Datenstrukturen & Algorithmen 27. Mai 2015

Handout 13

Sebastian Millius, Sandro Feuz, Daniel Graf

Thema: Geometrische Algorithmen: Scanline, Range Trees, Anmerkung: Longest Increasing Subsequence

Referenz: Widmayer, Kapitel 7, Cormen et. al, Kapitel 33

Links

- Eine ausführliche Darstellung findet sich in *de Berg, M., Cheong, O., van Kreveld and M., Overmars, M.: Computational Geometry: Algorithms and Applications*. Ist (via VPN) auf Springerlink frei verfügbar
  http://springerlink.com/content/k18243/ (http://goo.gl/bMGLr)
  Belfrage eine Übersicht/Zusammenfassung der Darstellung über Range Trees zusammengestellt: http://www.belfrage.net/eth/d&a/pdf/rangetrees.pdf (http://goo.gl/UKcsI)
- Konkrete Implementationen von Geometrischen Algorithmen in C finden sich beispielsweise in *O'Rourke, J.: Computational Geometry in C*
  http://maven.smith.edu/~orourke/books/compgeom.html (http://goo.gl/6KCMM)
- Belfrage Slides: http://www.belfrage.net/eth/d&a/pdf/uebung6_h.pdf (http://goo.gl/Pvjei)
- Prof. Erickson Lecture Notes: http://goo.gl/OZYSK
- Scanline: Demo-Applet
  http://www.cs.princeton.edu/courses/archive/spring09/cos226/demo/gishur/ (http://goo.gl/tSKkU)

Scanline

Viele geometrische Probleme können mittels *Scanline* Verfahren gelöst werden. Die Idee ist, dass man (Halte-)Punkte der zu bearbeitende Daten in einer Dimension sortiert und mit einer Scanline in Richtung dieser Dimension über den gesamten Raum fährt. Dabei wird in einer Datenstruktur immer ein bestimmter Zustand der momentanen Situation um die Scanline herum gespeichert. Die Haltepunkte sind auch in einer Datenstruktur gespeichert (sortiertes Array, min- Heap, ...) und definieren die Stellen an welchen die Scanline anhalten muss, da sich dort etwas an der Situation ändert. An jedem Haltepunkt wird diese Änderung bearbeitet und die Scanline-Datenstruktur aktualisiert. Eventuell werden auch neue Haltepunkte gefunden und der Haltepunkte-Datenstruktur hinzugefügt. Während dem verschieben der Scanline wird auch die Lösung des Problems stückweise aufgebaut.

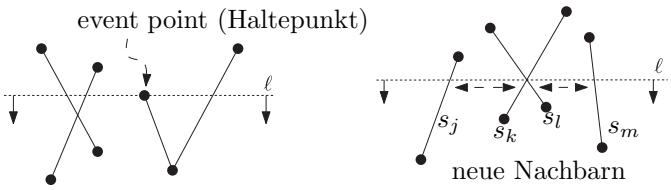
Anmerkung: Es gibt auch Situationen, in denen sich die Scanline nicht parallel fortbewegt, sondern z.B. als Halbgerade 360 Grad um einen Punkt herumgeht. In höheren Dimensionen kann man sich auch eine *scan plane* oder eine *scan hyperplane* vorstellen.

SWEEP/SCAN-LINE

Q = Haltepunkte // Haltepunkt Struktur, meist nach einem Kriterium sortiert
$L = \emptyset$ // Scan-Line Struktur, Menge der aktiven Objekte

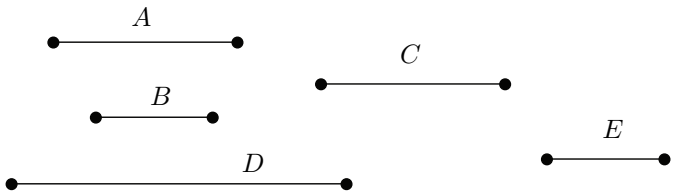
while $Q \neq \emptyset$
  Gehe zum nächsten Haltepunkt in Q
  AKTUALISIERE(L)
  Gebe Teillösungen aus // falls vorhanden

Beispielsweise sind beim Segment Schnitt Problem die Haltepunkte die Anfangs-, End- und Schnittpunkte von Segmenten.



Beispiel: Sichtbarkeitsproblem

Zwei Liniensegmente s_1 und s_2 in einer gegebenen Menge horizontaler Liniensegmente sind gegenseitig sichtbar, wenn es eine vertikale Gerade gibt, die s_1 und s_2 schneidet, aber kein weiteres Segment zwischen s_1 und s_2 liegend.



Wie können wir alle Paare von Segmenten ausgeben die sich gegenseitig sehen? (Think about it!) Im obigen Beispiel also die Paare $\{A, B\}, \{A, D\}, \{B, D\}, \{C, D\}$.

