Fachhochschule Köln
University of Applied Sciences Cologne
Campus Gummersbach

Algorithmische Anwendungen

Algorithmen zur Berechnung konvexer Hüllen von Punkten

Gruppe: C
Team: lila

Benz Andreas
Matrikel-Nr.: 11036930

Radke Eugen
Matrikel-Nr.: 11037089

Inhaltsverzeichnis

1. Einführung

Das Forschungsgebiet der algorithmischen Geometrie ist Ende der 70er Jahre aus dem Bereich der allgemeinen Analyse von Algorithmen hervorgegangen. Inzwischen sind eigene Journale und Konferenzen entstanden und das Gebiet verfügt über eine aktive Gemeinde von Forschern.

Zu den zu lösenden Problemen gehört z. B. die Berechnung der Schnittpunkte von Liniensegmenten, die Zerlegung von Polygonen in Dreiecke, die Erforschung spezieller Datenstrukturen für geometrische Probleme wie z. B. Segment-Bäume oder Range-Bäume und die Lokalisierung der eigenen Position anhand von geographischen Karten. Die Berechnung von sog. Voronoi- Diagrammen und der **konvexen Hülle** einer Menge von Punkten gehören zu den Grundproblemen der **algorithmischen Geometrie**.

Die algorithmische Geometrie hat viele Anwendungsgebiete, wie z. B. Computergrafik, Robotik, geographische Informationssysteme, Datenbanken, CAD/CAM, Modellierung von Molekülen oder Mustererkennung.

1.1 Was ist eine konvexe Hülle?

Vor einer Definition der *konvexen Hülle* wird der Begriff der *konvexen Menge* benötigt.

Def. 1 ***konvexe Menge:*** Eine Teilmenge $S \subseteq \mathbb{R}^d$ ist eine *konvexe Menge*, wenn für jedes Paar von Punkten $p, q \in S$ das Liniensegment $\overline{pq}$ vollständig in *S* enthalten ist.

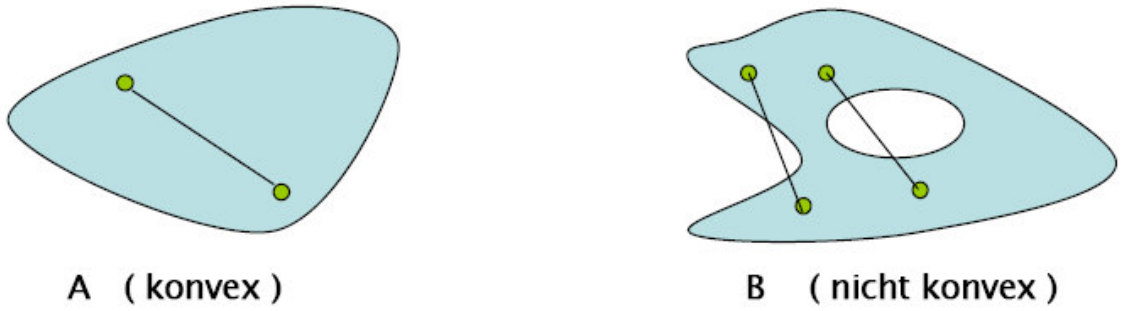


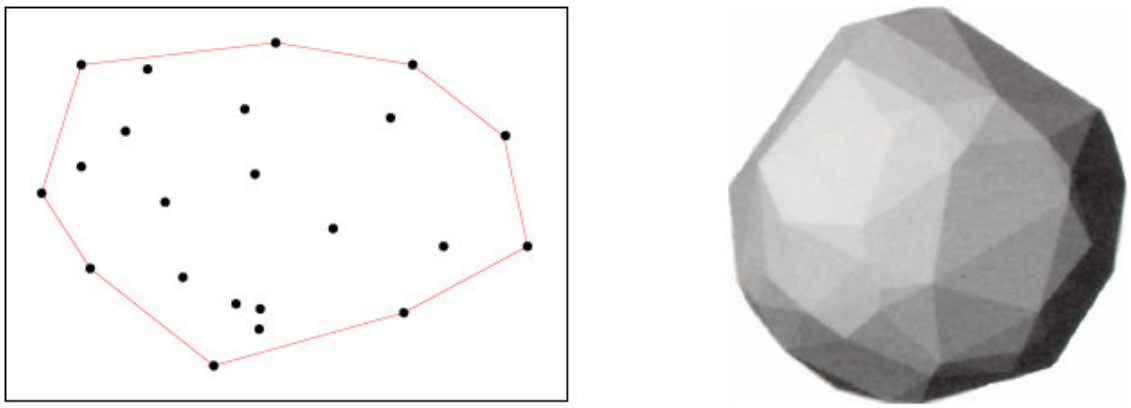
Abbildung 1: Konvexe und nicht konvexe Hüllen.

Def. 2 ***konvexe Hülle*****:** Die *konvexe Hülle CH(S)* von S ist die kleinste konvexe Menge, die S enthält.

Das „d“ in R^d können wir als Anzahl der Dimension verstehen. Das Ergebnis ist im R^2 ein *Polygon*, das durch die Folge seiner Randpunkte im Gegenuhrzeigersinn angegeben wird, im R^3 ein *Polyeder* und im R^n ein *Polytop*. Aber jetzt beschränken wir uns nur auf ebene Problemstellung bzw. im R hoch 2 oder 2D.

Ein *Polygon P* im R² ist eine endliche Menge gerader Liniensegmente, so dass jeder Endpunkt zu genau zwei Segmenten gehört. Die Segmente sind die *Kanten* des Polygons und die Endpunkte die *Ecken*.

Ein Polygon ist *einfach*, wenn alle Paare nicht aufeinander folgender Kanten keinen Punkt miteinander gemeinsam haben. Ein einfaches Polygon teilt die Ebene in zwei nicht zusammenhängende Regionen ein, eine (begrenzte) *innere* und eine (unbegrenzte) *äußere*, die durch das Polygon getrennt werden.



