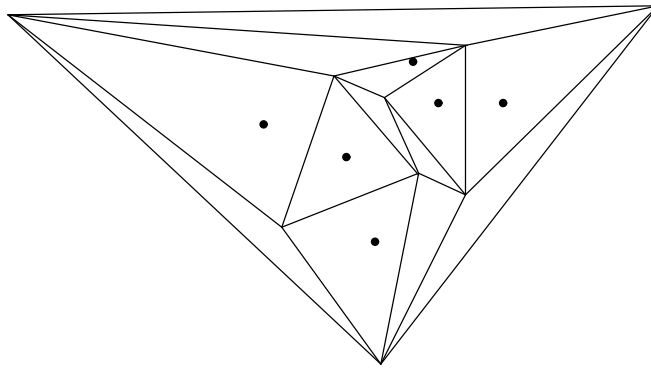


Abbildung 7.68

$O(\log N)$ Schritten beendet werden, wenn N die Anzahl der Punkte in P ist. Wesentlich für die Effizienz der Beantwortung der Suchanfrage ist jetzt noch, daß wir das Dreieck der Triangulierung, das q enthält, schnell finden.

Zu diesem Zweck vergröbern wir die bisher betrachtete Triangulierung in mehreren Schritten, bis wir schließlich nur noch ein Dreieck vorfinden. Ein Vergröberungsschritt besteht darin, eine Menge von Punkten, die nicht durch Kanten verbunden sind (unabhängige Punkte) und nicht auf dem Rand der Triangulierung liegen, zusammen mit ihren inzidenten Kanten zu entfernen und die entstehenden polygonalen Gebiete neu zu triangulieren. Ein Vergröberungsschritt macht also aus einer Triangulierung eine gröbere Triangulierung. Wir wenden nacheinander mehrere Vergröberungsschritte an, bis die Triangulierung schließlich nur noch aus einem einzigen Dreieck besteht. Die Abbildungen 7.69 bis 7.73 zeigen eine Folge von fünf Triangulierungen für unser Beispiel. Mit $\odot$ markierte Punkte werden im nächsten Schritt entfernt; Kanten, die im letzten Schritt hinzugenommen wurden, sind gestrichelt gezeichnet. Die Dreiecke sind für spätere Bezugnahme mit Namen versehen.

Suchen wir nun mit einem Anfragepunkt q nach einem Dreieck der feinsten Triangulierung, das q enthält, so beginnen wir die Suche mit der gröbsten Triangulierung. Für diese stellen wir fest, ob q überhaupt im Dreieck liegt. Dieser Test kann für einen gegebenen Punkt und ein gegebenes Dreieck in einer konstanten Anzahl von Schritten ausgeführt werden. Dann fahren wir mit der Suche in der nächstfeineren Triangulierung fort. Dort inspizieren wir alle Dreiecke, die mit dem soeben betrachteten einen nichtleeren Durchschnitt haben, denn nur in diesen Dreiecken kann q liegen. Eines der inspizierten Dreiecke muß q enthalten. Wir setzen das Verfahren mit diesem Dreieck und der nächstfeineren Triangulierung fort, bis wir schließlich in der feinsten Triangulierung dasjenige Dreieck bestimmt haben, das q enthält.

Um diesen Suchvorgang zu unterstützen, wird zunächst aus der Hierarchie der Triangulierungen eine spezielle verkettete Suchstruktur gebildet. Jeder Knoten der Suchstruktur repräsentiert ein Dreieck. Ein ausgezeichneter Knoten (die Wurzel) repräsentiert das Dreieck der gröbsten Triangulierung. Die Blätter der Struktur repräsentieren die Dreiecke der feinsten Triangulierung. Für jedes im Verlauf der Vergröberung der