Allgemeine Idee: Scanline. Alle Paare von Segmenten die sich gegenseitig sehen sind zu einem Zeitpunkt direkte Nachbarn in der y-Ordnung.

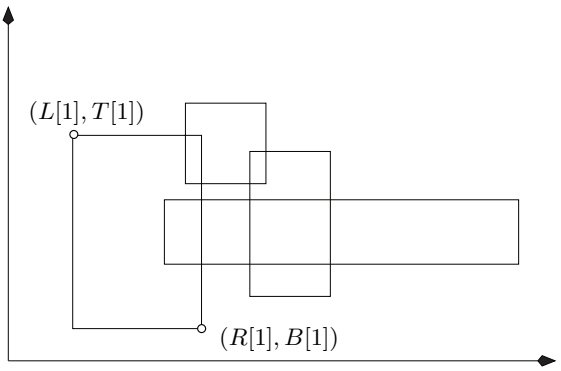
Sichtbarkeit(S)

// Scanline Algorithmus
Q = Folge der $2n$ Anfangs- und Endpunkte von
Elementen aus der Menge S von Liniensegmenten in aufsteigender x-Reihenfolge
$L = \emptyset$ // Menge der aktiven Segmente

while $Q \neq \emptyset$
  p = SELECT_AND_REMOVE(Q)
  bestimme Nachbarn s_1 und s_2 von Segment s
  if p ist linker Endpunkt von s
    füge s in L ein
    gib (s, s_1) und (s, s_2) aus
  else entferne s aus L
    gib (s_1, s_2) aus

Zusatzübungen

1. Gegeben seien n Rechtecke in der Ebene, jedes spezifiziert durch eine linke-obere $(L[i], T[i])$ und eine rechte untere Koordinate $(R[i], B[i])$.



   Gegeben sind also die vier Arrays $L[1 \ldots n], R[1 \ldots n], T[1 \ldots n]$ und $B[1 \ldots n]$

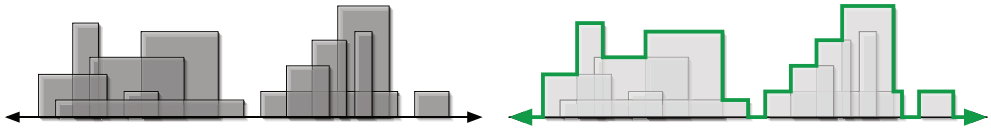
   a) Beschreibe und analysiere einen Algorithmus der bestimmt ob sich irgendwelche zwei Rechtecke schneiden. (in $\mathcal{O}(n \log n)$)

b) Beschreibe und analysiere einen Algorithmus der einen Punkt P bestimmt, der in den meisten Rechtecken liegt (in $\mathcal{O}(n \log n)$)

2. Beschreibe und analysiere einen Scanline Algorithmus der bestimmt, ob es in einer Menge von Kreisen zwei gibt, die sich schneiden (in $\mathcal{O}(n \log n)$). Jeder Kreis ist spezifiziert durch die Koordinaten seines Zentrums (x_i, y_i) und seinen Radius r_i.

3. Beschreibe und analysiere einen Scanline Algorithmus, der die Skyline der folgenden Städte bestimmt. D.h. gebe die Eckpunkte der Skyline aus, in $\mathcal{O}(n \log n)$, wobei n die Anzahl der Häuser ist.

   a) Manhattan: Jedes Gebäude ist ein Rechteck dessen untere Kante auf der x-Achse zu liegen kommt, und ist spezifiziert durch eine linke und rechte x-Koordinate sowie dessen Höhe.

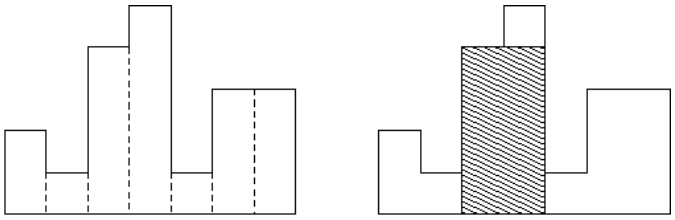


Die Manhattan Skyline

   b) Gizah: Jedes Gebäude ist ein gleichschenkliges rechtwinkliges Dreieck, dessen Hypothenuse auf der x-Achse zu liegen kommt, und ist spezifiziert durch die (x, y) Koordinate seiner Spitze.

