Computational Geometry

Alexander Gran, Gunter Spöcker

Proseminar: Datenstrukturen und Algorithmen SS 2005

1 Formalismus

Die Punkte liegen für gehwöhnlich als als Koordinatentupel (x,y). Linien typischerweise als Anfangs- und Endpunkt. Alternativ auch als Vektoren, Gerade etc. Die Definition der Linie über eine convexe Hülle der beiden Endpunkte findet sich auch. Mit Abstand ist hier der euklidische Abstand gemeint ($\sqrt{(x_1 - x_2)^2 - (y_1 - y_2)^2}$).

2 Closest Pair

2.1 Worum geht es?

Gegeben ist eine Menge vom Punkten. Aus dieser Menge soll dasjenige Paar vom Punkten gefunden werden, dass den kleinsten Abstand voneinander hat. Eine wichtige Anwendung des "Closest Pair"-Problems ist die der Kollisionserkennung zum Beispiel bei Flugzeugen.

2.2 Naiver Algorithmus

Dieser Algorithmus funktioniert genau so, wie man es direkt vermuten wird. Er testet alle Kombinationen von Punkten durch und merkt sich dabei den aktuell minimalen Abstand (und die dazugehörigen Punkte).

Laufzeitanalyse: Es ist unmittelbar ersichtlich, dass dieser Algorithmus eine Laufzeitkomplexität von $O(n^2)$ hat und ist damit für praktische Anwendungen nicht geeignet.

2.3 Divide-And-Conquer Algorithmus

Zunächst sortieren wir die Menge der Punkte sowohl in der x- als auch in der y-Richtung (nachfolgend werden diese zwei Mengen X und Y genannt).

2.3.1 Divide:

Sollte einem Funktionsaufruf maximal 3 Punkte übergeben werden, wird der naive Algorithmus darauf angewendet. Andernfalls werden die Mengen X und Y in jeweils 2 Mengen aufgeteilt, deren Punkte links, bzw. rechts einer imaginären, vertikalen Linie liegen. Dazu werden die zwei benachbarten Punkte in der Mitte der Menge X bestimmt und die imaginäre Linie dazwischen gelegt. Mit den zwei links der Linie liegenden Mengen wird ein neuer Rekursionsschritt gestartet; gleiches gilt für die rechten Mengen. Anschließend werden die beiden Teilergebnisse, X und Y an die Funktion "conquer" übergeben und zu einem Ergebnis zusammengefasst.

2.3.2 Conquer:

Sei "min" der kleinere Wert der beiden Teilergebnisse. Da bisher nur alle Punkte die links-, bzw. rechts der Linie liegen untereinander getestet wurden, könnte es sein, dass, im Abstand "min" zu jeder Seite der Linie, ein Paar von Punkten mit einer kleineren Distanz als "min" existiert. Um hier nicht alle Punkte wieder gegeneinander testen zu müssen, wird eine weitere Menge erstellt, deren Punkte alle im Abstand von maximal "min" zur Linie liegen (diese Menge muss auch in y-Richtung sortiert sein, deswegen wird diese Teilmenge aus Y gewonnen um nicht neu sortieren zu müssen). Da uns nur noch Abstände kleiner als "min" interessieren reicht es von jedem Punkt in dieser Menge nur die nächsten 7 auf deren Abstand zu diesem zu testen. Dies ist ausreichend (vergleiche Bild): Da auf jeder Seite der Linie die Punkte mindestens "min"-weit auseinander sind, können maximal 2 Punkte pro Punkte pro Seite nebeneinander (mit dem selben y-Wert) existieren. Somit kann es in dieser Menge maximal 4 Punkte nebeneinander geben (mit der Annahme, dass die zwei mittleren den selben x-Wert haben). Untereinander kann es auf jeder Seite auch nur maximal 2 Punkte geben. Sollte hierbei ein kleinerer Wert als "min" ermittelt werden wird dieser als Ergebnis zurückgeliefert.