(a) Polygon in $\mathbb{R}^2$ (b) Polytop in $\mathbb{R}^3$

Abbildung 2: Beispiele für die konvexe Hülle mit endlicher Punktmenge.

Ein einfaches Polygon *P* ist *konvex*, wenn seine Begrenzung und die innere Region eine konvexe Menge ist.

Theorem 1: *Die konvexe Hülle einer endlichen Menge S von Punkten in der Ebene ist ein einfaches Polygon. Außerdem muss jede Ecke der Hülle ein Punkt aus S sein.*

1.2 Eigenschaften von konvexen Hüllen

Die Eigenschaften der konvexen Hüllen, die uns bei der Implementierung der unten aufgeführten Algorithmen von Bedeutung sein werden, sind in folgenden Punkten zusammen gefasst:

1. Eckpunkte einer konvexen Hülle *CH(S)* ist eine Teilmenge der Punktemenge *S*.

2. Punkte mit den größten bzw. kleinsten x- und y- Koordinaten sind Elemente der konvexen Hülle in 2D.

3. Mindestens 3 und Maximal alle Punkte bilden die Konvexe Hülle in 2D.

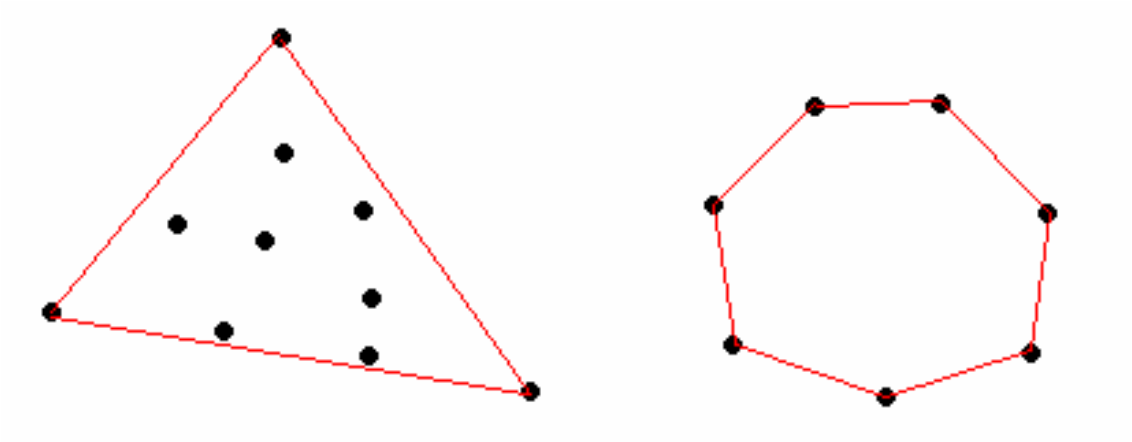


Abbildung 3: Visualisierung der 3. Eigenschaft.

2. Algorithmen

Das Problem der Berechnung der konvexen Hülle wirkt auf den ersten Blick bereits in der Ebene schwierig. So ergibt eine erste Annäherung an das Problem durch den Algorithmus INTERIORPOINTS eine Zeitkomplexität von $O(n^4)$ und eine Verbesserung durch den Algorithmus EXTREMEEDGES eine Zeitkomplexität von $O(n^3)$.

Im Laufe der Zeit wurden jedoch immer schnellere und einfacher zu implementierende Algorithmen entwickelt, von denen einige bekannte in diesem Abschnitt vorgestellt werden. Nicht alle Algorithmen lassen sich auf höhere Dimensionen erweitern oder weisen eine vergleichbar gute Zeitkomplexität wie in der Ebene auf.

2.1 ´Gift Wrapping´- Algorithmus

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.

Im zweidimensionalen Fall startet der Algorithmus am untersten Punkt der Punktmenge, der in jedem Fall zur konvexen Hülle gehört. Dann wird das „Papier" nach rechts gezogen, stramm gehalten und so lange nach oben bewegt, bis ein weiterer Punkt berührt wird. So fährt das Verfahren fort, bis es wieder bei dem Punkt angekommen ist, mit dem es begonnen hat. Alle besuchten Punkte bilden die konvexe Hülle.

Der Algorithmus macht sich die Eigenschaft zunutze, dass jeder Punkt der konvexen Hülle den jeweils kleinsten Polarwinkel zu seinem (bereits ermittelten) Vorgänger auf der Hülle besitzt. So müssen für jeden Punkt auf der Hülle *n Winkel* berechnet werden. Dies führt zu einer Laufzeit von $O(nh)$, wenn *h die Anzahl der Punkte* der konvexen Hülle ist, im *worst case* also zu $O(n^2)$.

Der Algorithmus lässt sich auf höhere Dimensionen erweitern und hat zumindest in der dritten Dimension ebenfalls eine Laufzeit von $O(n^2)$.

2.1.1 Die Idee

Wir werden sehen wie die konvexe Hülle entgegen dem Uhrzeigersinn dadurch konstruiert wird:

1. Im ersten Schritt muss der Starpunkt p_0 bestimmt werden. Da wir eine große Menge an Punkten haben nutzen wir die 2. konvex Hülleneigenschaft (siehe 1.2) zur Bestimmung von p_0.

2. Nach der Festlegung des Startknotens werden wir nun seine Nachbarn betrachten. Wir erstellen dazu eine virtuelle Gerade ausgehend vom Startknoten einfach nach rechts. Nun wir der kleinste Winkel zwischen den Nachbarn Knoten und der virtuellen Gerade bestimmt.