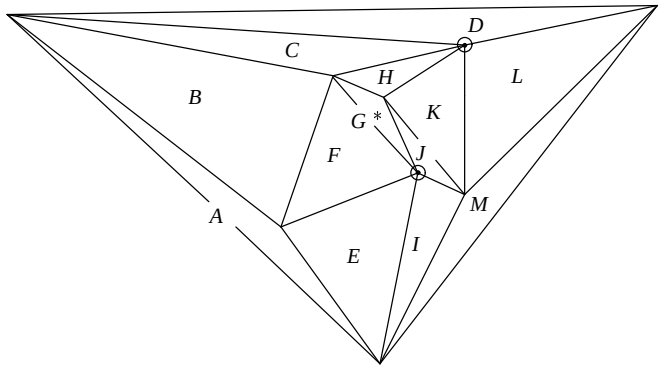


Abbildung 7.69

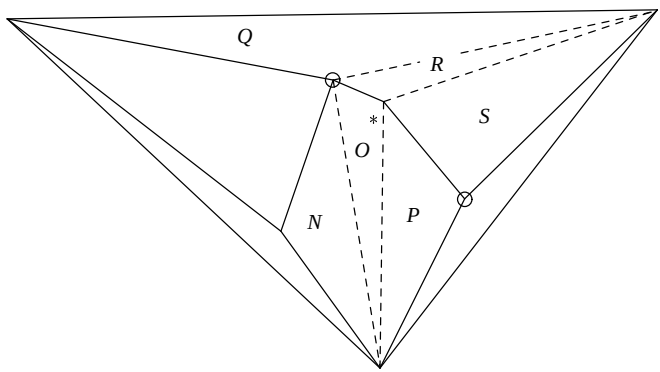


Abbildung 7.70

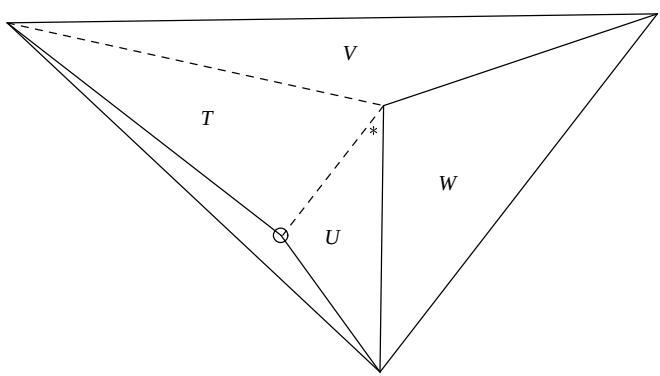


Abbildung 7.71

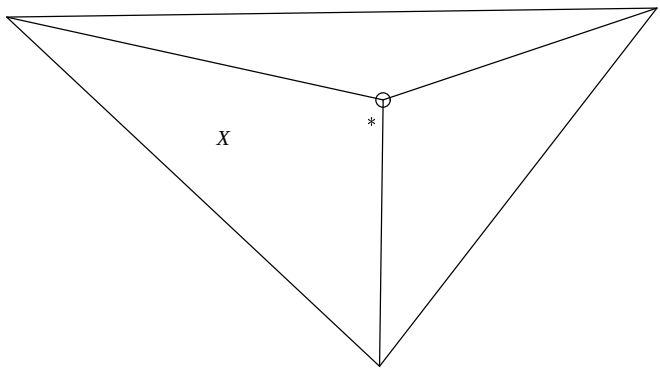


Abbildung 7.72

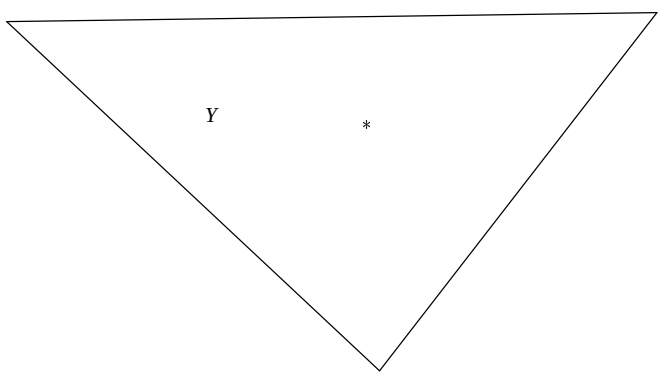


Abbildung 7.73

Triangulierung neugebildete Dreieck gibt es einen Knoten zwischen der Wurzel und den Blättern (inklusive der Wurzel selbst, die ja auch ein neugebildetes Dreieck repräsentiert). Die Verweise der Knoten untereinander sind wie folgt angelegt: Ein Knoten k, der Dreieck d repräsentiert, besitzt einen Zeiger auf Knoten k' mit Dreieck d' genau dann, wenn in einem Vergröberungsschritt von Triangulierung T' zu Triangulierung T Dreieck d' entfernt wurde, Dreieck d neu entstand und d und d' sich überlappen.

Für unser Beispiel sieht die Struktur für die Hierarchie der Triangulierungen wie in Abbildung 7.74 gezeigt aus.

Verfolgen wir die Suche nach dem mit $*$ in den Triangulierungen eingetragenen Punkt q^*. Zunächst stellen wir fest, ob q^* im Dreieck Y liegt. Da dies der Fall ist, prüfen wir für alle Nachfolger des Knotens Y, ob q^* im zugehörigen Dreieck liegt. Der Test mit V, W und X ergibt, daß q^* in X liegt. Jetzt ist X aktueller Knoten, und das Verfahren wird fortgesetzt. Unter den Nachfolgern T, U und A von X ist U das q^* enthaltende Dreieck. Von N und O enthält O q^*, und schließlich ist aus E, F und G das q^* enthaltende Dreieck

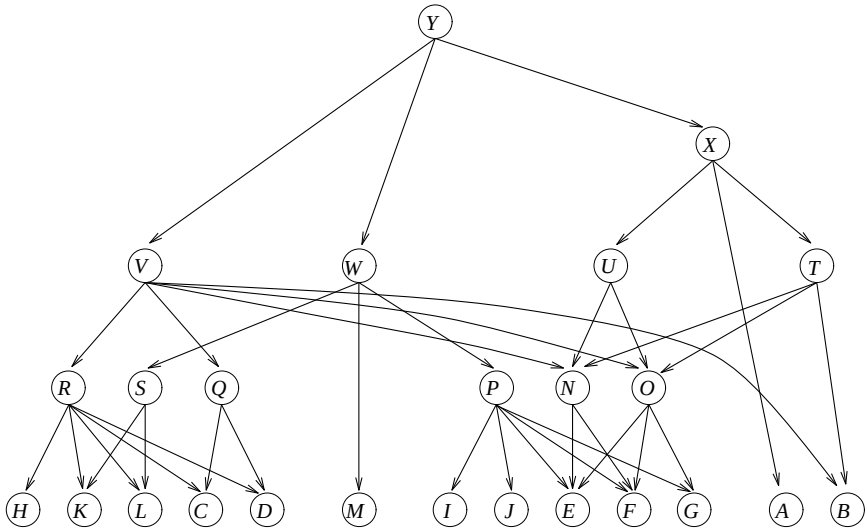


Abbildung 7.74

G. Da dies ein Blatt ist, sind wir bei der feinsten Triangulierung angelangt, und die zu Dreieck G gehörende Voronoi-Region $VR(p)$, die beispielsweise über einen weiteren Zeiger erreichbar ist, enthält q^*. Damit ist p_1 nächster Nachbar von q^*.

