FH University of Applied Sciences
TECHNIKUM WIEN

LABORPROTOKOLL

Im Studiengang AI Engineering - Advanced Programming

Convex Hull Algorithms (Gift-wrapping, Quick Hull)

Ausgeführt von: Hannah Kohlhofer, MSc.
Elias Marcon, MSc.
Ing. Fabian Steiner, BSc.

Personenkennzeichen: 2310585014
2310585020
2310585027

BegutachterIn: FH-Prof. Dipl.-Ing. Alexander Nimmervoll

Wien, den 4. Oktober 2023

Inhaltsverzeichnis

1 Die Algorithmen

Um die Konvexe Hülle einer Punktemenge zu bestimmen bedarf es verschiedener Algorithmen. Für dieses Projekt implementieren wir zwei verschieden Algorithmen - der Quickhull und der Giftwrapping Algorithmus. Beide Algorithmen haben das Ziel die Konvexe Hülle einer Punktmenge - also das kleinste konvexe Polygon zu bestimmen, wobei hier kein Punkt außerhalb dieser Hülle liegen darf. In anderen Worten: Jeder Punkt der Punktemenge liegt entweder auif der konvexen Hülle oder wird von dieser eingesperrt / umschlossen.

1.1 Quickhull

Der Quick Hull Algorithmus ist dazu da aus einer Punktemenge eine Convexe Hülle zu bilden. Die Grundidee hierbei ist das "Divide and ConquerPrinzip. Das bedeutet das Gesamtproblem wird zuerst in mehrere Teilprobleme zerlegt und diese werden dann für sich gelöst und abschließen wieder zusammengesetzt um eine Gesamtlösung zu finden.

1.1.1 Pseudocode

```
1  Funktion QuickHull(S)
2  {
3      // Bestimmt die konvexe Huelle der Menge S
4      Convex Hull := {}
5      A := Punkt ganz links
6      B := Punkt ganz rechts
7      Fuege die Punkte A und B der konvexen Huelle hinzu
8      // Die Gerade AB teilt die verbleibenden n - 2 Punkte in die Teilmengen
           S1 und S2
9      S1 := Menge der Punkte in S, die auf der rechten Seite von AB sind
10     S2 := Menge der Punkte in S, die auf der linken Seite von AB sind
11     FindHull(S1, A, B)
12     FindHull(S2, B, A)
13     Ausgabe: Convex Hull
14 }
15
16 Funktion FindHull(Sk, P, Q)
17 {
18     // Bestimmt die Punkte auf der konvexen Huelle der Menge Sk, die auf der
           rechten Seite der Geraden PQ sind
```

```
19      Wenn Sk keine Punkte enthaelt dann
20          Ausgabe: Convex Hull
21      C := Der Punkt in Sk, der den groessten Abstand von der Geraden PQ hat
22      Fuege den Punkt C in die konvexe Huelle zwischen den Punkten P und Q ein
23      // Die drei Punkte P, Q und C teilen die verbliebenen Punkte von Sk in
            die Teilmengen S0, S1 und S2
24      S0 := Die Punkte innerhalb des Dreiecks PCQ
25      S1 := Die Punkte auf der rechten Seite der Geraden PC
26      S2 := Die Punkte auf der rechten Seite der Geraden CQ
27      FindHull(S1, P, C)
28      FindHull(S2, C, Q)
29      Ausgabe: Convex Hull
30  }
```

Listing 1.1: Quick Hull Pseudocode [**?**]

Hierbei ist gut zu sehen, wie der Algorithmus das Gesamtproblem aufteilt: beim ersten Aufruf wird die Funktion Quick Hull aufgerufen und diese teilt die Menge der Punkte in der Hälfte auf und ruft damit jeweils einmal die Funktion FindHull auf, welche die Punkte wieder aufteilt und sich mit den zwei Hälften selbst erneut aufruft.

