Praktikum Algorithmische Anwendungen WS 2006/07

Algorithmische Geometrie – Konvexe Hülle von Punkten

Gruppe C_blau_Ala0607

Tanja Neubacher, 11042463, tanja.neubacher@web.de
André Wolf, 11042424, ai731@gm.fh-koeln.de
Aysel Yildiz, 11041004, ai822@gm.fh-koeln.de

25.01.2007

Inhalt

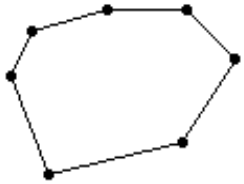
1. Einleitung

Die algorithmische Geometrie ist ein 1975 entstandenes Teilgebiet der Informatik, das sich mit der Entwicklung von effizienten und praktikablen Algorithmen zur Lösung geometrischer Probleme und der Bestimmung deren algorithmischer Komplexität beschäftigt. Zu ihren Aufgaben zählen u.a. die Berechnung der Schnittpunkte von Liniensegmenten, lineare Optimierung oder Segmentierung von Räumen.
Wir beschäftigen uns hier mit dem Teilgebiet der konvexen Hüllen.

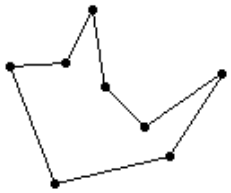
Die konvexe Hülle einer Punktemenge M ist per Definition „das kleinste konvexe Polygon, in dem M enthalten ist“. Um mit dieser Definition etwas anfangen zu können, muss man zuerst wissen, was ein konvexes Polygon ist und was „konvex“ überhaupt bedeutet.

Konvex ist das lateinische Wort für gewölbt. In der Mathematik beschreibt man hiermit Formen, die nach außen gewölbt sind. Einfache Beispiele hierfür sind Kreise und im dreidimensionalen Bereich Kugeln. Das Gegenteil von konvex ist konkav, womit nach innen gewölbte Formen beschrieben werden. Ein aufgeblasener Luftballon z.B. ist konvex. Drückt man aber mit einem Finger auf den Luftballon, wölbt er sich an der Stelle nach innen und wird konkav, bis man den Finger wieder wegnimmt.

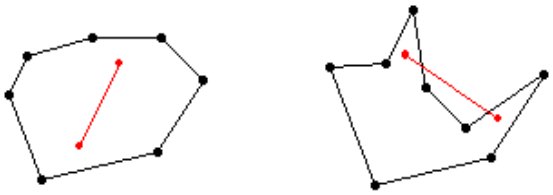
Ein konvexes Polygon ist demnach ein nach außen gewölbtes Polygon und sieht z.B. so aus:



Ein konkaves Polygon dagegen kann eine sehr unregelmäßige Form haben:



Konvexe Polygone zeichnen sich dadurch aus, dass man zwei beliebige Punkte im Inneren des Polygons wählen und miteinander durch eine Gerade verbinden kann, ohne dass man jemals den Rand des Polygons schneiden wird, was bei konkaven Polygonen nicht möglich ist:

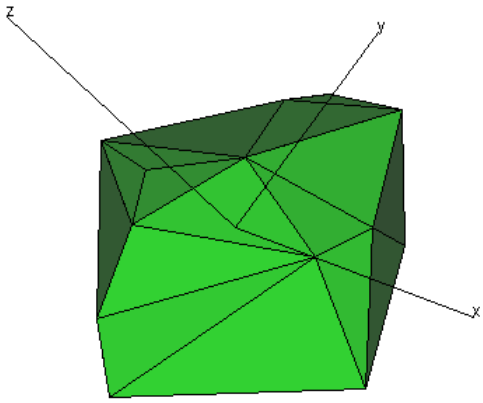


Wie bereits erwähnt ist eine konvexe Hülle das kleinste konvexe Polygon, welches eine Punktemenge komplett einschließt:

Konvexe Hüllen haben folgende Eigenschaften:

1. Sie bestehen aus minimal drei und maximal allen Punkten der Punktemenge
2. Alle Punkte des Hüllpolygons gehören zur Punktemenge die es umschließt
3. Diejenigen Punkte der Menge mit den größten bzw. kleinsten x- und y-Koordinaten liegen immer auf der konvexen Hülle

Auch für dreidimensionale Punktemengen können konvexe Hüllen berechnet werden. Das Ergebnis ist hier ein konvexes Polyeder wie z.B. dieses:



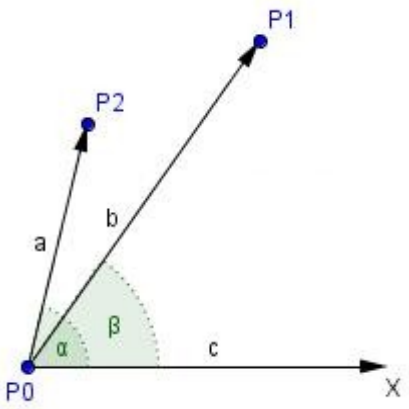
Es gibt diverse Einsatzgebiete für Algorithmen zur Berechnung von konvexen Hüllen. So werden sie z.B. in der Computergrafik und in der Robotik zur Kollisionserkennung verwendet und auch bei Geo-Informationssystemen und zur Erstellung von Voronoi-Diagrammen benötigt man konvexe Hüllen.