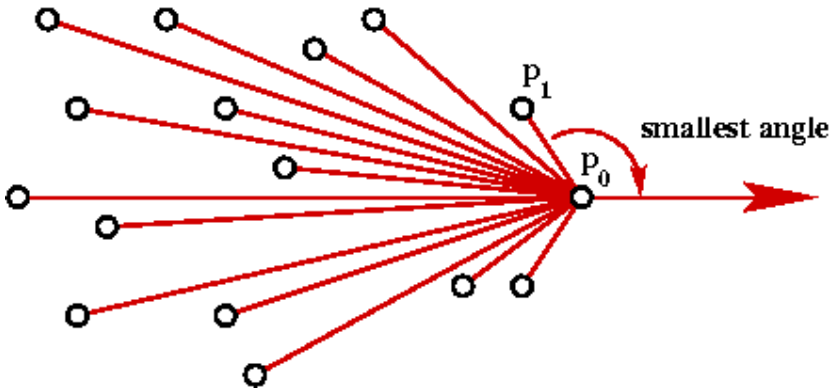


Abbildung 4: Winkelbestimmung zwischen P0 und P1 mit Hilfe einer virtuellen Gerade.

3. Ausgehend von dem erreichten Knoten wird die Bestimmung des kleinsten Winkels zu seinen Nachbarn vorgeführt, wobei den Bezug bildet die im vorherigem Schritt gefundene Kante der konvexen Hülle. Diese Schritte wiederholen wir solange, bis wir durch das Verfahren wieder auf dem ursprünglichen Startknoten landen. In diesem Moment ist die konvexe Hülle durch die Besuchsreihenfolge der Knoten erstellt.

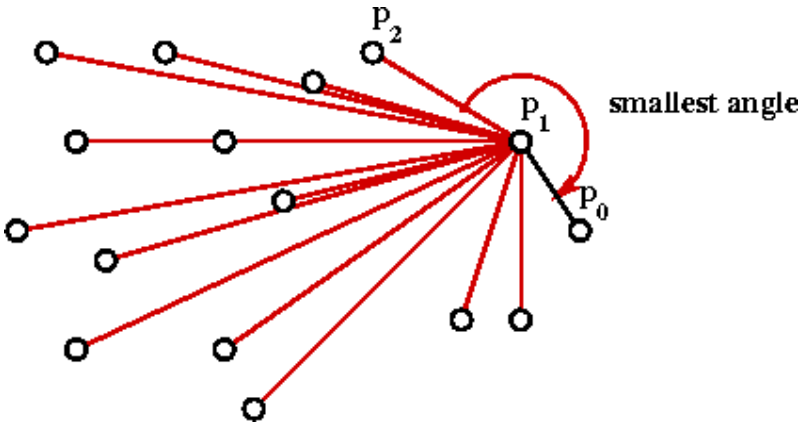


Abbildung 5: Vorführen des Verfahrens (1).

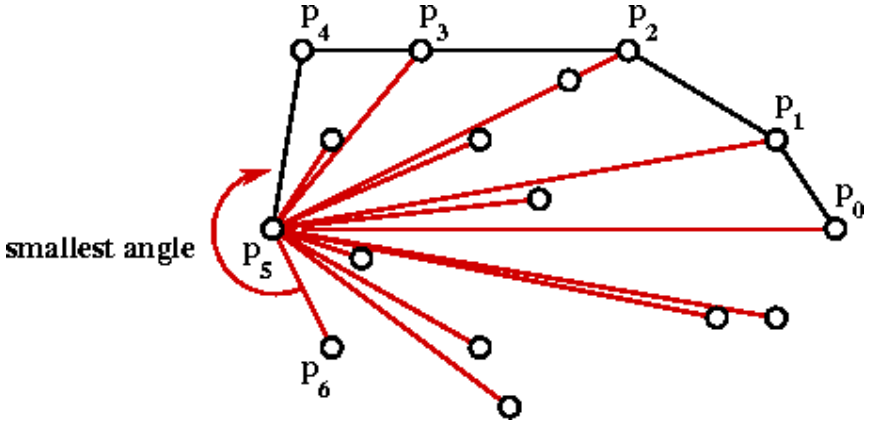


Abbildung 6: Vorführen des Verfahrens (2).

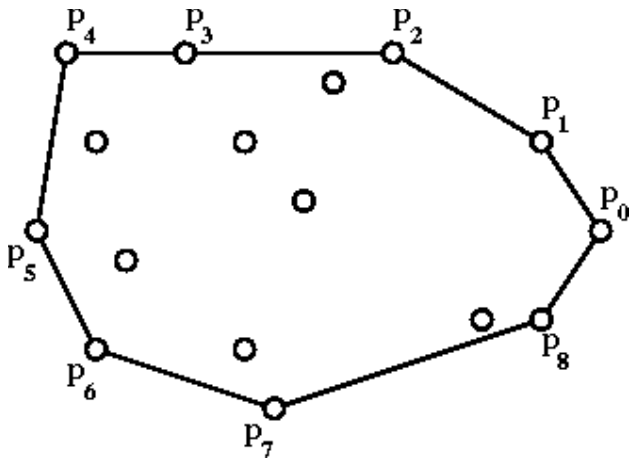


Abbildung 7: Die mit ´Gift Wrapping´ gefundene konvexe Hülle.

2.1.2 Laufzeit

Das finden des Befestigungspunktes kostet uns *O(n)* Zeit, da alle Punkte der Menge jeweils untersucht werden müssen. Haben wir den Punkt gefunden, können wir die weiteren Schritte untersuchen. In jedem Schritt wird ein Eckpunkt der konvexen Hülle gefunden, wodurch wir also höchstens *h*, die Anzahl der Eckpunkte der konvexen Hülle, Schritte benötigen werden. Wir haben also jeweils *n* Punkte zu testen und das *h*-mal, wodurch wir eine Laufzeit von *O(hn)* erhalten.

Natürlich ist auch eine *worst case* Betrachtung unbedingt erforderlich. Wir haben eigentlich *O(hn)*, aber wenn alle Punkte der Menge *S* auch zur konvexen Hülle *CH(S)* gehören, kommt man auf eine quadratische Laufzeit von $O(n^2)$. Sollten die Punkte wirklich so ausgerichtet sein, ist dieser Algorithmus mit seiner quadratischen Laufzeit nicht effizient. Dagegen ist der Algorithmus natürlich sehr schnell, wenn nur wenige Punkte der gesamten Punkte-Menge *S* zur konvexen Hülle gehören.