Laufzeitanalyse: Hauptsächlich wird die Laufzeitkomplexität durch das Sortieren der Anfangsmenge bestimmt. Wie aus Datenstrukturen und Algorithmen bekannt ist, ist dies bestenfalls in $O(n \log n)$ möglich.

2.4 Sweep Algorithmus

Hierbei wird die Menge von Punkten nur in x-Richtung sortiert und eine imaginäre, vertikale Linie von links nach rechts durch die Menge geschoben (Sweep). Der Algorithmus benötigt einen autobalancierenden Suchbaum ((2,4)-Baum). Durchfährt die Linie einen Punkt, verfährt der Algorithmus wie folgt: Dieser Punkt wird in den Baum (in y-Richtung sortiert) eingefügt. Anschließend werden alle Punkte aus dem Baum gelöscht, die weiter als das aktuell bestimmte Minimum von der Linie weg sind, dies kann über die sortierte Menge sehr schnell erledigt werden. Danach werden die 6 benachbarten Punkte zum Aktuellen mit Hilfe des Baumes bestimmt und diese auf Ihren Abstand zum momentan betrachteten getestet. Es müssen, aus ähnlichen Gründen wie oben genannt, nur 6 Punkte untersucht werden. Sollte dabei ein kleiner Abstand ermittelt werden, wird dieser übernommen und die zugehörigen Punkte gemerkt.
Ein autobalancierender Suchbaum wird benötigt um die Operationen "insert", "delete" und "search" in $\log n$ zu realisieren.

Laufzeitanalyse: Da jeder Punkt einmal in den Baum eingefügt und wieder gelöscht wird, hat dieser Algorithmus eine Laufzeitkomplexität von $O(n \log n)$.

2.5 Vergleich der Laufzeiten

Diese Ergebnisse sollen einen ungefähren Eindruck geben wie Laufzeiten der Algorithmen aussehen. Sie wurden mit dem angegebenen (Fußnote dieser Seite) Quellcode ermittelt. Natürlich lässt sich noch das ein oder andere im Quellcode optimieren.

2.6 Ausblick

Offensichtlich ist der Naive Algorithmus wegen seiner Laufzeitkomplexität nur für sehr kleine Mengen verwendbar. Bei realistischen Datenvolumen im Bereich von $10^3 - 10^6$ Punkten kommen nur der Sweeping und der Divide and Conquer Algorithmus in Betracht. Doch obwohl Beide in $O(n \log n)$ liegen gibt es hier Unterschiede: Divide and Conquer ist um einiges langsamer als Sweep, u.A. da er am Anfang die Punkte doppelt sortieren muss, wohingegen der Sweep im Normalfall im Baum geringeren Sortieraufwand hat.

3 Line Segments

3.1 Worum geht es?

Gegeben ist eine Menge von n Linien. Gesucht sind die Linien, die sich schneiden.
Anwendungen findet "Linesegement Intersection" unter anderem bei Landkarten, wenn mehrere verschiedenen "aufeinandergelegt" werden sollen, um neue Informationen zu generieren. Auch die Frage, ob sich 2 Polygone schneiden wird damit gelößt, indem einfach beim Schnitttest noch nach Herkunft der Linie unterschieden wird (Die Laufzeit bleibt gleich).

3.2 Der Naive Algorithmus

Am einfachsten ist sicherlich jede Linie mit allen anderen zu vergleichen. Dabei speichert man alle Linienpaare, die sich schneiden (und verwirft Duplikate). Dazu wird zunächst geprüft, ob die beiden Rechtecke, die die Liniensegmente umgeben, sich schneiden. Wenn dem so ist, wird überprüft, ob der erste Punkt der ersten Linie links bzw. rechts von der anderen liegt, das selbe mit dem zweiten Punkt. Liegen die beiden Punkte auf verschiedenen Seiten, schneiden sich die Linien. Wir nennen diesen Algorithmus SegmentsNaiv.

Laufzeitanalyse: SegmentsNaiv liegt in $O(n^2)$, was klar ist, da n Linien mit n-1 verglichen werden müssen. Der Vergleich erfolgt recht einfach in $O(1)$ Für typische Applikationen mit $10^3 - 10^6$ Linien ist das viel zu langsam.