Die Laufzeit dieses Verfahrens hängt ab von der Länge des längsten Pfades in der Suchstruktur von der Wurzel zu einem Blatt und von der Anzahl der Nachfolger von Knoten. Das erstere ist gerade die Anzahl der Triangulierungen (Vergröberungsschritte), das letztere die Anzahl der Dreiecke, die ein neugebildetes Dreieck in der nächstfeineren Triangulierung höchstens überlappen kann. Für beides ist offenbar die Wahl der zu entfernenden Punkte in einem Vergröberungsschritt maßgebend.

Die Suche nach dem Elementardreieck (Dreieck der feinsten Triangulierung), das einen gegebenen Punkt enthält, kann nicht schneller als in $\Omega(\log N)$ Zeit für $\Theta(N)$ Elementardreiecke ausgeführt werden, weil dies schon eine untere Schranke für die Suche im eindimensionalen Fall ist. Ein Suchverfahren ist also sicher optimal, wenn es mit $O(\log N)$ Schritten auskommt. Das ist der Fall, wenn es höchstens $O(\log N)$ Triangulierungen gibt und wenn jedes neugebildete Dreieck höchstens eine konstante Anzahl von Dreiecken der nächstfeineren Triangulierung überlappt. Dann werden nämlich bei der Suche nur $O(\log N)$ Knoten insgesamt betrachtet. Weil die Anzahl der Elementardreiecke proportional ist zur Anzahl der Voronoi-Knoten und weil diese wiederum proportional ist zur Anzahl N der gegebenen Punkte, ergibt sich damit ein Suchverfahren, mit dem die Suche nach dem nächsten Nachbarn in Zeit $O(\log N)$ ausgeführt werden kann, das also optimal ist.

Auch der Speicherbedarf für eine solche Suchstruktur ist minimal: Da es insgesamt $\Theta(N)$ Knoten in dieser Struktur gibt, von denen jeder nur konstant viele Verweise speichert, genügt $\Theta(N)$ Speicherplatz.

Wir wollen nun die von Kirkpatrick vorgeschlagene Wahl für die zu entfernenden Punkte angeben, die beide gestellten Bedingungen erfüllt. Daß die Anzahl aller Trian-

gulierungen durch $O(\log N)$ beschränkt ist, zeigen wir, indem wir nachweisen, daß sich bei jedem Vergröberungsschritt die Anzahl der Punkte einer Triangulierung mindestens um einen konstanten Faktor verringert. Die Regel für das Entfernen von Punkten ist dann die folgende: Entferne eine Menge unabhängiger Punkte, die jeweils einen Grad kleiner als g haben; g ist eine sorgfältig gewählte Konstante.

In einem Vergröberungsschritt inspizieren wir also in beliebiger Reihenfolge alle Punkte der Triangulierung, die nicht auf dem Rand liegen, und entfernen jeden Punkt mit Grad kleiner als g, es sei denn, einer seiner Nachbarn ist bereits entfernt worden. Es ist offensichtlich, daß dann jedes neu gebildete Dreieck nur weniger als g alte Dreiecke überlappen kann.

Um zu zeigen, daß stets mindestens ein fester Anteil aller Punkte auf diese Weise entfernt werden kann, folgen wir dem Gang der vereinfachten Argumentation aus [149], die eine asymptotische Aussage abzuleiten gestattet. Nach Euler gibt es in einer Triangulierung mit $n = \Theta(N)$ Punkten genau $3n-6$ Kanten, wenn der Rand der Triangulierung ein Dreieck ist. Summiert man die Grade aller Punkte, so ergibt sich ein Wert kleiner als $6n$, weil jede der $3n-6$ Kanten zum Grad von genau 2 Punkten beiträgt. Also muß es mindestens $n/2$ Punkte mit Grad kleiner als 12 geben (sonst würde die Summe der Grade der $n/2$ Punkte mit höchsten Graden schon mindestens $6n$ betragen). Wählen wir also für den das Entfernen bestimmenden Grad g den Wert 12. Wenn ein Punkt mit Grad kleiner als 12 entfernt wird, so können seine Nachbarn nicht mehr entfernt werden; die Anzahl dieser Nachbarn ist der Grad des Punktes, also weniger als 12. Von allen Punkten mit Grad kleiner als 12 können also gegebenenfalls die drei Eckpunkte auf dem Rand der Triangulierung nicht entfernt werden, und von den verbleibenden Punkten kann im schlechtesten Fall nur $\frac{1}{12}$ entfernt werden. Die Anzahl v der zu entfernenden Punkte ist also nach unten beschränkt durch