Ägyptische Skyline

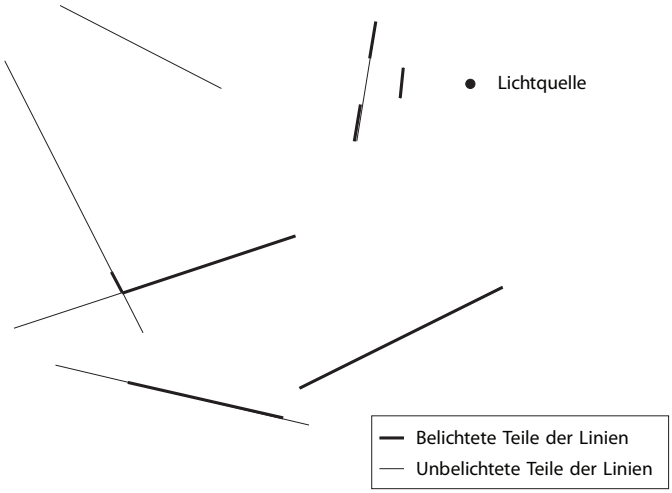
4. Ein Histogramm ist ein Polygon aus einer Folge von Rechtecken, die auf einer gemeinsamen Grundlinie ausgerichtet sind. Die Rechtecke haben alle die gleiche Breite könn aber unterschiedliche Höhen haben. Zum Beispiel zeigt folgende Abbildung (links) ein Histogramm, dass aus Rechtecken besteht mit den Höhen 2, 1, 4, 5, 1, 3, 3, in Einheiten gemessen, wobei 1 die Breite der Rechtecke ist:



   Normalerweise werden Histogramme verwendet, um diskrete Verteilungen darstellen, z. B. die Häufigkeiten von Zeichen in Texten. Gesucht ist die Fläche des grössten Rechtecks in einem Histogramm, dass an den gemeinsamen Grundlinie ausgerichtet ist. Die Abbildung rechts zeigt dieses Rechteck. Gegeben n und die Höhen der Rechtecke eines Histogramms in dieser Reihenfolge $h[1], \ldots, h[n]$, bestimme die Fläche des grösstten Rechtecks in diesem Histogramm.

   (möglich in $\mathcal{O}(n)$)

5. Gegeben sei eine Menge von n Liniensegmenten mit beliebiger Orientierung in der Ebene, in allgemeiner Lage. Sei weiter eine punktförmige Lichtquelle gegeben, die die Ebene belichtet. Jedes Liniensegment sei undurchsichtig und schirme das Licht ab. Gebe ein Verfahren an, das möglichst e?zient feststellt, welche Teile jedes Segments belichtet sind. Speicherbedarf, Laufzeit? Welche Datenstrukturen?



Range Tree (d-dimensional)

Range Trees sind Bäume, die es ermöglichen, Bereichsanfragen im d-dimensionalen Raum effizient zu beantworten.

Für $d = 1$ sind das gewöhnliche (balancierte) Binäre Bäume, deren Blätter meist verkettet sind (Blattsuchbaum). Dadurch kann man die Liste durchgehen, um alle Elemente im Bereich zu ermitteln.

Für höhere Dimensionen lässt sich diese Idee verallgemeinern. Der Baum hat die x-Koordinate als Schlüssel. Jeder innere Knoten k zeigt auf einen y-Baum (y-Koordinate dient als Schlüssel), in dem alle im Teilbaum von k liegenden Werte enthalten sind. Für die dritte Dimension zeigt wiederum jeder innere Knoten jedes y-Baumes auf einen z-Baum. Dies kann in beliebige Dimensionen getrieben werden.

Analyse

Query

$\mathcal{O}(\log^d n + k)$, k: inspizierte Elemente ($\neq$ Anzahl Elemente im Bereich!)

Jede Dimension benötigt $\mathcal{O}(\log n)$ für die Suche in jeweils einem Baum, und es werden jeweils auch $\mathcal{O}(\log n)$ Bäume der nächsten Dimension durchsucht.

Speicherplatz

$\mathcal{O}(n \cdot \log^{d-1} n)$

Baum der 1. Dimension (x-Baum) benötigt $\mathcal{O}(n)$ Speicherplatz. Für jedes Level dieses Baumes kommt jedes Element (also insgesamt n Elemente) in den entsprechenden y-Bäumen genau einmal vor. Daher beanspruchen alle y-Bäume zusammen $\mathcal{O}(n \log n)$ Speicherplatz (es gibt $\mathcal{O}(\log n)$ viele Levels).

Nun kann man dies wiederum rekursiv anwenden. Wir wissen also, dass ein $2 - d$ Range-Tree $\mathcal{O}(n \log n)$ Speicherplatz benötigt.

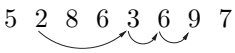
Bei einer dritten Dimension hat man wiederum eine obere Limite von $\mathcal{O}(n \log n)$ pro Level was zu $\mathcal{O}(n \log^2 n)$ führt, usw.

Wichtig: Range-Trees sind statisch, Dynamik (insert/remove) kriegt man nicht einfach so hin und muss dafür andere Ansätze verfolgen

Longest Increasing Subsequence

Problem: Gegeben ist ein Array A von Zahlen $A[1], ..., A[n]$. Eine *Subsequence* ist eine Teilmenge dieser Zahlen $A[i_1], A[i_2], ..., A[i_k]$ mit $1 \leqslant i_1 < i_2 < ... < i_k \leqslant n$.