3.3 Der Sweeping Algorithmus

3.3.1 Sweeping Idee

Die Idee des Sweeping[1] ist einfach: Je weiter die Linien von einander entfernt sind, desto unwahrscheinlicher ist es, dass sie sich schneiden. Eine erste Anwendung dieser Idee haben wir schon beim einfachen Test auf Schnitt von zwei Liniensegmenten gesehen, hier werden zunächst die umgebenden Rechtecke geprüft.

Problem: Wie definiert man Entfernung bei Linien, und wie nutzt man das.

Vorgehen: Betrachten des x-Abstands, und sortieren der Punkte nach diesem.

Wenn die Punkte sortiert vorliegen, kann man darübersweepen, d.h. sie von links nach rechts durchfegen. Dabei versucht man, immer möglichst wenig Linien zu betrachten, um auch möglichst wenig Prüfungen auf Schnitt zu machen.

3.3.2 Sweeping Implementierung

Der Algorithmus "wandert" also mit einer gedachten (Sweep-)Linie von links nach rechts über die Punkte. Auf dem Bild sieht man einige der sogenannten Ereignis-Punkte an denen die Sweepline anhält.

Dabei speichert er die (y) Reihenfolge der Linien, die sich mit der Sweeplinie schneiden. Zur Speicherung wird ein selbstbalancierender Suchbaum verwendet. Es ist klar, das sich daran nur dann etwas ändert, wenn

1. Die Sweeplinie einen Startpunkt passiert
2. Die Sweeplinie einen Endpunkt passiert
3. Die Sweeplinie einen Schnittpunkt passiert

Also muss nur an diesen Punkten angehalten werden, wir nennen sie Ereignis-Punkte. Da 3. aber nicht direkt berechnet werden kann - das ist ja Ziel unseres Algorithmus - muss ein Weg gefunden werden, dies bei 1. und 2. zu erkennen.

Bei 1. wird nun das jeweilige Liniensegment in den Baum eingefügt, bei 2. wird es wieder entfernt.

Außerdem muss jedesmal überprüft werden, ob sich etwaige "neue Nachbarn" schneiden. Wichtig ist, dass nicht alle Linien im Baum überprüft werden, sondern nur die Linien, die einen neuen Nachbarn bekommen haben. Wird bei 1. ein Schnittpunkt gefunden, so wird er in die X-Ordung als zusätzlicher Ereignis-Punkt einsortiert, damit dort angehalten werden kann (Die Y-Ordnung im Baum ändert sich dort). Außerdem werden die jeweiligen Linien natürlich als schneidend ausgegeben.

Laufzeitanalyse: Mehrere Dinge beinflussen die Laufzeit:

- Sortieren der Punkte: Dies geht bekanntermaßen in $O(n * \log(n))$

[1] engl. fegend, sausend

- Einfügen, Entfernen und Finden der Nachbarn in einer sortierten Struktur. Das geschieht am Besten in einem sortierten autobalancierenden Baum, der obige Operationen in $O(\log(n))$ schafft. Es bieten sich z.B. AVL Bäume an, wir verwenden hier einen (2,4)-Baum.
- Prüfung auf Schnitt von 2 Liniensegmenten Dies geht in $O(1)$.

  Es egibt sich damit die Gesammtlaufzeit von $O(n \log(n))$:
  Es werden an jedem der n Punkte folgende Operationen ausgeführt:

  1* Einfügen oder Löschen $\rightarrow O(\log(n))$

  1* Nachbarn finden$\rightarrow O(\log(n))$

  2* Schnittprüfung $\rightarrow O(1)$

  Ergibt bei n Punkten also $O(n \log(n))$ als Gesammtlaufzeit. Bei beispielsweise 10^6 Linien ist das - unter Vernachlässigung von Konstanten - ca. 72000 mal schneller als der naive Algorithmus. Allerdings sein angemerkt, das jeder Algorithmus, der alle sich schneidenden Linien ausgibt, in $O(n^2)$ läuft, da maximal $(n^2)/2$ ausgegeben werden müssen.