$$v \geq \lfloor \frac{1}{12}(\frac{n}{2} - 3) \rfloor$$

Der Anteil β der mindestens zu entfernenden Punkte unter n Punkten ist dann

$$\beta = \frac{v}{n} \geq \frac{1}{24} - \frac{1}{4n}$$

Für genügend großes n, etwa $n \geq 12$, ist dies

$$\beta \geq \frac{1}{24} - \frac{1}{48} = \frac{1}{48} > 0$$

Damit ist gezeigt, daß stets ein fester (wenn auch sehr kleiner) Bruchteil aller Punkte in einem Vergröberungsschritt entfernt wird und folglich die Anzahl aller Triangulierungen mit $O(\log N)$ beschränkt ist. Also arbeitet der beschriebene Algorithmus zur Suche eines nächsten Nachbarn in einer Menge von N Punkten in optimaler Zeit, mit $O(\log N)$ Schritten.

Das Herstellen der hierarchischen Triangulierung beginnt mit der Berechnung des Voronoi-Diagramms in $O(N \log N)$ Schritten. Dann wird die feinste Triangulierung in $O(N \log N)$ Schritten berechnet. In jedem Vergröberungsschritt werden alle Punkte und alle Kanten der jeweiligen Triangulierung inspiziert, um zu entscheiden, welche Punkte entfernt werden. Da jeweils ein fester Bruchteil aller Punkte (und damit auch aller Kanten) entfernt wird, inspiziert man somit insgesamt $O(N)$ Punkte und Kanten. Für jeden entfernten Punkt muß ein neu entstandenes Polygon trianguliert werden. Da dieses Polygon aber nur konstant viele Kanten besitzt (nämlich weniger als g), kann eine solche Triangulierung in konstanter Zeit gefunden werden. Für alle $O(N)$ Triangulierungen genügen also insgesamt $O(N)$ Schritte. Die Zeiger der Suchstruktur ergeben sich dabei asymptotisch ohne zusätzlichen Aufwand. Damit genügen $O(N \log N)$ Schritte für das Herstellen der Hierarchie der Triangulierungen.

Die Methode der hierarchischen Triangulierungen ist also ein effizientes Verfahren, um eine Suchstruktur über einer beliebig gegebenen Zerlegung eines Gebietes in Polygone zu konstruieren. Besteht die anfänglich gegebene Zerlegung aus insgesamt n Kanten, so kann die Suchstruktur der hierarchischen Triangulierungen in $O(n \log n)$ Zeit konstruiert werden; sie benötigt $O(n)$ Speicherplatz. Zu einem beliebigen Anfragepunkt kann dann das Polygon der ursprünglichen Zerlegung, in das der Anfragepunkt fällt, in Zeit $O(\log n)$ bestimmt werden.

7.8 Aufgaben

Aufgabe 7.1

Wir betrachten n Geraden in der Ebene, die in allgemeiner Position liegen sollen, d.h. keine drei Geraden schneiden sich in einem Punkt und keine Geraden sind parallel, vgl. das Beispiel in Abbildung 7.75.

a) Zeigen Sie, daß sich die Geraden in $\binom{n}{2}$ Punkten schneiden.

b) Zeigen Sie, daß die Geraden die Ebene in $\binom{n+1}{2} + 1$ Gebiete unterteilen. (Hinweis: Verwenden Sie eine imaginäre Scan-line, die von $x = -\infty$ bis $x = +\infty$ über die Geraden gleitet, als Zählhilfe. Betrachten Sie, wie sich die Anzahl der Gebiete bei Überquerung eines Schnittpunktes verändert. Sie können voraussetzen, daß es keine vertikalen Geraden gibt.)

c) Berechnen Sie die Anzahl der Kanten, d.h. der Liniensegmente zwischen zwei Schnittpunkten und der Halbgeraden, auf denen sich kein Schnittpunkt befindet, die sich durch die n Geraden ergeben.

Aufgabe 7.2

Geben Sie an, in welcher Reihenfolge die Schnittpunkte in der folgenden Menge von Liniensegmenten in der Ebene berichtet werden, wenn Sie eine Scan-line von links nach rechts über die Ebene schwenken (Abbildung 7.76).

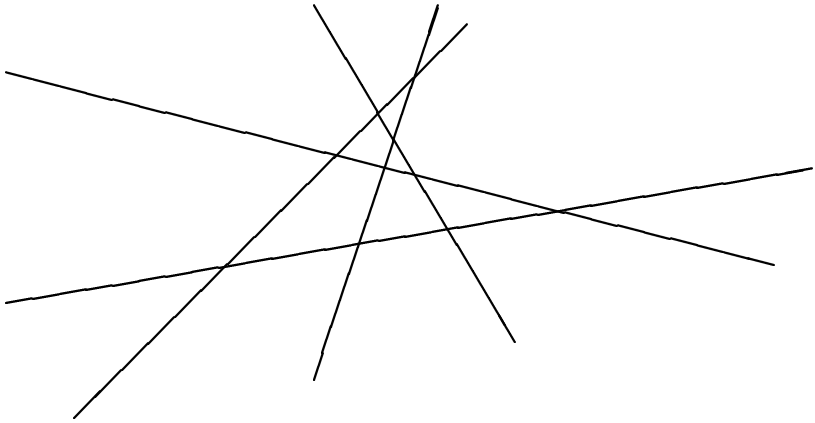


Abbildung 7.75: 5 Gerade zerteilen die Ebene in $\binom{6}{2} + 1 = 16$ Gebiete.