2.1.3 Psoudocode

```
Algorithm GiftWrapHull(S)
 Input: S is a set of points on the plane.
 Output: The convex hull of S.

 H <- empty sequence
 a <- rightmost point in S (i.e. maximum x-value)
 p <- a
 do
   H.insertLast(p)
   r <- some point in S not equal to p
   for each point q in S not equal to p or q do
     if orientation(p,q,r) = LEFT then
       r <- q
   p <- r
 while p # a
 return H
```

2.2 ´Graham Scan´- Algorithmus

Graham entwickelte 1972 den ersten Algorithmus für die Ebene mit einer Zeitkomplexität von *O(n log n)*.

Der Algorithmus wählt zuerst den Punkt mit der kleinsten y-Koordinate aus (den am weitesten links liegenden, falls es mehrere Punkte gibt) und sortiert alle anderen Punkte anhand ihres Winkels zu diesem ersten Punkt. Danach besucht der Algorithmus nacheinander jeden Punkt in sortierter Reihenfolge und überprüft, ob die letzten beiden der Punkte, die er während des Durchlaufs auf einem Stapel ablegt, eine Kurve nach rechts bilden. Falls dies so ist, werden so lange Punkte vom Stapel entfernt, bis die letzte Bedingung nicht mehr zutrifft und der Algorithmus fährt mit dem nächsten Punkt fort, bis alle Punkte abgearbeitet sind.

Die Suche nach dem Punkt mit der kleinsten y-Koordinate benötigt *O(n)* Schritte. Die Sortierung kann in *O(n log n)* Schritten bewältigt werden. Die Schleife zum Abarbeiten der sortierten Punkte wird *O(n)* mal durchlaufen. Jede Pop-Operation entfernt einen Punkt dauerhaft, so dass insgesamt nur *n* Punkte vom Stapel entfernt werden können, d.h. die Schleife benötigt höchstens *2n* Schritte. Damit ergibt sich eine Gesamtlaufzeit von *O(n log n)*.

Der Algorithmus lässt sich aufgrund der Sortierung nach Winkeln nicht auf höherdimensionale Fälle erweitern.

2.2.1 Die Idee

Wir werden sehen, wie die konvexe Hülle entgegen dem Uhrzeigersinn dadurch konstruiert wird und schrittweise so genannte konkave Ecken überbrückt werden. Die Idee des Algorithmus ist, ein sternförmiges Polygon zu konstruieren und dieses daraufhin in ein konvexes Polygon umzuformen, welches später unsere gesuchte konvexe Hülle ist. Die drei Arbeitsschritte lassen sich folgendermaßen beschreiben:

1. Im ersten Schritt muss der Starpunkt p_0 bestimmt werden. Da wir eine große Menge an Punkten haben nutzen wir die 2. konvex Hülleneigenschaft (siehe 1.2) zur Bestimmung von p_0.

2. Ausgehend vom Startknoten werden nun alle Winkel zu allen anderen Punkten der Punkte-Menge bestimmt. Nach diesen Winkeln werden nun alle Punkte in der Punkte-Menge entgegen dem Uhrzeigersinn sortiert. Es bildet sich ein sternförmiger Polygonzug. Wir haben also jetzt eine Liste *S*, welche entgegen dem Uhrzeigersinn nach Größen der Winkel zu dem Startknoten sortiert ist.

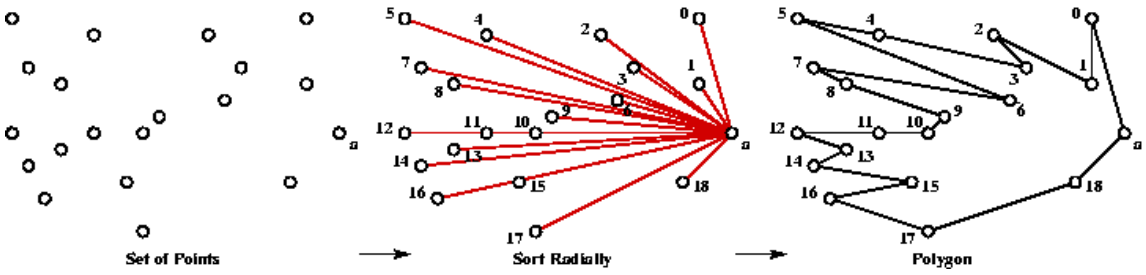


Abbildung 8: Bilden des sternförmigen Polygons.

3. Weiterhin fügen wir noch den Startknoten an die erste und letzte Position an die Liste *S* an. Jetzt kommen wir zu dem bekannten Scan, wodurch der Algorithmus seinen Namen bekam. Wir 'scannen' durch die nach Winkeln sortierte Liste. Dabei stellen wir in jedem Schritt sicher, dass eine weitere Liste *CH* mit den Eckpunkten der konvexen Hüllen immer eine 'konvexe Kette' um die bereits betrachteten Punkte bildet. Wir kommen so jeden Schritt der endgültigen konvexen Hülle näher, müssen aber folgende Tests bei jedem neuen Betrachtungspunkt durchführen.

   • Wenn der aktuelle Betrachtungspunkt eine Linksdrehung mit den letzten beiden Punkten aus *CH* vollzieht oder wenn *CH* weniger als zwei Punkte enthält, dann wird der aktuelle Betrachtungspunkt an das Ende von *CH* hinzugefügt.
   • Sollte dies nicht der Fall sein, wird der letzte Punkt aus *CH* gelöscht und der Test für den aktuellen Betrachtungspunkt wiederholt.