1.1.2 Beschreibung des Alogrithmus

Der Algorithmus beginnt damit, dass er den linkesten und den rechtesten Punkt auswählt und sie zur konvexen Hülle hinzufügt. Dadruch entsteht eine Linie (im Pseudocode bezeichnet als AB). Nun wird für die Punktemenge auf jeder Seite der Hülle weiter betrachtet. Man sucht hier nun jeweils nach dem am weitesten entfernten Punkt C zur Geraden. Die Auswahl des am weitesten Entfernten Punktes führt hier zu den Besten Ergebnissen, wie in The Quickhull Algorithm for Convex Hulls"[**?**] beschrieben wird. Man verbindet nun die Gerade AB mit dem Punkt C und erhält dadurch ein Dreieck ($S0$) in dem sich (wahrscheinlich) Punkte befinden, die nicht mehr weiter betrachtet werden müssen (da sie schon innerhalb der Hülle liegen). Wenn sich außerhalb des Dreiecks linkerseits zu AC und rechterseits zu BC Punkte befinden, wird für diese der Schritt FindHull"wiederholt, bis sich alle Punkte in der Hülle befinden.

1.1.3 Aufwandsabschätzungen und Testfälle

Theorie

Laut Barber, Dobkin und Huhdanpaa [**?**] liegt die oberste Schranke bei $\mathcal{O}(n * log(n))$, wenn die Pivotpunkte gut gewählt werden (zum Beispiel der am weitesten entfernte Punkt). Werden sie zufällig gewählt, so liegt die oberste Schranke im 2D bei $\mathcal{O}(n^2)$.

Messwerte

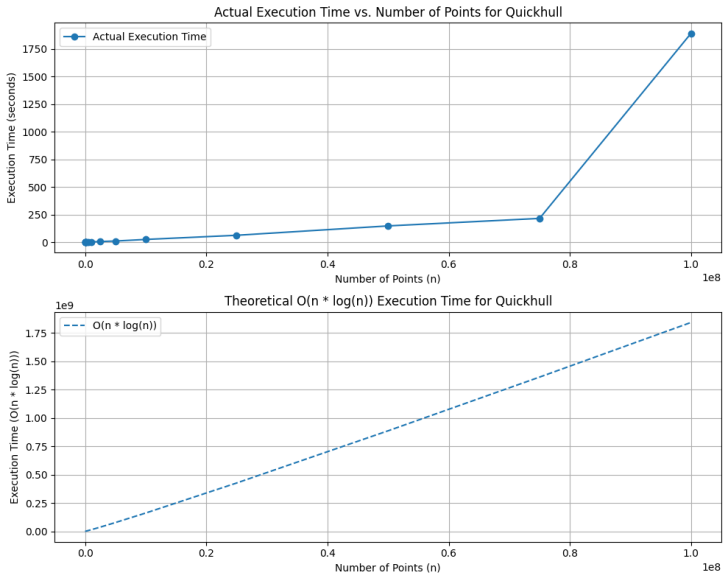

Abbildung 1: QuickHUll: Vergleich von Messwerten (oberer Graph) mit $\mathcal{O}(n * log(n))$

Der Algorithmus verhält sich nur bei gewissen (balancierten) Eingaben $\mathcal{O}(n * log(n))$, bei unbalancierten Eingaben geht das Verhalten gegen $\mathcal{O}(n^2)$. Allerdings ist auch festzuhalten, dass durch die Implementierung in Python nie eine optimale Laufzeitmessung erreicht werden kann, sich der Verlauf der Laufzeit jedoch schon beobachten lässt. Erschwerend kommt hinzu, dass bei hohen Datenmengen die Begrenzung der Hardware (RAM) schlagend wird - damit könnte man auch den plötzlichen Anstieg erklären.

1.2 Giftwrapping

Der Giftwrapping Algorithmus, dient, wie zuvor beschrieben, ebenfalls zur Berechnung der Konvexen Hülle (CH) einer Punktemenge (Q). Im Zwei-Dimensionalen Fall wird der Giftwrapping Algorithmus auch als Jarvis-March bezeichnet, benannt nach seinem ËntdeckerR. A. Jarvis im Jahr 1973 (siehe [**?**]).