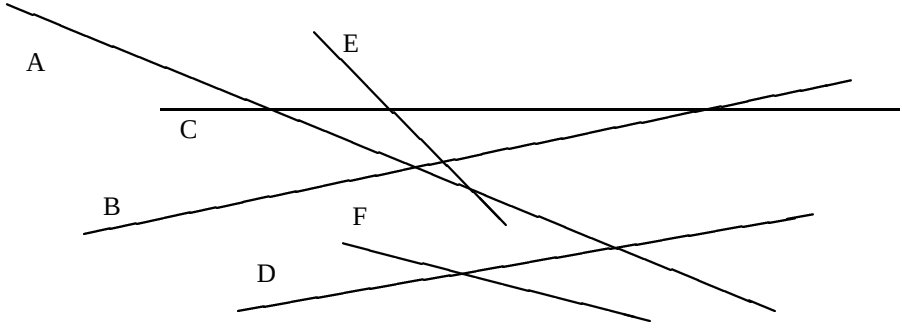


Abbildung 7.76

Aufgabe 7.3

Geben Sie ein Beispiel mit der kleinstmöglichen Anzahl von Liniensegmenten an, so daß der erste durch das Scan-line-Verfahren gefundene Schnittpunkt nicht der am weitesten links liegende ist.

Aufgabe 7.4

a) Warum kann ein Scan-line-Verfahren für ein Problem der Gröë n nie weniger als $cn \log n$ Schritte benötigen, für ein konstantes $c \in \mathbb{R}$?

b) Betrachten Sie das folgende, sogenannte *Element-uniqueness Problem*: Zu einer Zahlenfolge von n reellen Zahlen ist festzustellen, ob in der Folge zwei gleiche Zahlen vorkommen. Man kann zeigen, daß zur Lösung dieses Problems mindestens $\Omega(n \log n)$ Schritte benötigt werden.

Zeigen Sie, daß es mindestens ebenso schwierig ist festzustellen, ob sich n horizontale und n vertikale Liniensegmente schneiden. (Hinweis: Nehmen Sie an, Sie haben ein Verfahren für das Schnittproblem gegeben. Zeigen Sie, daß Sie durch eine geschickte Transformation das Element-uniqueness Problem lösen können. Sie können voraussetzen, daß auch einpunktige Liniensegmente zugelassen sind.)

Aufgabe 7.5

Wir betrachten n Punkte in der Ebene. Für zwei Punkte $x = (x_1, x_2)$ und $y = (y_1, y_2)$ sagen wir *x dominiert y*, falls $x_1 \geq y_1$ und $x_2 \geq y_2$. Ein Punkt ist *maximal*, wenn er von keinem anderen dominiert wird. Geben Sie ein möglichst effizientes Verfahren an, das alle maximalen Punkte berechnet.

Aufgabe 7.6

Wird eine Menge von Liniensegmenten, die wir uns als undurchsichtige Mauern vorstellen können, von einer punktförmigen Lichtquelle beschienen, so sind, im allgemeinen nur Teile der Segmente beleuchtet. Geben Sie ein möglichst effizientes Verfahren zur Berechnung der beleuchteten Segmentteile an und diskutieren Sie dessen Komplexität. (Hinweis: Verwenden Sie eine um die Lichtquelle rotierende Scan-line, wie in Abbildung 7.77 gezeigt.)

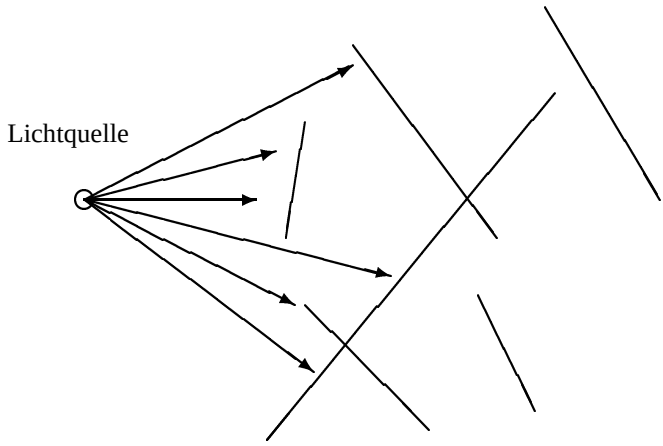


Abbildung 7.77: Ein Beispiel für eine Menge von Segmenten, die beleuchtet wird.

Aufgabe 7.7

Gegeben seien die horizontalen Segmente $A,\ldots,H$, die durch ihre linken und rechten Endpunkte repräsentiert sind, sowie die vertikalen Segmente $a,\ldots,g$. Durch (wiederholte) Aufteilung der Menge infolge rekursiver Aufrufe des Divide-and-conquer-Verfahrens zur Bestimmung aller Liniensegmentschnitte sind die in Abbildung 7.78 gezeigten Mengen S_1 und S_2 entstanden.

