



Datenstrukturen, Algorithmen und Programmierung 2 (DAP2)

Graphalgorithmen

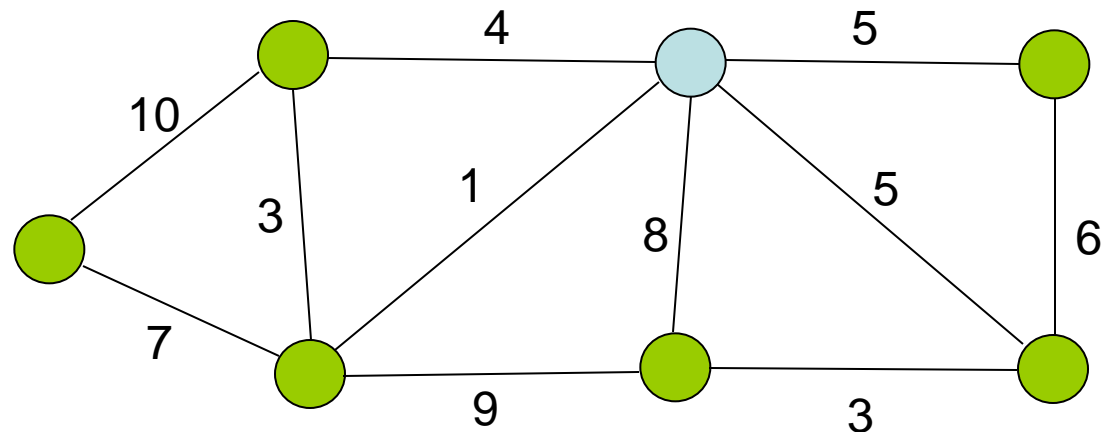
Idee des Algorithmus von Prim

- Verwende generischen Algorithmus
- Nimm immer eine Kante mit minimalem Gewicht, die einen Knoten in Baum A mit einem Knoten verbindet, der nicht in Baum A ist und füge diese zu A hinzu
- Die Kante ist eine leichte Kante, die Baum A mit einem weiteren Knoten verbindet
- Damit ist sie sicher für A

Graphalgorithmen (siehe Vollversion)

Prim(G,r)

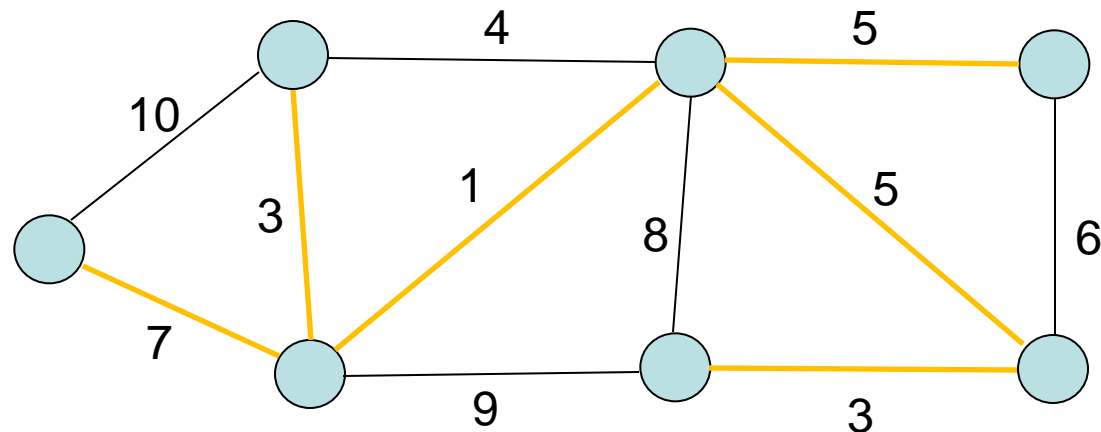
1. $Q \leftarrow V - \{r\}$
2. $A \leftarrow \emptyset$
3. **while** $Q \neq \emptyset$
4. Finde Kante (u,v) mit minimalem Gewicht, die den Schnitt $(Q, V-Q)$ kreuzt, wobei $u \in Q$
5. $Q \leftarrow Q - \{u\}$
6. $A \leftarrow A \cup \{(u,v)\}$
7. **return** A



Graphalgorithmen (siehe Vollversion)

Prim(G,r)