Im Folgenden stellen wir einige bekannte Algorithmen zur Berechnung der konvexen Hülle einer Punktemenge vor.

Alle diese Algorithmen arbeiten mit Winkel- oder Entfernungsvergleichen zwischen zwei Punkten. Es ist aber nicht nötig, diese Winkel oder Entfernungen tatsächlich zu berechnen, weil der genaue Wert für die Aufgabenstellung nicht relevant ist. Es reicht es zu wissen, welcher Punkt den größeren Winkel bzw. die größere Entfernung hat. Um dies herauszufinden, verwenden wir folgende Formel:

$$T(p_0,p_1,p_2) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0) = \begin{cases} > 0 & p_2\ ist\ rechts\ von\ p_0\ nach\ p_1 \\ = 0 & p_2\ ist\ auf \quad p_0\ nach\ p_1 \\ < 0 & p_2\ ist\ links\ von\ p_0\ nach\ p_1 \end{cases}$$

Diese Formel berechnet mittels des Kreuzproduktes den Flächeninhalt des Dreiecks aus p_0, p_1 und p_2. Aus dem Ergebnis lässt sich ablesen, ob der Punkt p_2 sich rechts oder links der Geraden von p_0 nach p_1 befindet, oder ob er auf ihr liegt. Dies gibt Aufschluss über seinen Winkel in Bezug zu p_0:

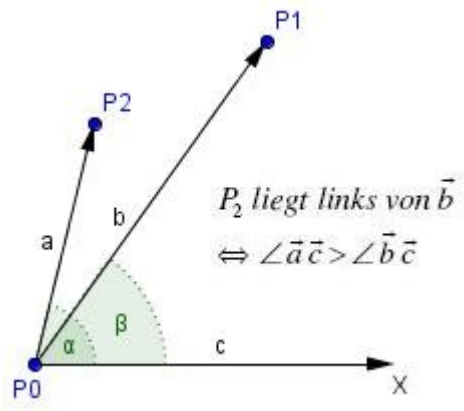


Da p_2 im Bild links der Geraden b von p_0 nach p_1 liegt, ist der Winkel zwischen a und c größer als der Winkel zwischen b und c.
Je weiter der Punkt von der Geraden entfernt ist, desto größer wird das aufgespannte Dreieck. Diese Tatsache nutzen wir, um die Entfernungen von zwei Punkten anhand des Betrags des Formelwertes zu vergleichen.

2. Graham Scan

2.1 Idee

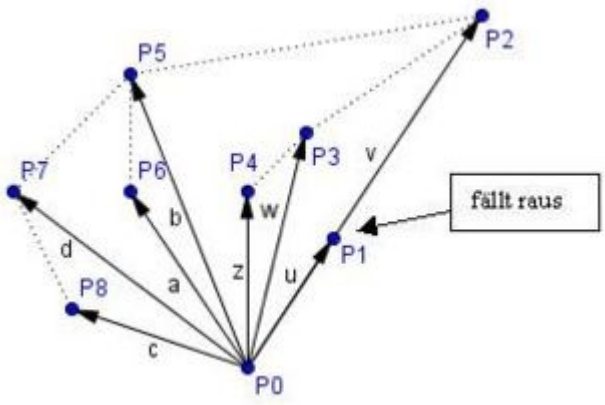
Der Mathematiker Ronald Graham entdeckte 1972 einen effizienten Algorithmus zur Berechnung der konvexen Hülle eines simplen Polygons. Seine asymptotische Laufzeit ist O(n log n).



Sei S = {P} eine endliche Punktmenge. Der Algorithmus beginnt mit einem Punkt der Menge welcher garantiert ein Eckpunkt der konvexen Hülle ist. Man sucht sich dazu den Punkt mit der kleinsten y-Koordinate. Sind dies mehrere, so sucht man sich aus diesen Punkten den mit der kleinsten x-Koordinate. Diese Suche kann in O(n) Schritten durchgeführt werden. Nachdem der Startpunkt P_0 gefunden wurde, sortiert der Algorithmus die restlichen Punkte P in S nach aufsteigendem Winkel zwischen P_0 -> P und der x-Achse gegen den Uhrzeigersinn. Haben dabei zwei Punkte den gleichen Winkel (d.h. liegen mit P_0 auf einer Linie), so wird der Punkt welcher näher an P_0 liegt verworfen.