1.2.1 Pseudocode

Im Folgenden Code geltend die Bezeichnungen P für den jeweiligen Startpunkt und S für die gesamte Punktemenge.

```
1    startpunkt = Punkt mit kleinster Ordinate
2    i = 0
3    wiederhole
4        P[i] = startpunkt
```

```
5         endpunkt = S[0]
6         wenn startpunkt == endpunkt
7           endpunkt = S[1]
8         fuer j von 1 bis |S|
9           ist (endpunkt == startpunkt) oder (S[j] links von der Geraden
                zwischen startpunkt und endpunkt)
10            endpunkt = S[j]
11        startpunkt = endpunkt
12        i++
13    bis endpunkt == P[0]
```

Listing 1.2: Giftwrapping Pseudocode [**?**]

1.2.2 Beschreibung des Algorithmus

Der Giftwrapping Algorithmus startet beim untersten Punkt der Punktemenge p (kleinster y-Wert). Sollten mehrere Punkte den gleichen y-Wert aufweisen, so wird jener Punkt gewählt, der zusätzlich den kleinsten x-Wert aufweist. Dann werden iterativ alle anderen Punkte mittels des Winkels von p sortiert. Der Punkt mit dem kleinsten Winkel (a) liegt demzufolge auf der konvexen Hülle und kann mit dem Ausgangspunkt verbunden werde. Infolgedessen steht Punkt a als neuer Ausgangspunkt zur Verfügung und es werden erneut alle Winkel von diesem Punkt aus berechnet. So wiederholt sich der Algorithmus, bis am Ende die Konvexe Hülle geformt werden kann.

Die Bezeichnung "Winkel berechnen"kann hierbei etwas irreführend sein. Um den Algorithmus zu vereinfachen werden nicht tatsächliche Winkelwerte berechnet, sondern es werden einfache gerade Linien vom Ausgangspunkt P zu allen anderen Punkten gezogen. Es wird somit für jeden Punkt bzw. jede gerade überprüft, ob sich links von dieser Gerade noch eine weitere befindet. Sollte dies der Fall sein, wird jeweils die linkeste Geradefür die Konvexe Hülle verwendet, solange keine neue gefunden wird, die noch weiter links ist. Der Algorithmus arbeitet sich also entgegen dem Uhrzeigersinn voran, bis die gesamt Konvexe Hülle gebildet ist.

1.2.3 Aufwandsabschätzungen und Testfälle

Theorie

Laut Jarvis ([**?**]) hat der Giftwrapping Algorithmus einen Aufwand von $\mathcal{O}(n(h+1))$, für n Punkt in der Punktmenge und $n \leq h$ auf der Hülle liegende Punkte. Andere Quellen berichten von einer Laufzeit von $\mathcal{O}(nh)$. In den meisten Fällen, wird jedoch nicht einmal dieser Aufwand benötigt werde, da der Algorithmus vereinfacht werden kann, sowohl durch den zuvor beschriebenen Austausch der Winkelberechnungen mit simplen Checks auf welcher Seite ein Punkt liegt, als auch durch den gezielten Ausschluss von Punkten für alle weiteren Berechnungen. Punkte

werden somit ausgeschlossen, wenn sie entweder bereits als auf der konvexen Hülle liegend bestimmt wurden, oder sie in dem Bereich liegen, der vom ersten zum letzten Punkt auf der Hülle eingegrenzt wird. Diese Vereinfachung ist im Pseudocode nicht zu sehen, kann jedoch ganz einfach implementiert werden.