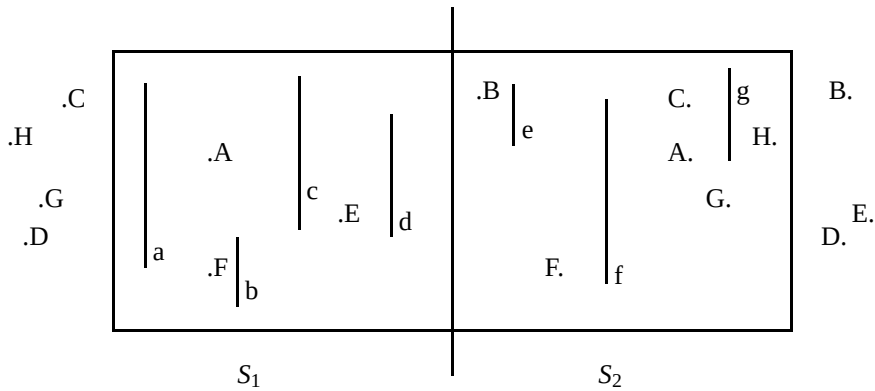


Abbildung 7.78

$.X$ bezeichnet den linken Endpunkt und $X.$ den rechten Endpunkt des Segments X. Geben Sie an, welche Segmentschnitte im Merge-Schritt (bei Vereinigung von S_1 und S_2) noch berichtet werden müssen.

Aufgabe 7.8

Wir betrachten eine Menge P von n Punkten in der Ebene. Die *konvexe Hülle* $conv(P)$ von P ist die kleinste konvexe Menge, die P enthält; $conv(P)$ ist offensichtlich ein konvexes Polygon. Geben Sie ein möglichst effizientes Verfahren zur Berechnung von $conv(P)$ an.

Aufgabe 7.9

Gegeben sei die Menge $\{A,B,C,D,E,F\}$ von Intervallen mit

$$A=[2,3], B=[5,9], C=[1,4], D=[3,7], E=[6,8] \text{ und } F=[8,10].$$

a) Geben Sie einen Intervallbaum möglichst geringer Höhe zur Speicherung dieser Intervallmenge an.

b) Führen Sie eine Aufspießanfrage für den Punkt $x=3$ durch und geben Sie an, in welcher Reihenfolge die aufgespießten Intervalle entdeckt werden (ausgehend vom Intervallbaum aus a)).

Aufgabe 7.10

Bei der im Abschnitt 7.4.2 vorgestellten Version von Segment-Bäumen waren die Knotenlisten nicht-sortierte, doppelt verkettete Listen von Intervallnamen. Zusätzlich wurde (um das Entfernen von Intervallnamen zu unterstützen) ein separates Wörterbuch für alle Intervalle aufrechterhalten.

Überlegen Sie sich eine andere, möglichst effiziente Möglichkeit zur Organisation der Knotenlisten, die es erlaubt, auf das zusätzliche Wörterbuch zu verzichten. Geben Sie eine möglichst genaue Abschätzung der Worst-case-Laufzeit der Einfüge- und Entferne-Operation in Ihrer Datenstruktur an.

Aufgabe 7.11

Entwerfen Sie einen möglichst effizienten Algorithmus zur Lösung des folgenden Problems:

Gegeben sei eine Menge von n Rechtecken in der Ebene. Es ist der Umfang der von den Rechtecken gebildeten Polygone zu bestimmen. Unter Umfang sei die Länge des Randes einschließlich des Randes der entstehenden Löcher verstanden, vgl. das Beispiel in Abbildung 7.79.

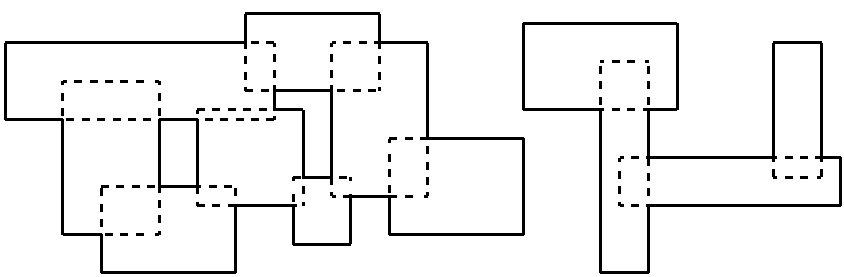


Abbildung 7.79: Die Länge der durchgezogenen Linie ist zu berechnen.

Formulieren Sie den Algorithmus in Pseudo-Pascal und geben Sie insbesondere Ihre Prozeduren zur Manipulation der verwendeten Datenstrukturen an. Analysieren Sie die Komplexität des von Ihnen verwendeten Verfahrens.

(Hinweis: Sie können davon ausgehen, daß die x-Koordinaten der vertikalen Seiten und die y-Koordinaten der horizontalen Seiten jeweils paarweise verschieden sind. Es ist ratsam, zwei Scan-line Durchläufe zu verwenden, womit man eine Laufzeit von $O(n \log n)$ erreichen kann.)

Aufgabe 7.12

Gegeben sie die folgende Menge von Punkten in der Ebene:
(3,7) (4,2) (5,8) (2,1) (1,4) (6,3) (7,9) (8,5)

a) Fügen Sie die Punkte der Reihe nach in das anfangs leere Skelett eines Prioritäts-Suchbaumes ein.

b) Bestimmen Sie die Menge aller Punkte im Bereich $3 \leq x \leq 6$ und $y \leq 5$ durch eine Bereichsanfrage im Prioritäts-Suchbaum.

c) Entfernen Sie die Punkte in der umgekehrten Reihenfolge aus dem Prioritäts-Suchbaum.