Will man bestimmen, ob ein Punkt links von einem anderen liegt d.h. ob die Linie eines Punkts mit dem Startpunkt einen größeren Winkel zur x-Achse hat als ein anderer Punkt mit dem Startpunkt, kann man schon in der Einleitung erwähnte Formel benutzen.

2.2 Ablauf

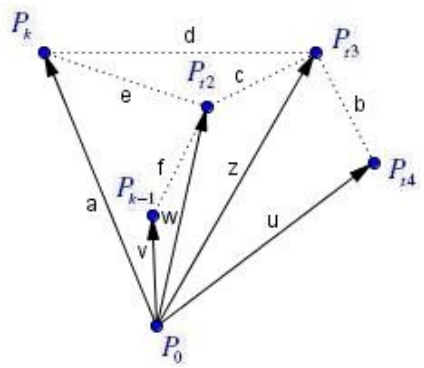


S sei nun die sortierte Punktmenge. Als nächstes läuft man alle Punkte in S durch und prüft, ob diese Eckpunkte der konvexen Hülle sind. Es wird ein Stapelspeicher (Stack) angelegt, auf welchem sich alle Eckpunkte der konvexen Hülle für alle bereits abgearbeiteten Punkte befinden. Zu Beginn liegen P_0, P_1 und P_2 auf dem Stapel. Im k-ten Schritt wird P_k zur Betrachtung herangezogen und berechnet, wie er die vorherige konvexe Hülle verändert. Aufgrund der Sortierung liegt P_k immer außerhalb der Hülle der vorherigen Punkte P_i mit $i < k$.

Durch das Hinzufügen des Punktes kann es vorkommen, dass bereits auf dem Stapel liegende Punkte nicht mehr zur neuen konvexen Hülle gehören. Diese Punkte müssen mittels der „pop“-Operation vom Stapel entfernt werden. Ob ein Punkt noch zur konvexen Hülle gehört oder nicht ermitteln man, indem man berechnet, ob P_k links oder rechts des Vektors PT2->PT1 liegt (PT1 = oberstes Element des Stapels, PT2 = Element direkt unter PT1). Liegt P_k links, so bleibt PT1 weiterhin auf dem Stapel und P_k wird mit „push“ auf dem Stapel abgelegt, liegt P_k rechts, so wird PT1 von der neuen konvexen Hülle „verschluckt“, liegt also „innen", vom Stapel entfernt und die nächsten beiden oberen Punkte untersucht.

Dieser Test wird solange durchgeführt, bis P_k links des Vektors PT2->PT1 oder nur noch P_0 auf dem Stapel liegt. In beiden Fällen wird dann P_k auf dem Stapel abgelegt und mit dem nächsten Punkt P_{k+1} weitergerechnet. Die folgende Abbildung zeigt ein Beispiel, in welchem alle Fälle des eben beschriebenen Tests auftreten.

2.3 Beispiel



In nebenstehender Abbildung werden zunächst die Punkte P_{t4}, P_{t3}, P_{t2} und P_{k-1} auf den Stack gelegt. Zu jedem Zeitpunkt bilden die Punkte auf dem Stack ein konvexes Polygon (gestrichelte Linien). Erst als P_k hinzukommen soll, fliegen P_{k-1} und P_{t2} wieder raus, da sie zusammen mit P_k nicht konvex sind. Die konvexe Hülle dieser Punktmenge besteht aus P_0, P_{t4}, P_{t3} und P_k. P_0 liegt dabei auf dem Stack ganz unten und P_k ganz oben. Die Punkte des gesuchten konvexen Polygons können mit "pop" im Uhrzeigersinn vom Stapel geholt werden.

2.4 Pseudocode

Graham Scan(Q)

let p_0 be the point in Q with minimum y-coordinate, or the leftmost such point in case of a tie

let (p_1, p_2, ..., p_m) be the remaining points in Q, sorted by polar angle in counterclockwise order around p_0 (if more than one point has the same angle, remove all but the one that is farthest from p_0)

PUSH(p_0, S)
PUSH(p_1, S)
PUSH(p_2, S)

for i ← 3 to m
do while the angle formed by the points NEXT-TO-TOP(S), TOP(S), and p_i makes a nonleft turn

do POP(S)