   Wenn wir an dem Startknoten wieder angekommen sind beinhaltet *CH* die konvexe Hülle unserer Punkte-Menge *P*. Bildlich gesprochen haben den sternförmigen Polygonzug ausgehend vom Startknoten durchlaufen und dabei konkave Ecken einfach überbrückt.

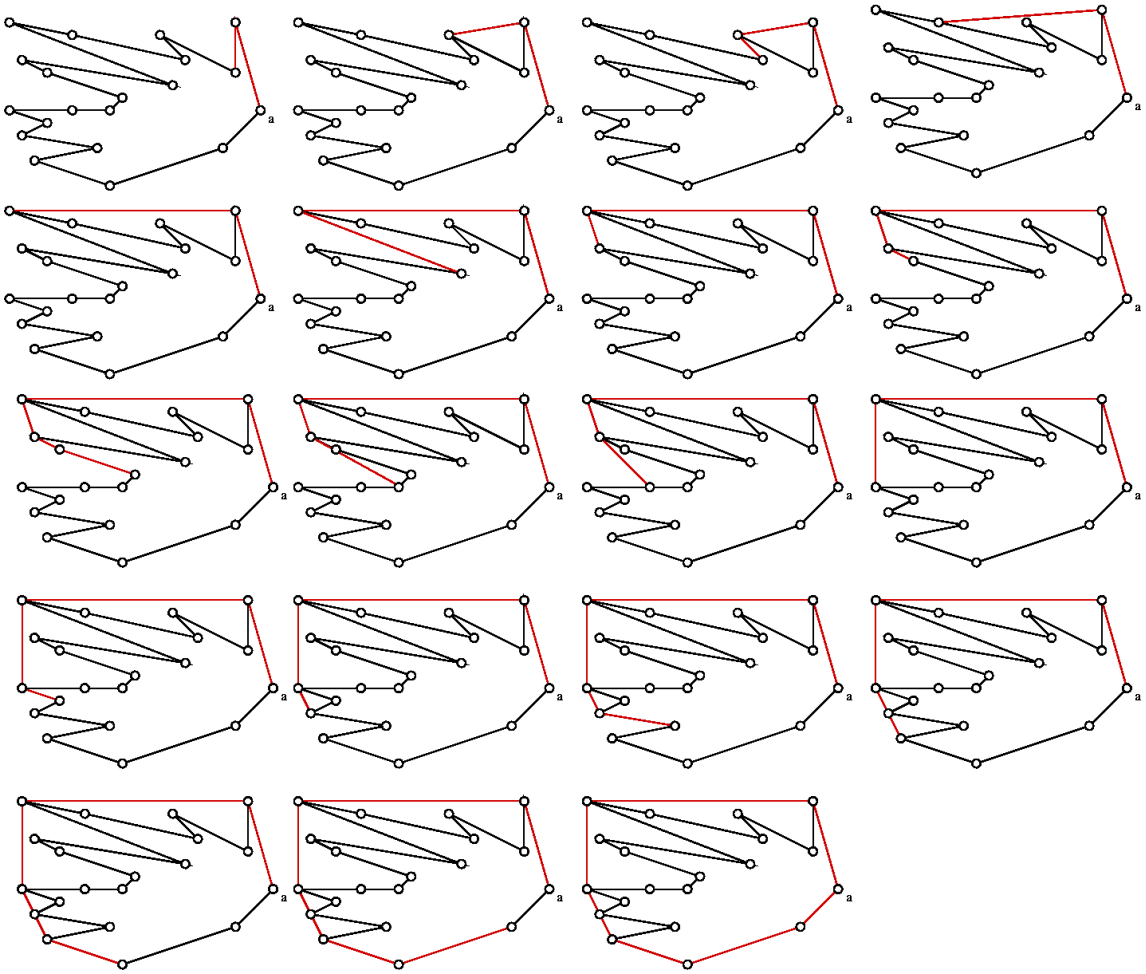


Abbildung 9: Der Scan- Schritt.

2.2.2 Laufzeit

Zur Betrachtung der Laufzeit müssen wir die einzelnen Schritte differenzieren. Wir setzen wieder *n* als Anzahl aller Punkte aus der Punkte-Menge *S*. Schritt 1, das Suchen des Startknotens, dauert *O(n),* da ja alle Punkte untersucht werden müssen. Der zweite Schritt des Algorithmus hängt enorm von der Wahl des Sortieralgorithmus ab. Wählen wir einen asymptotisch guten Sortieralgorithmus, wie die bekannten Heap-Sort oder Merge-Sort, erhalten wir für das Sortieren der Liste nach Winkeln eine Laufzeit von *O(n * log n).*

Der dritte Schritt des Algorithmus ist sehr interessant - man denke an die repeat- Schleife im Pseudocode. Wenn wir bspw. annehmen, dass die repeat- Schleife *2* mal ausgeführt wird, haben wir dank der elementaren Operationen der Liste mit den festen Positionsangaben eine Laufzeit von *O(2n)* für den dritten Schritt des Algorithmus. Abschließend können wir also sagen, dass das Hauptproblem in Schritt 2 liegt - der Sortierung. Daraus resultiert auch die Laufzeit des Algorithmus 'Graham Scan' mit *O(n * log n).*

2.2.3 Psoudocode

```
Algorithm GrahamScanHull(S)
 Input: Set of points S.
 Output: The convex hull of S.

a <- rightmost point in S
P <- array containing the points in S sorted angularly about a
     (assume that a is the first point in P)
s <- empty stack
s.push(a)
p <- P[1]
i <- 2
while i < P.length do
  if orientation(s.top(), p, P[i]) = RIGHT then
    p <- s.pop()
  else
    s.push(p)
    p <- P[i]
    i <- i+1
if orientation(s.top(), p, a) = LEFT then
  s.push(p)
return s
```

2.3 ´Divide and Conquer´- Algorithmus

Wie der Name schon sagt arbeitet der Algorithmus nach dem *divide and conquer*- Prinzip. Dieser Algorithmus lässt sich auch in höheren Dimensionen erweitern.