3.4 Ausblick

Wenn k die Anzahl der gefundenen sich schneideden Kanten sind, ist der Algorithmus:

- inputdeterminiert in $O(n \log(n))$
- outputdeterminiert in $O(k)$ bzw. $O(n^2)$
- platzdeterminiert in $O(2n + k)$ bzw. $O(n^2)$

In der Regel ist aber $k << n^2$.Es ist zur Zeit kein besserer Algorithmus bekannt. Höherdimensionale Berechnungen sind auch möglich: Im 3-dimensionalen Raum wird beispielsweise eine Sweep-Ebene verwendet.

4 Convex Hull

4.1 Was ist eine konvexe Hülle?

Es sei eine Menge von Punkten gegeben. Im 2-dimensionalen Raum ist eine konvexe Hülle ein minimales Ploygon (minimal bzgl. der Ecken), dessen Ecken aus Punkten der Menge bestehen, so dass alle anderen Punkte innerhalb des Vielecks liegen. In höheren Dimensionen ist stattdessen eine n-dimensionale Figur, die ebenfalls alle Punkte enthält.

Bildlich kann man dies auch so verstehen: Aus einem Brett schauen einige Nägel
ein Stück weit heraus; spannt man nun ein Gummiband um all diese Nägel, bildet dieses die konvexe Hülle der Nägel ab.
Ein sehr wichtiges Anwendungsfeld der konvexen Hülle ist zum Beispiel die Texterkennung

4.2 Naiver Algorithmus

Dieser Algorithmus nimmt sich ein geordnetes Paar von Punkten und überprüft,
ob alle anderen Punkte der Menge rechts von der Linie liegen, die durch die beiden ersteren Punkte gebildet wird (rechts: wenn man am ersten Punkt steht und Richtung Zweitem schaut, dann muss der Dritte rechts davon liegen). Falls dies so ist, merkt sich der Algorithmus dieses Paar. Nachdem alle Paare überprüft wurden, wird eine Ergebnisliste der Punkte der Hülle (im Uhrzeigersinn, beginnend bei einem beliebigen Punkt) nach folgendem Schema erstellt:
Beide Punkte des ersten gespeichertes Paares werden in die Ergenisliste eingefügt, danach wird dieses Tupel aus der Merkliste gelöscht. Anschließend wird, solange noch Paare in der Merkliste sind, das Tupel, dessen erster Punkt der letzte in der Ergebnisliste ist, gesucht, der zweite Punkt an die Eregnisliste angefügt und danach das Tupel gelöscht.
Ein sehr großes Problem dieses Algorithmus ist es, dass bei der Überprüfung ob ein Punkt rechts der Linie liegt, Rundungsfehler auftreten können. Diese können dazu führen, dass die berechnete konvexe Hülle Löcher aufweist oder sogar Mehrfachkanten enthält.

Laufzeitanalyse: Da zu jedem Punktpaar alle anderen Punkte getestet werden müssen, hat dieser Algorithmus eine Laufzeitkomplexität von $O(n^3)$.