PUSH(p_i, S)
return S

2.5 Laufzeit

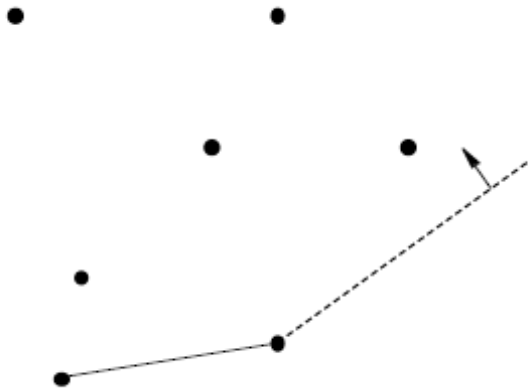
Die Anzahl der "push" und "pop" Operationen übersteigt die obere Grenze von 2n (n = Anzahl der Punkte in der Eingabemenge) nicht. Die Berechnung ist also O(n). Die Sortierung der Punkte nach Winkel kann mit jedem beliebigen Sortieralgorithmus durchgeführt werden, z. B. dem QuickSort. Dieser hat im Durchschnitt *average case* die asympt. Laufzeit von O(n log n). Das bedeutet, dass die Laufzeit des Algorithmus durch die Sortierung vorgegeben ist da O(n) + O(n log n) = O(n log n).

3. Der Einwickel-Algorithmus (Gift Wrapping, Jarvis's March)

Dieser Algorithmus ist auch unter dem Namen *Jarvis's March* oder *Package Wrapping* bekannt. Er wurde ursprünglich 1970 von Chand und Kapur entwickelt und stellte über Jahre den wichtigsten Algorithmus für die Berechnung der konvexen Hülle in höheren Dimensionen dar. Der Name stammt daher, dass der Algorithmus das Einpacken eines Geschenkes mit Papier simuliert.

3.1 Idee

Der Einwickel-Algorithmus geht so vor, wie man üblicherweise Geschenke einwickelt: Man wickelt das Papier oder das Band unter Spannung zyklisch um den Gegenstand. Als Ergebnis erhält man ein eingewickeltes Geschenk, das unabhängig von der Form des Gegenstands konvex ist.

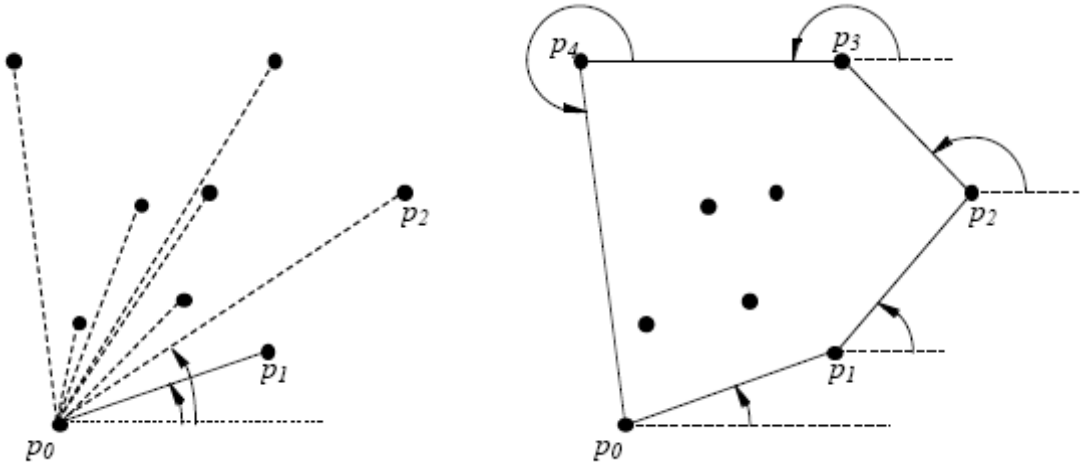


3.2 Ablauf

Um den Algorithmus entwickeln zu können, müssen folgende Fragen geklärt werden:

1. Wie findet man einen Eckpunkt (Extrempunkt) des Polygons als Startpunkt?
2. Angenommen, man hat bereits mindestens einen Eckpunkt gefunden, wie findet man den nächsten Eckpunkt in der gewählten Reihenfolge (Uhrzeiger- oder Gegenuhrzeigersinn)?

Frage 1 ist relativ leicht zu beantworten, denn der linkeste, rechteste, oberste, oder unterste Punkt sind alle Eckpunkte (bei fehlender Eindeutigkeit nimmt man z.B. den untersten linkesten Punkt). Wir wählen also z.B. den (linkesten) untersten Punkt (kleinstes y) p_0 als Ausgangspunkt. Die konvexe Hülle soll im Gegenuhrzeigersinn konstruiert werden.