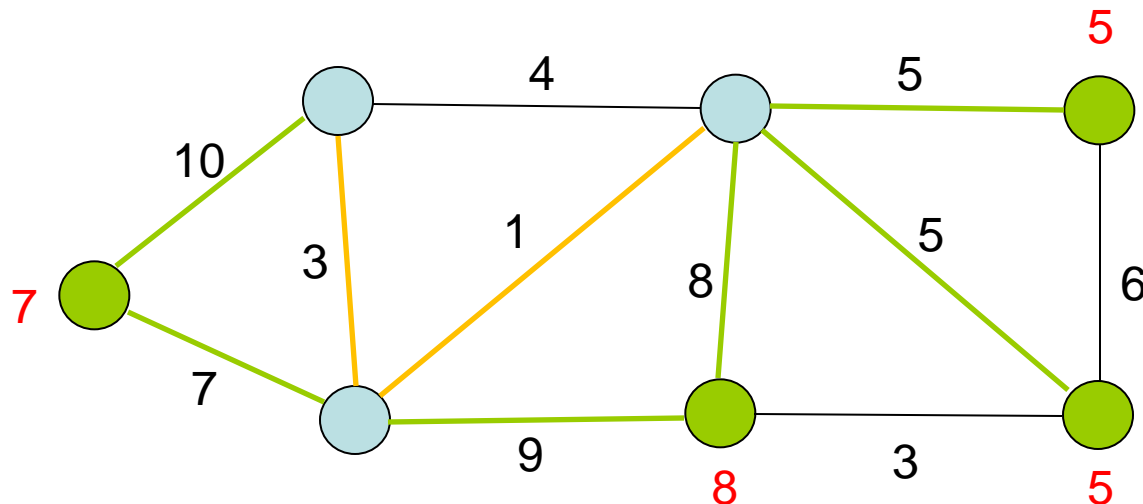
1. $Q \leftarrow V - \{r\}$
2. $A \leftarrow \emptyset$
3. **while** $Q \neq \emptyset$
4. Finde Kante (u,v) mit minimalem Gewicht, die den Schnitt $(Q, V-Q)$ kreuzt, wobei $u \in Q$
5. $Q \leftarrow Q - \{u\}$
6. $A \leftarrow A \cup \{(u,v)\}$
7. **return** A



Graphalgorithmen

Prioritätenschlange

- Alle Knoten, die noch nicht zum Baum gehören, werden in Prioritätenschlange Q abgespeichert
- $key[v]$: minimales Gewicht einer Kante, die v mit Baum verbindet
- $parent[v]$: Vorgänger von v im Baum
- Menge A implizit gegeben durch $A = \{(v, parent[v]) \mid v \in V - \{r\} - Q\}$



Graphalgorithmen

Prim(G,r)

1. $Q \leftarrow V$
2. **For each** vertex $u \in Q$ **do** $key[u] \leftarrow \infty$
3. $key[r] \leftarrow 0, parent[r] \leftarrow nil$
4. **While** $Q \neq \emptyset$ **do**
5. $u \leftarrow \text{Extract-Min}(Q)$
6. **For each** $v \in \text{Adj}[u]$ **do**
7. **If** $v \in Q$ **and** $w(u,v) < key[v]$ **then**
8. $key[v] \leftarrow w(u,v), parent[v] \leftarrow u$

Zu Beginn ist jeder Knoten in der Schlange Q

r bildet die Wurzel des Spannbaums

u ist inzident zu leichter Kante, die Schnitt (Q, V-Q) kreuzt

Gehört v noch nicht zum Baum und könnte über u billiger erreicht werden als bisher bekannt?

Wenn ja, dann ist u neuer Vorgänger von v und $key[v]$ gibt die neuen Kosten für v an

Graphalgorithmen

Implementierung

- Q als binäre Halde

Laufzeit

- Initialisierung von Q: $O(|V|)$
- Ausführung der while-Schleife: $O(|V|)$ -mal
- Extract-Min: $O(\log |V|)$
- Länge aller Adjazenzlisten: $O(|E|)$
- Test $v \in Q$: $O(1)$
- Änderung eines Schlüsselwertes: $O(\log |V|)$
- Gesamtlaufzeit: $O(|V| \log |V| + |E| \log |V|) = O(|E| \log |V|)$

Graphalgorithmen

Satz 76

Der Algorithmus von Prim berechnet einen minimalen Spannbaum eines gewichteten, zusammenhängenden, ungerichteten Graphen in $O(|E| \log |V|)$ Zeit.

Beweis

- Die Laufzeit haben wir bereits analysiert.
- Die Korrektheit folgt aus der Tatsache, dass wir nur sichere Kanten verwenden und dass der Algorithmus einen Baum berechnet

Graphalgorithmen

Graphtraversierung

- Breitensuche, Tiefensuche

Kürzeste Wege

- Breitensuche, Algorithmus von Dijkstra, Algorithmus von Floyd-Warshall

Minimale Spannbäume

- Algorithmen von Kruskal und Prim

Weiterführende Fragestellungen

Grundlegende Fragestellungen

- Kann man zeigen, dass ein Problem nicht effizient(er) gelöst werden kann?
- Wie geht man mit Problemen um, die man nicht effizient lösen kann?