2.3.1 Die Idee

1. wenn $|S| <= 3$ ist, dann berechne die konvexe Hülle mit *brute force* in *O(1)* Zeit und gehen zurück.

2. Sonst verteile die Punkte aus *S* in zwei Teile *A* und *B*, in denen *A* aus der Hälfte der Punkte mit den niedrigsten x-Koordinaten und *B* aus der Hälfte der Punkte mit den höchsten x-Koordinaten besteht.

3. Berechne rekursiv die konvexen Hüllen von *CH(A)* und *CH(B).*

4. Vermische die beiden Hüllen in eine allgemeine konvexe Hülle *CH*, indem man die obere und untere Tangente für *CH(A)* und *CH(B)* berechnet. Verwerfe alle Punkte, die zwischen diesen zwei Tangenten liegen.

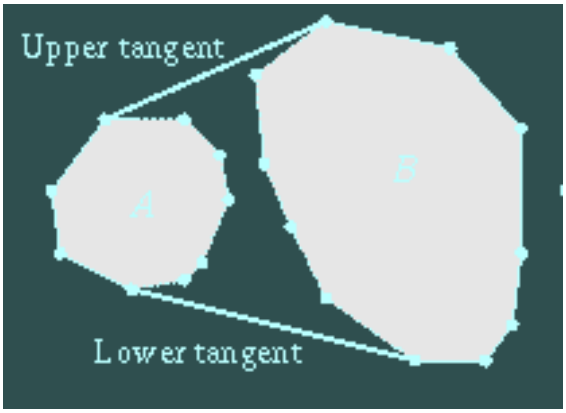


Abbildung 10: Verbinden der beiden Teillösungen zur einer gemeinsamen konvexen Hülle.

2.3.2 Laufzeit

Wie wir es auch von den Algorithmen her kennen, die nach dem *divide and conquer*- Prinzip funktionieren, besitz auch dieser die gesamt Laufzeit von *O(n*log(n)).*

2.4 Algorithmus von Chan

Der Algorithmus von Chan dient zur Berechnung einer konvexen Hüllen für eine gegeben Menge von Punkten in einer Ebene oder in einem drei dimensionalem Raum. Dieser wurde 1996 in Discrete & Computational Geomethry von T.M. Chan veröffentlicht.

2.4.1 Die Idee

Kombination aus ´Gift Wrapping´ und ´Graham Scan´ in 2D und ´Gift Wrapping´ und ´Divide and Conquer´ in 3D.

1. Beschleunigung der Einwicklungsschritten durch Preprocessing:
   - mit ´Graham Scan´ in 2D
   - bzw. ´Divide and Conquer´ in 3D.

2. ´Gift Wrapping´ fügt die aus dem Preprocessing entstandenen konvexen Hüllen zur einen gemeinsamen Hülle in *h* Einwicklungsschritten zusammen.

2.4.1.1 Anwendungen in einer Ebene

In einer Ebene läst sich der Algorithmus von Chan durch die Kombination von dem *´Gift Wrapping´*- und *´Graham Scan´*- Algorithmus realisieren.

Der Chan Algorithmus macht sich den *´divide and conquer´*- Prinzip zu nutze und Teilt die gegeben Menge *S* mit $n \geq 3$ Punkten in *m* Teile. Die Berechnung der konvexen Hülle *CH(Si)* der Teilmengen *Si* erfolgt mit dem *´Graham Scan´*. Das Zusammenfügen der einzelne konvexen Hülle *CH(Si)* zur einer gemeinsamen Lösung *CH(S)* geschieht in dem *´counquer´*- Schritt von dem *´Gift Wrapping´* Algorithmus.

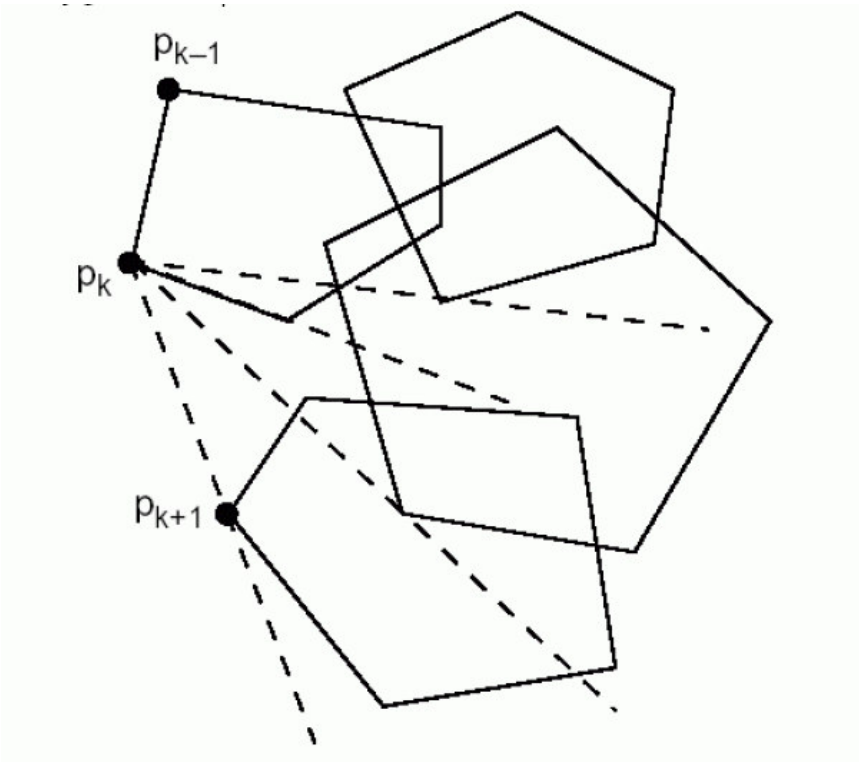


Abbildung 11: Algorithmus von Chan in 2D.