Der nächste Punkt der konvexen Hülle ist offensichtlich derjenige, der von allen verbleibenden Punkten den kleinsten Polarwinkel bezüglich p_0 bildet. Wir berechnen also alle Polarwinkel bezüglich p_0 als Ursprung und bestimmen das Minimum. Dies führt uns zum nächsten Knoten p_1. Ihn wählen wir wieder als Bezugspunkt und berechnen erneut die Polarwinkel, um den Punkt mit dem Minimum als nächsten Polygonpunkt zu finden. Dies setzt sich fort, bis wir beim Ausgangspunkt p_0 wieder angekommen sind.

3.3 Pseudocode

```
GIFT_WRAP(S)
        l ← tiefster Punkt in S
        nextPoint ← l

        do
                nextPoint auf Stack ablegen

                for each Point p in S
                        berechne den Winkel zwischen nextPoint und p
                        if Winkel < vorher berechneter Winkel dann q ← p

                nextPoint ← q
        while nextPoint != l
```

Es wird zuerst der tiefste Punkt in der Menge gesucht und in l abgelegt. Dann wird eine Variable nextPoint angelegt, die zunächst wiederum l enthält. In der do-while-Schleife wird zunächst nextPoint, also der tiefste Punkt, auf dem Stack abgelegt. Dann werden alle Punkte betrachtet und der nächste Hüllpunkt gesucht. Dazu werden die Winkel zwischen dem letzten gefundenen Hüllpunkt und dem aktuell betrachteten Punkt verglichen. Dabei wird sich derjenige Punkt gemerkt, der den kleinsten Winkel zum letzten gefundenen Hüllpunkt hat. Dieser Punkt wird wiederum in nextPoint gespeichert und auf dem Stack abgelegt. Der Algorithmus endet, wenn er wieder am Ausgangspunkt ankommt.

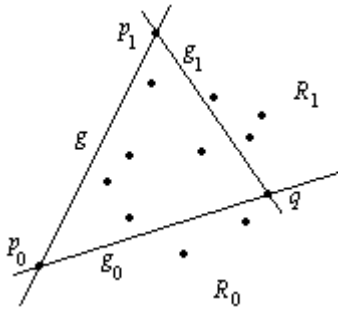
3.4 Komplexität des Einwickelns

- Die Bestimmung eines Ausgangspunkts bedeutet eine Minimumbildung auf der Menge der Punkte, also einen Aufwand von $O(n)$ für die Initialisierung.
- Die Berechnung des Winkels kann offensichtlich in konstanter Zeit durchgeführt werden
- Da die innere Schleife für jeden Punkt diesen Winkel berechnet, besitzt sie eine Komplexität von $O(n)$.
- Die äußere Schleife wird für jeden Eckpunkt des Hüllenpolygons durchgeführt.
- Bezeichnen wir mit h die Anzahl der Polygonpunkte, so ergibt sich ein Aufwand von $O(n\ h)$ für den gesamten Algorithmus.
- Seine Laufzeit hängt daher stark von der Lage der Punkte ab.
- Im ungünstigsten Fall gilt $h = n$ (z.B. wenn alle Punkte auf der Hülle liegen), und der Algorithmus besitzt quadratische asymptotische Laufzeit.
- Der Algorithmus lässt sich auf höhere Dimensionen erweitern und hat zumindest in der dritten Dimension ebenfalls eine Laufzeit von $O(n^2)$.

4. Quickhull

4.1 Idee

Der Quickhull-Algorithmus verwendet eine ähnliche Technik wie Quicksort: Die Punktemenge wird anhand eines Pivotelements in mehrere Teilmengen partitioniert, die dann rekursiv weiterbearbeitet werden. Hierfür werden zuerst zwei Punkte gesucht, die mit Sicherheit zur konvexen Hülle gehören. Dies sind z.B. der tiefste und der höchste Punkt der Menge, also die Punkte mit der minimalen bzw. maximalen y-Koordinate. Alternativ können auch die beiden Punkte mit der minimalen und maximalen x-Koordinate genommen werden, für den Algorithmus spielt dies keine Rolle. Durch die beiden gefundenen Punkte wird eine Gerade gezogen. Jetzt wird der Punkt gesucht, der den größten Abstand zu dieser Gerade hat. Dies ist das Pivotelement und in jedem Fall auch ein Punkt der konvexen Hülle. Von diesem Punkt aus wird je eine weitere Gerade zum Anfangs- und Endpunkt der ersten Gerade gezogen, so dass sich ein Dreieck ergibt. Bild 1 verdeutlicht dies:



P_0 ist der tiefste Punkt, P_1 der höchste. Als Pivotelement wird Punkt q gewählt, weil er den größten Abstand zur Gerade g hat. Durch die Geraden g_0 und g_1 wird nun zusammen mit g ein Dreieck geformt. Es ergeben sich damit die Teilmengen R_0, die alle Punkte rechts von g_0 enthält, und R_1, mit allen Punkten rechts von g_1. Die Punkte innerhalb des Dreiecks können in keinem Fall auf der konvexen Hülle liegen und werden nicht weiter betrachtet, was sich positiv auf die Geschwindigkeit des Algorithmus auswirkt. Für die Mengen R_0 und R_1 mit ihren Geraden g_0 und g_1 wird das Verfahren jetzt rekursiv fortgesetzt. Die Rekursion stoppt, wenn die betrachtete Teilmenge nur drei oder weniger Punkte enthält. Dies sind dann die Endpunkte der jeweiligen Gerade und evtl. das neue Pivotelement, die alle zur konvexen Hülle gehören.