Aufgabe 7.13
Entwickeln Sie einen Einfügealgorithmus für einen balancierten Prioritäts-Suchbaum, der das Einfügen eines Punktes in logarithmischer Zeit ermöglicht. Verwenden Sie als zugrunde liegende Baumstruktur Rot-schwarz-Bäume.

Aufgabe 7.14
Ein Kantenzug C heißt monoton, falls jede horizontale Gerade C in höchstens einem Punkt schneidet. Ein Polygon P heißt monoton, falls der Rand von P in zwei monotone Kantenzüge zerlegt werden kann.

a) Wieviele Kantenschnitte können zwei Polygone mit n_1 und n_2 Kanten maximal miteinander haben? Wieviele sind es, falls beide Polygone monoton sind?

b) In wieviele Zickzacks zerfällt eine Menge von beliebigen Polygonen, monotonen Polygonen oder konvexen Polygonen, falls die Polygone insgesamt n Kanten haben?

Aufgabe 7.15
Zerlegen Sie die in Abbildung 7.80 dargestellte Menge von Polygonen in Zickzacks und bestimmen Sie die für einen Scan von oben nach unten geeignete Ordnung der Top-Segmente.

Aufgabe 7.16
Das *Slot-Assignment-Problem* für eine Menge horizontaler Liniensegmente in der Ebene ist folgendes Problem:

Finde die kleinste Zahl m (die minimale Slot-Anzahl) und eine Numerierung der Segmente mit "Slot-Nummern" aus $1, \ldots, m$ derart, daß gilt: Für jede vertikale Gerade, die irgendwelche horizontalen Segmente schneidet, sind die Slot-Nummern der geschnittenen Segmente längs der Geraden absteigend (aber nicht notwendigerweise lückenlos) sortiert.

a) Lösen Sie das Slot-Assignment-Problem für die Menge von Segmenten aus Abbildung 7.81.

b) Geben Sie ein allgemeines Verfahren zur Lösung des Slot-Assignment-Problems an und analysieren Sie die Laufzeit Ihres Verfahrens.

(Hinweis: Es ist möglich, das Slot-Assignment-Problem für eine Menge von n Segmenten in Zeit $O(n \log n)$ und Platz $O(n)$ zu lösen!)

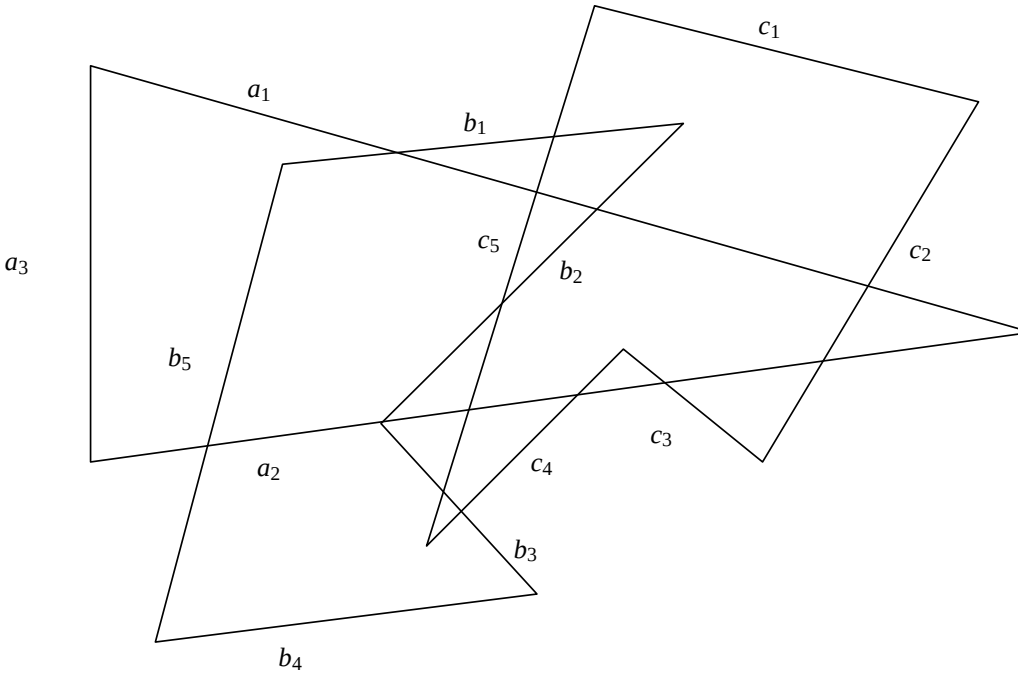


Abbildung 7.80

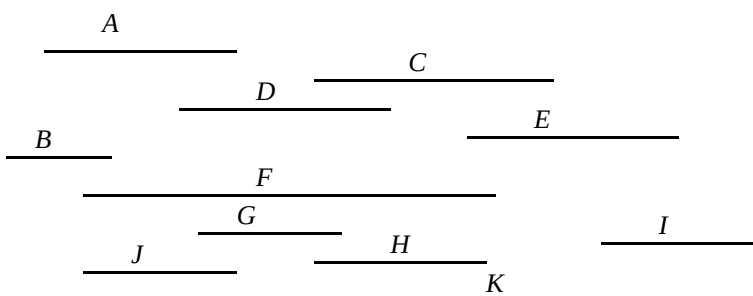


Abbildung 7.81

Aufgabe 7.17

Sei P ein Menge von n Punkten in der Ebene, von denen keine vier auf einem Kreis liegen, und $p_1, p_2, p_3 \in P$. Beweisen Sie, daß