2.4.1.2 Psoudocode

Hull2D(S, m, H) mit $P \subset D^2$, $3 \leq m \leq n$ und $H \geq 1$

1. partitionire die Menge *S* in $S_1,\ldots,S\lceil n/m \rceil$ Teilmengen der Größe *m*
2. for $i = 1,\ldots, \lceil n/m \rceil$ do
3. berechne CH(*Si*) mit ´Graham Scan´ und speichere ihre Knoten gegen den Uhrzeigersinn in ein Array
4. $p_0 \leftarrow (0, -\infty)$
5. $p_1 \leftarrow$ der am weitesten rechts stehende Punkt aus *S*
6. for $k=1,\ldots,H$ do
7. for $i = 1,\ldots, \lceil n/m \rceil$ do
8. berechne den Punkt $q_i \in S_i$ mit dem maximalen Winkel $\angle p_{k-1}p_kq_i$ ($q_i \neq p_k$) durch das Durchführen einer Binärsuche auf die Kanten von CH(S_i)
9. $p_{k+1} \leftarrow$ Punkt *q* aus $\{q_1,\ldots,q\lceil n/m \rceil\}$ so das max. $\angle p_{k-1}p_kq$
10. if $p_{k+1} = p_1$ then return Liste$\{p_1,\ldots,p_k\}$
11. return „nicht komplett“

2.4.2.1 Anwendungen in einem drei dimensionalem Raum

Das Zusammenfügen der einzelne konvexen Hülle *CH(Si)* zur einer gemeinsamen Lösung *CH(S)* geschieht auch hier durch den *´Gift Wrapping´*- Algorithmus. Der Aufruf des *´Graham Scan´* wird allerdings durch den *´Preparat and Hong´*- Algorithmus für die dreidimensionale konvexe Hülle ersetzt mit der gleichen Komplexität. Dieser wird auch als der *´Divide and Conquer´*- Algorithmus bezeichnet. Um die gleiche Zeitkomplexität wie in dem 2D- Raum zu erreichen werden die durch den *´Preparat and Hong´*- Algorithmus berechnete *CH(Si)* in einer *´Dobkin-Kirkpatrik´*- Hierarchie gespeichert.

In höheren Dimensionen werden mit *´Gift Wrapping´* die Facetten der Hülle folgender Massen berechnet: aus einer gegebener Facette *f*, werden drei angrenzenden Facetten *fj* durch die Durchführung des „*wrapping*"- Schrittes über jeden der drei Winkel *ej* aus *f (j =1, 2, 3)* generiert. Um die logarithmische Zeit wie sie in dem 2D Raum vorhanden ist zur erreichen wir zum speichern der vom *´Divide and Conquer´*- Algorithmus berechneter konvexer Hüllen, für die einzelnen Teilbereiche, wird die so genante *´Dobkin-Kirkpatrik*- Hierarchie (welche nur Linearzeitaufbereitung erfordert) genutzt**.**

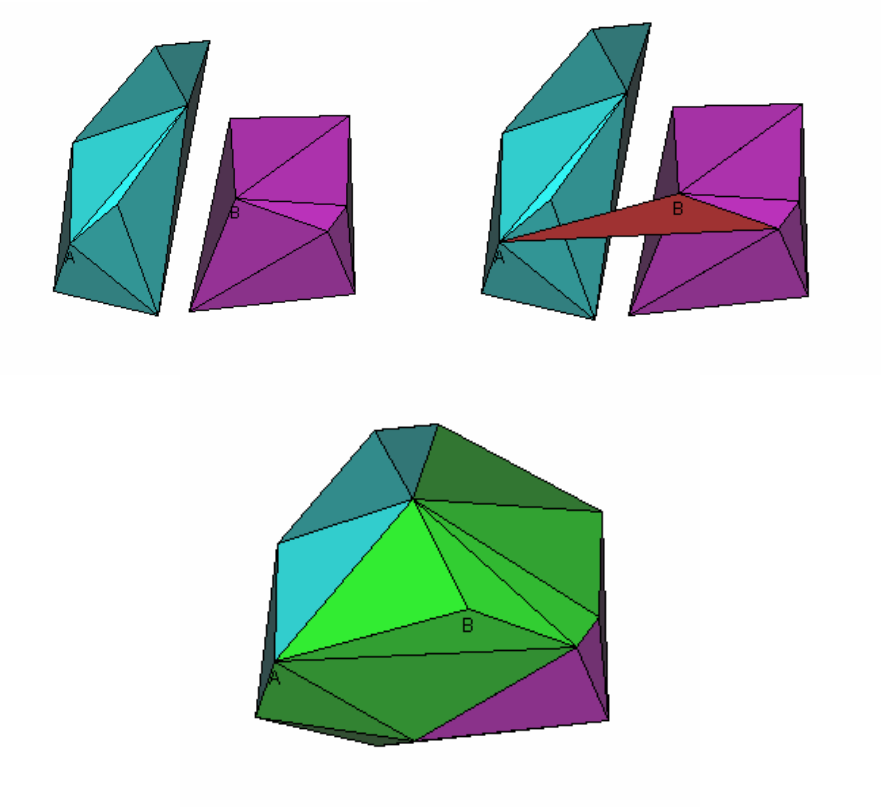


Abbildung 122: Zusammen fügen 2 Teillösungen zur gemeinsamen konvexen Hülle in 3D.