4.2 Pseudocode

```
QUICKHULL(S)
        l ← tiefster Punkt in S
        r ← höchster Punkt in S

        A ← Alle Punkte links von Gerade lr
        B ← Alle Punkte rechts von Gerade lr

        l auf Stack ablegen
        QHULL(A, l, r)
        r auf Stack ablegen
        QHULL(B, l, r)



QHULL(S, l, r)
        if |S| < 3 then Abbruch
        else
                q ← entferntester Punkt von Gerade lr
                if |S| = 3 then
                        q auf Stack ablegen
                        Abbruch
                else
                        A ← Alle Punkte links von Gerade lq
                        B ← Alle Punkte links von Gerade qr

                        QHULL(A, l, q)
                        q auf Stack ablegen
                        QHULL(B, q, r)
```

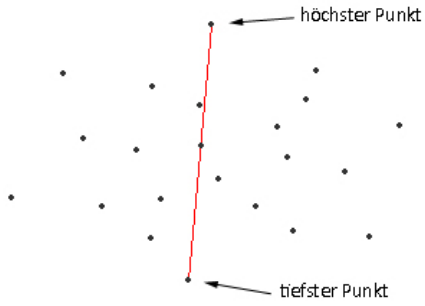
Die Prozedur QUICKHULL sucht die beiden Ausgangspunkte und teilt die Menge in die beiden Teilmengen rechts und links der Gerade auf. Danach wird für beide Teilmengen die Prozedur QHULL aufgerufen, die sich um die rekursive Bearbeitung kümmert. Am Ende des Algorithmus sollen alle Hüllpunkte auf einem Stack liegen.
In QHULL wird zuerst geprüft, ob die untersuchte Menge weniger als drei Punkte enthält. Sie besteht in diesem Fall nur als den Punkten l und r und muss daher nicht weiter untersucht werden. Dann wird der entfernteste Punkt q gesucht, der ein Hüllpunkt ist. Wenn die Menge S nur drei Punkte enthält, handelt es sich hierbei um l, r und q. In diesem Fall wird q als Hüllpunkt auf den Stack gelegt und die Rekursion abgebrochen, da es keine weiteren Punkte mehr zu untersuchen gibt.
Enthält die Menge mehr als drei Punkte, werden die Teilmengen A und B entsprechend der neu erzeugten Geraden gebildet und rekursiv weiteruntersucht. Der Punkt q wird zwischen den beiden rekursiven Aufrufen auf den Stack gelegt.
Die Zeitpunkte, zu denen die Punkte auf den Stack gelegt werden, sind so gewählt, dass der Stack am Ende des Algorithmus alle Hüllpunkte in der richtigen Reihenfolge enthält.

4.3 Zeitkomplexität

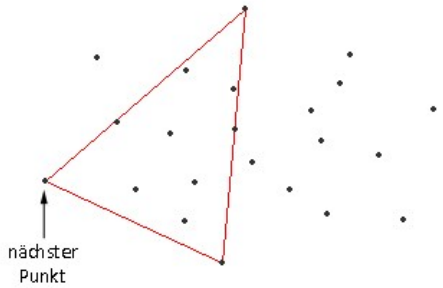
Um die beiden Startpunkte zu finden, müssen alle Punkte einmal betrachtet werden, dies kann also in O(n) Zeit erledigt werden.
Um den entferntesten Punkt von der Geraden zu bestimmen, müssen in jedem Rekursionsschritt alle Punkte der jeweiligen Teilmenge einmal betrachtet werden. Dies entspricht jeweils O(n), bezogen auf die Größe der Teilmenge.
Im besten Fall wird die untersuchte Menge in zwei gleich große Teilmengen aufgeteilt. Dadurch ergibt sich eine Rekursionstiefe von log(n) und somit eine Gesamtkomplexität von O(n log n).
Im worst case beinhaltet die untersuchte Teilmenge in jedem Schritt alle verbleibenden Punkte, so dass sich in jedem Schritt die Teilmenge nur um das Pivotelement verkleinert. In diesem Fall ergeben sich n-1 Rekursionsebenen und damit eine Gesamtkomplexität von $O(n^2)$.