4.3 Inkrementeller Algorithmus

Zunächst wird die Menge der Punkte in x-Richtung sortiert (sollten zwei Punkte
den gleichen x-Wert haben, werden diese nach y sortiert); im folgenden X genannt. Dieser Algorithmus erstellt zuerst die "obere" Hälfte der Hülle und anschließend in einem zweiten Schritt die "untere".
Sei O das Ergebnisarray für die obere Hülle: Es werden nun nach und nach alle Punkte aus X an O angehängt. Ab dem dritten eingefügten Punkt, wird O auf Korrektheit getestet:
Dazu wird untersucht, ob die letzten drei Punkte aus O einen "Knick nach rechts" beschreiben. Sollte dies so sein, muss nichts weiter getan werden und der nächste Punkt kann angehängt werden. Ist dies nicht der Fall (dies gilt auch wenn die 3 Punkte auf einer Linie liegen), so muss der zweitletzte Punkt aus O wieder entfernt werden und anschließend erneut die Korrektheit von O auf die gleiche Weise überprüft werden.
Ist man nun beim letzten Punkt aus X angelangt, sind in O alle Punkte der "oberen" konvexen Hülle enthalten. Als nächstes muss das gleiche nochmal für die "untere" Hülle durchgeführt werden, dies geschieht jetzt allerding von rechts nach links (also beginnend beim letzten Punkt in X). Angenommen die Punkte der "unteren" Hülle wurden in dem Array U gespeichert, dann werden nun O und U wie folgt vereinigt:
Man übernimmt O komplett und hängt U an, allerdings wird aus U zuvor der erste und der letzte Punkt entfernt.
Bei diesem Algorithmus sind Rundungsfehler nicht weiter tragisch, da die Hülle auf jeden Fall geschlossen ist. Die kleinen Abweichungen in der Hülle, die dadurch entstehen können, dass z.B. ein Punkt zuviel aufgenommen wurde, sind in der Praxis meistens nicht von Bedeutung.

Laufzeitanalyse: Hauptsächlich wird die Laufzeitkomplexität durch das Sortieren der Anfangsmenge bestimmt, denn das eigentliche Finden der konvexen Hülle kann in $O(n)$ erledigt werden. Somit resultiert eine Komplexität von $O(n \log n)$.

4.4 Konvexe Hülle im 3-Dimensionalen - Randomisierter Inkrementeller Alg.

Der Algorithmus sucht sich 4 Punkte aus der Eingabemenge, die nicht alle in einer Ebene liegen.
Im Detail: Er nimmt die ersten beiden Punkte, sucht sich einen dritten, der nicht auf einer Linie mit diesen liegt, und sucht danach nach einem vierten, der nicht in der Ebene der ersten 3 liegt. Sollten keine solchen vier Punkte gefunden werden können, liegen alle Punkte der Menge in einer Ebene und man kann den oben vorgestellten Algorithmus verwenden. Andernfalls verfährt man wie folgt:

Man generiert eine zufällig gewählt Permutation $p_5, \ldots, p_n$ aus den verbliebenen Punkten. Nun muss überprüft werden, ob p_i mit $i \geq 5$ zur konvexen Hülle gehört oder nicht, dazu wird getestet, ob p_i innerhalb des bereits bestehenden Polytops ("3-dimensionales Polygon") liegt. Falls ja, dann ist nichts weiter zu tun und p_i kann verworfen werden, falls nicht, dann werden alle Punkte des Polytops wieder herausgenommen, die, wenn man von p_i aus auf den Horizont des Polytops schaut, auf der Vorderseite liegen und nicht zum Horizont gehören. Als nächstes wird p_i dem Polytop hinzugefügt.

Laufzeitanalyse: Die Laufzeitkomplexität zum Finden der 3-dimensionalen konvexen Hülle ist $O(n \log n)$.

4.5 Ausblick

Offensichtlich ist der Naive Algorithmus wegen seiner Laufzeitkomplexität nur für sehr kleine Mengen verwendbar. Bei realistischen Datenvolumen im Bereich von $10^3 - 10^6$ Punkten kommt von den hier vorgestellten Algorithmen nur der inkrementelle in Frage. Wie man hier leicht sehen kann beträgt die Laufzeitkomplexität für die Berechnung der 3-dimensionalen konvexen Hülle auch nur $O(n \log n)$, dies hätte man wohl im ersten Moment nicht gedacht.

Literatur

[Cormen, Leiserson, Rivest, Stein:] Introduction to Algorithms. MIT Press 2001.

[Preparata, Shamos:] Computational Geometry: An Indroduction. Springer-Verlag, 1985.

[Sam Bakhtiar] `http://www.cs.mcgill.ca/~cs251/ClosestPair/ClosestPairPS.html`

Den Sourcecode der Algorithmen gibt es online unter http://zodiac.dnsalias.org/misc/ProSemDastru.tar.gz