Die Laufzeit des Giftwrapping Algorithmus hängt von der Größe des Outputs ab, was ihn zu einem sogenannten öutput sensitiveÄlgorithmus macht. Wie im vorherigen Absatz erwähnt, hängt der Algorithmus linear von der Anzahl der Punkte auf der Hülle ab. Demnach kann die Laufzeit nur dann, wie bei anderen Algorithmen, gleich schnell oder schneller als $\mathcal{O}(nlog(n))$ sein, wenn die Anzahl der auf der Hülle liegenden Punkte h kleiner ist als $nlog(n)$. Der Idealfall, für die kürzest Mögliche Laufzeit würde eintreten, wenn immer direkt der linkeste Punkt gefunden wird. Der "Best Case"hat somit eine Laufzeit von $\mathcal{O}(n)$. Der "Worst Case"hingegen tritt ein, wenn alle Punkte auf einem Kreis liegen. In diesem Fall ist $h = n$ und somit beträgt die Laufzeit $\mathcal{O}(n^2)$, da jeder Punkt mit jedem Punkt verglichen werden muss.

Messwerte

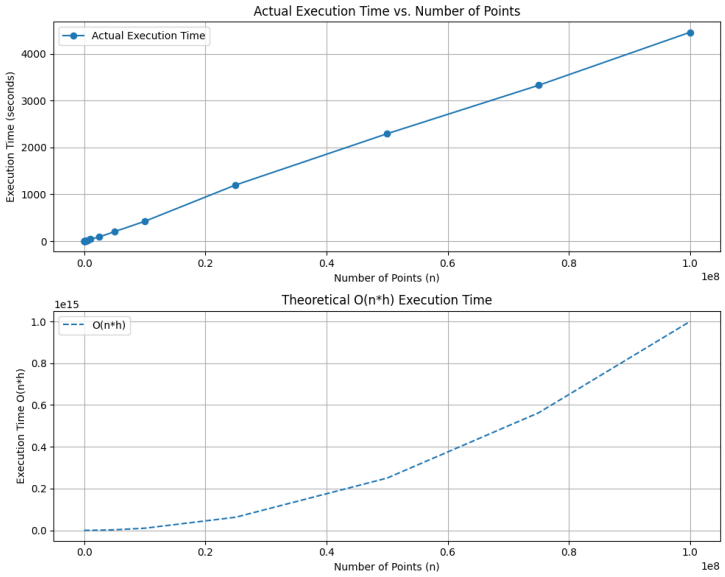

Abbildung 2: Giftwrapping: Vergleich von Messwerten bei verschiedenen Testfällen des Giftwrapping Algorithmus. Der obere Graph zeigt die tatsächliche Laufzeit des Algorithmus mit $\sim \mathcal{O}(n)$. Im unteren Graph ist die theoretische Laufzeit von $\mathcal{O}(n * h)$ abgebildet.

Auch wenn man in Abbildung **??** zu erkennen vermag, dass der Algorithmus immer eine ideale Laufzeit von $\mathcal{O}(n)$ hat, gibt es wie fast immer Ausnahmen. Betrachtet man den Bereich mit sehr wenigen Punkten, so sieht man, dass hier auch teilweise eine stärkere Steigung aufgewiesen wird als linear. Dies hängt natürlich auch mit der bereits zuvor erwähnten schwächeren Leistung von Python im Vergleich zu C++ zusammen, kann aber natürlich auch einen Ursprung im

Algorithmus bzw. seiner Implementierung finden.

Zusammenfassend kann hier festgehalten werden, dass sich unser Algorithmus nahe der idealen Laufzeit von $\mathcal{O}(n)$ bewegt und somit besser ist als theoretisch (im Mittel) erwartet wird. Warum genau das so ist, ist jedoch schwer zu sagen.

Literaturverzeichnis

[1] Gift wrapping. Accessed 29-September-2023 https://de.wikipedia.org/wiki/Gift-Wrapping-Algorithmus.

[2] Quickhull. Accessed 29-September-2023 https://de.wikipedia.org/wiki/QuickHull.

[3] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, dec 1996.

[4] R.A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2(1):18–21, 1973.

Abbildungsverzeichnis