4.4 Beispiel

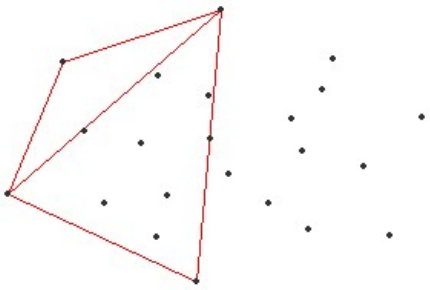
Im Folgenden zeigen wir den Quickhull-Algorithmus an einem Beispiel aus unserem Java-Programm:



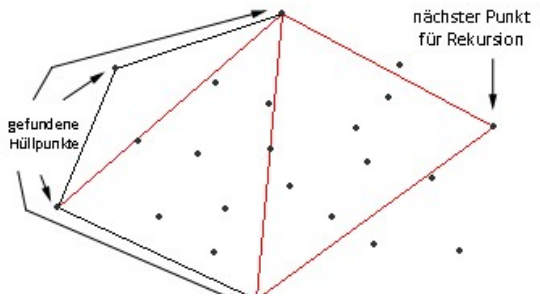
Im ersten Schritt werden der höchste und der tiefste Punkt gesucht und durch eine Gerade verbunden. Es ergeben sich zwei Teilmengen, die getrennt weiterbearbeitet werden: Die Punkte links und rechts von der Gerade.



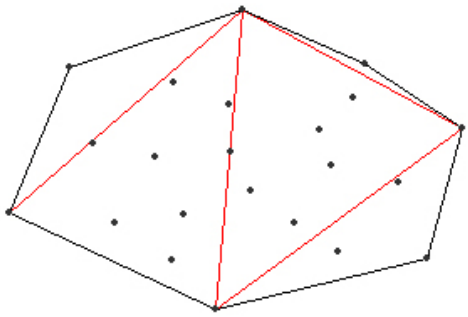
Im zweiten Schritt wird nun die linke Seite betrachtet und der am weitesten entfernte Punkt von der Gerade gesucht. Dieser wird wiederum durch Geraden mit den ersten beiden Punkten verbunden.



Durch die Aufteilung von Schritt zwei wird die Menge weiter partitioniert. Die Menge links von der oberen Gerade wird rekursiv weiteruntersucht. Links von der unteren Kante des Dreiecks sind keine Punkte mehr vorhanden, für diese Kante ist die Rekursion damit abgeschlossen.



Die linke Hälfte der konvexen Hülle ist gefunden und wird schwarz markiert. Nun wird nach demselben Verfahren der rechte Teil gesucht.



Nachdem auch der rechte Teil der Punktemenge bearbeitet ist, ist die konvexe Hülle komplett gefunden (schwarze Kanten) und der Algorithmus beendet.

5. Algorithmus von Chan

5.1 Idee

Der Algorithmus von Chan ist ein schneller Algorithmus zur Berechnung der konvexen Hülle von zwei- oder dreidimensionalen Punktemengen.

Der Algorithmus geht in zwei Schritten vor:
Im ersten Schritt, dem Vorverarbeitungsschritt, teilt der Algorithmus die Punktemenge in mehrere Teilmengen auf und berechnet jeweils die konvexe Hülle mit Graham Scan, falls es sich um eine zweidimensionale Punktmenge handelt. Andernfalls muss ein anderes Verfahren eingesetzt werden, z.B. Quickhull, weil Graham Scan nicht für den dreidimensionalen Raum geeignet ist.

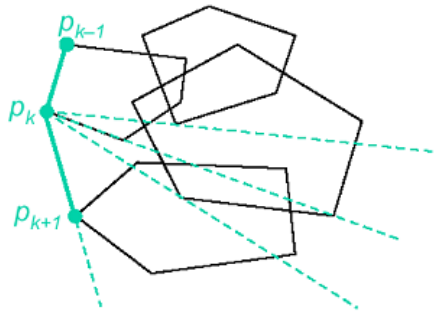
Im zweiten Schritt, dem Einwickelschritt, werden die einzelnen Teilhüllen durch das von Jarvis March bekannte Gift Wrap-Verfahren zu einer Gesamthülle verbunden.

Der Algorithmus ist also eine Kombination von Graham Scan bzw. einem 3D-geeigneten Algorithmus und Jarvis March.

5.2 Genaues Verfahren und Zeitkomplexität

Vorverarbeitungsschritt:
Es wird ein m gewählt (1 ≤ m ≤ n) und die Punktemenge in $\lceil n/m \rceil$ Teilmengen aufgeteilt. Die Größe jeder dieser Teilmengen ist ≤ m. Die einzelnen Teilhüllen können dann mit Graham Scan oder Quickhull in O(m log m) Zeit berechnet werden. Für alle $\lceil n/m \rceil$ Teilhüllen zusammen genommen ergibt sich eine Laufzeit von O((n/m) m log m) = O(n log m).