2.4.2.2 Psoudocode

Hull3D(S, m, H) mit P$\subset$ D^3, 4≤m≤n und H≥1

1. partitionire die Menge *S* in $S_1,\ldots,S_{\lceil n/m \rceil}$ Teilmengen der Größe *m*
2. for $i = 1,\ldots, \lceil n/m \rceil$ do
3. berechne CH(S_i) mit ´Preparat and Hong´ und speichere sie in einer ´Dobkin-Kirkpatrik´- Hierarchie
4. $F, Q \leftarrow \{f_0\}$, wenn f_0 irgendeine Ausgangsfacette von CH(S_i) ist
5. for k=1,…,2H-4 do
6. if $Q = 0$ then return F
7. nehme irgendeine $f \in Q$ und setze $Q \leftarrow Q - \{f\}$
8. sei e_j die Winkel von f (j = 1, 2, 3)
9. for j = 1, 2, 3 do
10. for $i = 1,\ldots, \lceil n/m \rceil$ do
11. berechne den Punkt $q_i \in S_i$ so das Maximum der Winkel zwischen f und CH($e_j \cup \{q_i\}$) durch das Durchsuchen der Hierarchie von CH(S_i)
12. $p_j \leftarrow$ Punkt q aus $\{q_1,\ldots,q_{\lceil n/m \rceil}\}$ mit Maximum der Winkel zwischen f und CH($e_j \cup \{q_i\}$) ($q \notin e_j$)
13. $f_j \leftarrow$ CH($e_j \cup \{p_j\}$)
14. if $f_j \notin F$ then
15. $F \leftarrow F \cup \{f_j\}$, $Q \leftarrow Q \cup \{f_j\}$
16. return „nicht komplett“

Q kann entweder als „queue“ oder „stack“ und F als „dictionary“ implementiert sein.

2.4.3 Laufzeit

Das Berechnen der einzelnen Teillösungen *m* geschieht mit dem *´Graham Scan´* in $O(m*\log(m))$. Betrachtet man nun die Laufzeit des *´Graham Scans´* für die Gesamte Menge der Punkte *n*, so muss der Aufteilungsgrad *n/m* hinzugenommen werden. Aus der so entstandenen Formel $O((n/m)(m*\log(m)))$ ergibt sich die gesamt Laufzeit des *´Graham Scans´* von $O(n \log(m))$.

Das zusammen Fügen der einzelnen Teillösungen zu einer gemeinsamen geschieht mit dem *´Gift Wrapping´*- Algorithmus in $O(h*((n/m) \log(m)))$ Zeit. Diese ergibt sich aus folgender Überlegung und zwar die Anzahl der der zur konvexen Hülle gehörender Knoten *h* müssen nicht mit allen *n* Knoten verglichen werden, sondern nur *log(m)*- mal, durch den Preprozess mit den *´Graham Scan´*. *n/m* ist auch hier der vorgenommene Aufteilungsgrad.

Fügt wir nun die so erhaltenen Laufzeit zu einer gesamt Laufzeit des *Chan Algorithmus* zusammen. Durch die zwei oberen Formeln erhält man eine Laufzeit von $O(n*\log(m) + h*((n/m) \log(m)))$ bzw. $O(n(1 + h/m) \log(m))$ durch das ausklammern von *n*.

Im *bast case* ist das *m* so gewählt worden, das $h = m$ ist und so mit sich eine Laufzeit von $O(n \log(h))$ ergibt.

3. ´Dobkin-Kirkpatrik´- Hierarchie

Ausgehend von einem konvexen Polyeder $P_1 = P$ konstruiert man $Pi+1$ aus P_i durch entfernen möglichst vieler, paarweise nicht benachbarter Eckpunkte, bis man einen Tetraeder P_k erhält. Man beachte, dass dadurch stets neue Kanten und Flächen entstehen.

Da die entfernten Punkte nicht benachbart sind, gibt es zu jeder neuen Seitenfläche von $Pi+1$ nur einen, eindeutig bestimmten, „gegenüberliegenden“ Punkt aus P_i, der entfernt wurde. Umgekehrt entstehen für jeden entfernten Punkt höchstens so viele neue Seitenflächen wie dieser Punkt Nachbarpunkte hat, und es gilt offensichtlich

$P_k \subset \ldots \subset P_2 \subset P_1$.

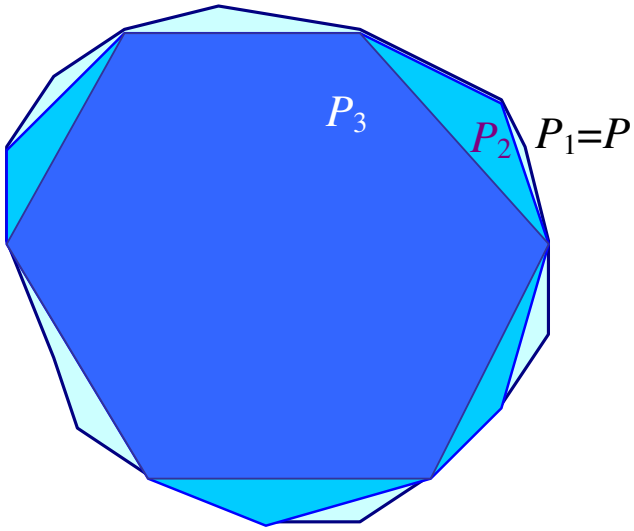


Abbildung 133: ´Dobkin-Kirkpatrik´- Hierarchie.

4. Quellen

[GR1972] GRAHAM, RONALD: *An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set*. Information processing letters., 1(1972), 132-133.

[CH1996] T.M. CHAN: *Optimal output-sensitive convex hull algorithms in two and three dimensions,* Discrete & Computional Geometry, Vol. 16(1996), 361-368.

[COR2003] THOMAS H. CORMAN, CHARLES E. LEISERSON, RONALD L. RIVEST and CLIFFORD STEIN: *Instruction to Algorithms Second Edition*., 2(2003), 947-957.

[MS2003] STOLPE, MARCO: Diplomarbeit: *Ein Algorithmus zur Lösung des Farthest-Pair-Problems.*, 1(2003), 47-51.

[MCGILL] MCGILL UNIVERSITY SCHOOL OF COMPUTER SCIENCE: *Convex Hull*., http://cgm.cs.mcgill.ca/~msuder/courses/250/lectures/convex_hull/

[FHFLENS] FH- FLENSBURG: *Konvexe Hülle*., http://www.iti.fh-flensburg.de/lang/algorithmen/geo/convex.htm