a) das Dreieck mit den Eckpunkten p_1, p_2 und p_3 genau dann ein Teil der Delaunay-Triangulierung ist, wenn der Kreis durch p_1, p_2 und p_3 keine weiteren Punkte aus P enthält;

b) das Liniensegment von p_1 nach p_2 genau dann ein Teil der Delaunay-Triangulierung ist, wenn es einen Kreis K durch p_1 und p_2 gibt, der keine anderen Punkte von P enthält.

Aufgabe 7.18

Sei P ein Menge von n Punkten in der Ebene, von denen keine vier auf einem Kreis liegen. Der *Gabriel-Graph* $G(P)$ von P ist wie folgt definiert: Eine Kante $e = <p_1, p_2>$ mit $p_1, p_2 \in P$ gehört zu $G(P)$, falls für alle $p_3 \in P \setminus \{p_1, p_2\}$ gilt, daß

$$d^2(p_1, p_3) + d^2(p_2, p_3) > d^2(p_1, p_2)$$

ist, wobei d den euklidischen Abstand zwischen zwei Punkten bezeichnet.

a) Zeigen Sie, daß der minimale spannende Baum von P ein Teilgraph des Gabriel-Graphen ist.

b) Zeigen Sie, daß jede Kante des Gabriel-Graphen $G(P)$ auch eine Kante der Delaunay-Triangulierung $DT(P)$ ist (Hinweis: Beachten Sie Aufgabe 7.17).

c) Zeigen Sie, daß $e \in DT(P)$ genau dann eine Kante von $G(P)$ ist, falls e die Kante e' des Voronoi-Diagramms schneidet — wobei e' die zu e duale Kante ist (vgl. Abbildung 7.82).

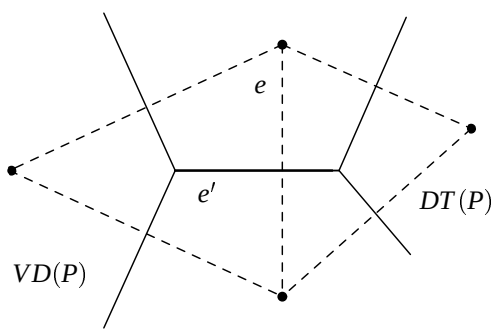


Abbildung 7.82: Eine Kante der Delaunay-Triangulation schneidet die duale Voronoi-Kante.

d) Geben Sie einen Algorithmus an, der in einer Zeit von $O(n)$ den Gabriel-Graphen berechnet, falls die Delaunay-Triangulierung gegeben ist.

Aufgabe 7.19
Geben Sie einen Algorithmus an, der zu einer gegebenen Menge von Punkten in linearer Zeit die konvexe Hülle berechnet, falls das Voronoi-Diagramm der Punkte schon vorliegt.

Aufgabe 7.20
Gegeben sei die Menge P von sieben Punkten in der Ebene, wie in Abbildung 7.83 dargestellt.

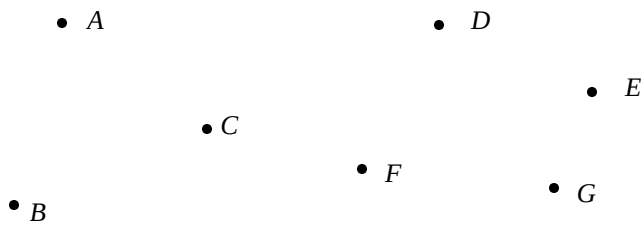


Abbildung 7.83

a) Konstruieren Sie das Voronoi-Diagramm für diese Punktmenge.

b) Geben Sie die Delaunay-Triangulierung von P an.

c) Geben Sie den Gabriel-Graphen und einen minimalen spannenden Baum für P an.

d) Geben Sie eine doubly connected edge list an, beschränkt auf alle Kanten der Voronoi-Regionen der Punkte A, B und C.

e) Zeigen Sie graphisch, wie man aus den Voronoi-Diagrammen für die Mengen $\{A,B,C\}$ und $\{D,E,F,G\}$ das Voronoi-Diagramm für die gesamte Punktmenge konstruieren kann (Merge-Schritt des Divide-and-conquer-Algorithmus).

Aufgabe 7.21
Entwerfen Sie einen Algorithmus, der für einen gegebenen Punkt und ein konvexes Polygon testet, ob der Punkt innerhalb oder außerhalb dieses Polygons liegt. Nehmen Sie an, daß die Eckpunkte des Polygons als ein nach Winkeln sortiertes Array vorliegen. Bestimmen Sie den Aufwand ihres Verfahrens. (Hinweis: Das Verfahren sollte nicht mehr als $O(\log n)$ Schritte benötigen, falls n die Anzahl der Kanten des Polygons ist.)