Einwickelschritt:
Um die nächste Kante der konvexen Hülle zu finden, müssen nicht mehr wie beim reinen Jarvis March-verfahren alle Punkte betrachtet werden, sondern nur die Punkte auf den Teilhüllen. Es werden jetzt vom aktuellen Punkt Tangenten an alle Teilhüllen gelegt. Hierfür wird mit der binären Suche der in Betracht kommende Punkt gesucht, was jeweils in O(log m) Zeit geschehen kann:

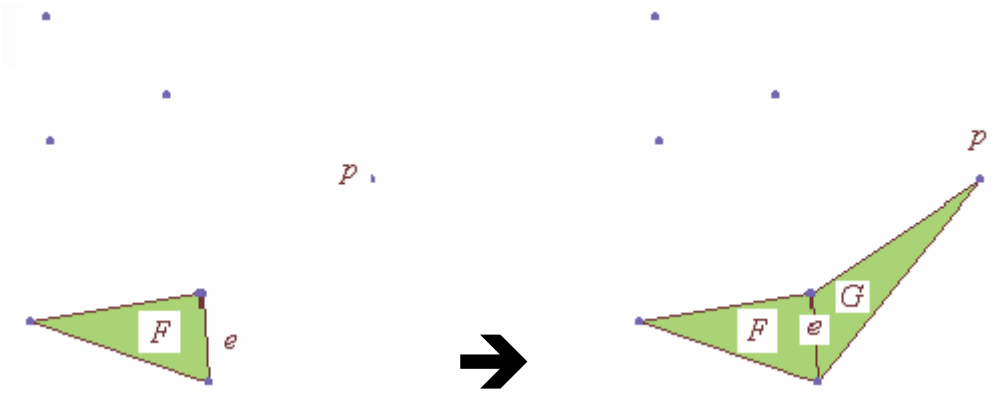


Da $\lceil n/m \rceil$ Teilhüllen existieren, beträgt die Laufzeit für einen Einwickelschritt O((n/m) log m). Für die gesamte konvexe Hülle der Punktmenge, die aus h Hüllpunkten besteht, ergibt sich also für den zweiten Schritt des Chan-Algorithmus eine Laufzeit von O(h * ((n/m) log m)).

Die Gesamtlaufzeit des Chan-Algorithmus beträgt somit
O(n log m + h * ((n/m) log m)) = O(n (1 + h/m) log m).

Im best case wird m so gewählt, dass es h entspricht und die Laufzeit in diesem Fall
O(n (1 + h/h) log h) = O(n (1 + 1) log h) = O(n log h) beträgt.

Auch bei dreidimensionalen Punktehüllen zeigt Chans Algorithmus ein gutes Laufzeitverhalten. Der reine Gift-Wrapping-Algorithmus hat im dreidimensionalen Raum eine Komplexität von $O(n^2)$. Durch die Vorverarbeitung des Algorithmus von Chan wird jedoch die relevante Punktemenge verringert, wodurch Laufzeit gespart werden kann. Das Prinzip des Einwickelns ist dasselbe wie im zweidimensionalen Raum, außer dass nicht mit Punkten sondern mit Facetten der dreidimensionalen Hülle gearbeitet wird. Zuerst wird eine Facette F ausgewählt, die in jedem Fall auf der konvexen Hülle liegt. Von dieser Facette wird eine Kante e gewählt. Nun denkt man sich eine Ebene, die um die Kante e in Richtung der Punktmenge „gebogen“ wird, bis sie auf einen Punkt trifft. Dieser Punkt ist ein neuer Hüllpunkt und es wird eine neue Facette G mit diesem Punkt erstellt:



Genau wie im zweidimensionalen Raum wird hier also mit einem minimalen Winkel gearbeitet. Man sucht den minimalen Drehwinkel der gedachten Ebene zur Facette F.

Quellen

http://www.inf.fh-flensburg.de/lang/algorithmen/geo/index.htm
http://de.wikipedia.org/wiki/Graham_Scan
http://www.ikg.uni-hannover.de/lehre/katalog/alg_geom/v5_Punkthuellen.pdf
http://www.ig.jku.at/strobl/2006-06steiner.pdf
http://www.pi6.fernuni-hagen.de/GI/wasist/
http://de.wikipedia.org/wiki/Algorithmische_Geometrie

Computational Geometry, An Introduction, Preparata, Shamos
Introduction to Algorithms, Cormen, Leiserson, Rivest, Stein