Eine *Increasing Subsequence* ist eine Sequenz in der die Zahlen strikt grösser werden. z.B. die längste aufsteigende Sequenz in 5, 2, 8, 6, 3, 6, 9, 7 ist 2, 3, 6, 9



Wir suchen nun die längste aufsteigende Sequenz in einem Array.

1. Struktur

Denken wir darüber nach, was eine optimale Lösung charakterisiert. Betrachten wir das letzte Element (sei dies das Element i im Array, die 9 im obigen Beispiel) in einer längsten aufsteigenden Sequenz. Das vorletzte Element in der Sequenz (die 6 im obigen Beispiel) ist eines, dass im Array vor i kommt, und der Teil der optimalen Lösung ohne das letzte Element ist *eine längste aufsteigende Sequenz*, die beim vorletzten Element in der Sequenz endet.

2. Rekursion

Versuchen wir obige Überlegungen zusammenzufassen.

Sei $lis(i)$ die längste aufsteigende Sequenz die beim Element $A[i]$ endet.

Die Lösung wäre also die längste davon, d.h. $\max_{i \in \{1,..,n\}} lis(i)$ (wobei das i, welches das maximum erreicht, gerade das letzte Element der Lösung ist).

Formuliere $lis(i)$ rekursiv:

$lis(i) = 1 + \min_{j \in \{1,..,i-1\} \text{ mit } A[j]<A[i]} lis(j)$

Wir gehen also alle Elemente vor dem Element i durch und wählen von denen die kleiner als das i-te Element sind, dasjenige aus, bei dem die längste bisherige aufsteigende Sequenz endet. Daraus können wir eine um eins längere aufsteigende Sequenz bis zum i-ten Element bilden (wir hängen das i-te Element einfach hinten dran).

3. Bottom-Up

LIS(A, n)
  // Berechne die Länge der longest increasing subsequence

  for $i = 1$ **to** n
    $lis[i] = 1$ // Finde längste Sequenz mit Schlusselement i
    $pre[i] = -1$ // Vorgänger in der Sequenz
    for $j = 1$ **to** $i - 1$
      if $A[j] < A[i]$ and $lis[j] + 1 > lis[i]$
        $lis[i] = lis[j] + 1$
        $pre[i] = j$

  return $\max_i lis[i]$

Die Laufzeit ist (*think about it!*) $O(n^2)$

4. Lösung berechnen

Das letzte Element der Lösung ist gerade dasjenige mit dem grössten $lis[]$ Wert, also $\arg\max_i lis[i]$. Es ist dann sehr einfach die Sequenz zu rekonstruieren, wenn wir das Feld $pre[i]$ nutzen, in welchem jeweils der direkte Vorgänger von i in der längsten Sequenz mit Endelement i steht.

LIS2(lis, pre, n)

// Berechne eine Instanz der longest increasing subsequence

$path =$""
$i = \arg\max_i\{lis[i]\}$
while $i \neq -1$
    $path = i+","+path$ // i vorne an den Pfad hängen
    $i = pre[i]$

return $path$

Lösung in $\mathcal{O}(n \log n)$

Der Algorithmus kann auf eine Laufzeit von $\mathcal{O}(n \log n)$ verbessert werden mit Hilfe von folgenden Beobachtungen: Es sei t^i_j der kleinste Wert bei dem eine Increasing Subsequence der Länge j endet auf der Sequenz $A[1], \ldots, A[i]$.

Beobachtung: Für jedes i gilt, dass $t^i_1 < t^i_2 < \ldots < t^i_j$, d.h. diese Werte liegen immer sortiert vor! Dies legt nahe, dass wenn wir die längste aufsteigende Teilsequenz finden möchten, die mit $A[i]$ endet, wir nach dem j suchen müssen, so dass $t^{i-1}_j < A[i] < t^{i-1}_{j+1}$ ist! Die Länge der längsten aufsteigenden Teilsequenz, die bei $A[i]$ endet, ist damit $j + 1$.

Dann ist $t^i_{j+1} = A[i]$ und $t^i_k = t^{i-1}_k$ für alle $k \neq j + 1$. Da die t^i_j immer in sortierter Reihenfolge vorliegen, und eine Änderung die Sortierheit bewahrt, kann in jedem Schritt eine binäre Suche durchgeführt werden, um j zu bestimmen.

Die Laufzeit ist damit $\mathcal{O}(n \log n)$ (vgl. Musterlösung zur Programmieraufgabe).

Siehe http://www.algorithmist.com/index.php/Longest_Increasing_Subsequence (http://goo.gl/1s87F) für eine Implementierung in C++.