Untere Schranken

Vergleichsbasiertes Sortieren

- Ein Algorithmus ist ein vergleichsbasierter Sortierer, wenn er
- (1) für eine Eingabe von n unterschiedlichen Zahlen a_1, \dots, a_n eine Reihenfolge π berechnet, so dass $a_{\pi(1)} < \dots < a_{\pi(n)}$ gilt und
- (2) wenn sich die Reihenfolge bereits zwingend aus den vom Algorithmus durchgeführten Vergleichen ($<$, $>$) zwischen Eingabeelementen ergibt

Erste Beobachtung

- MergeSort und Heapsort sind vergleichsbasiert
(die Algorithmen führen zwar \leq Operationen durch, wenn die Eingabe jedoch aus unterschiedlichen Zahlen besteht, kann man diese durch $<$ bzw. $>$ ersetzen)

Untere Schranken

Vergleichsbasiertes Sortieren

- Ein Algorithmus ist ein vergleichsbasierter Sortierer, wenn er
- (1) für eine Eingabe von n unterschiedlichen Zahlen a_1, \dots, a_n eine Reihenfolge π berechnet, so dass $a_{\pi(1)} < \dots < a_{\pi(n)}$ gilt und
- (2) wenn sich die Reihenfolge bereits zwingend aus den vom Algorithmus durchgeführten Vergleichen ($<$, $>$) zwischen Eingabeelementen ergibt

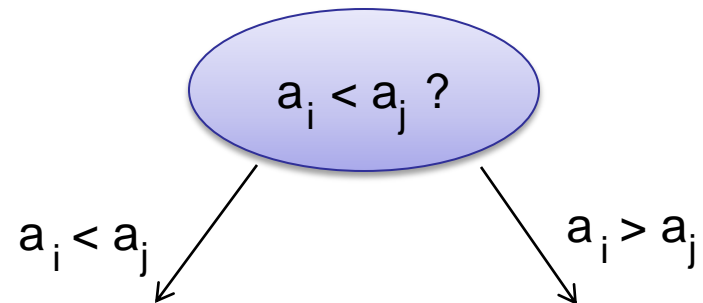
Zweite Beobachtung

- Jeder Vergleich benötigt $\Omega(1)$ Zeit.
- Benötigt ein Algorithmus $f(n)$ Vergleiche, so ist seine Laufzeit $\Omega(f(n))$
- Ziel: Zeige, dass jeder vergleichsbasierte Sortierer $\Omega(n \log n)$ Vergleiche benötigt

Untere Schranken

Baumdarstellung eines vergleichsbasierten Sortierers

- Wir geben Ablauf der Vergleiche an, die der Algorithmus bei Eingabe der Länge n ausführt
- Der Algorithmus führt einen eindeutigen ersten Vergleich aus, dieser wird Wurzel des Baum
- Je nach Ausgang des Vergleichs wird der Algorithmus auf unterschiedliche Weise fortgesetzt
- Das linke Kind eines Knotens entspricht dem Vergleichsausgang $a_i < a_j$, das rechte Kind dem Ausgang $a_i > a_j$
- Jedes Blatt wird mit einer Reihenfolge π bezeichnet

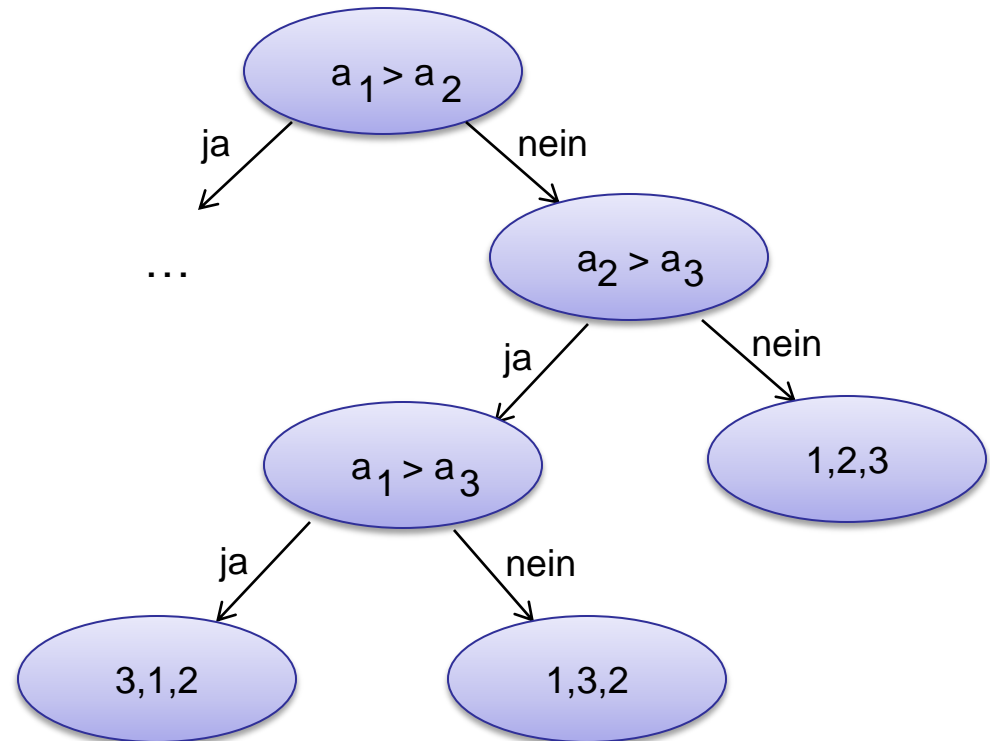


Untere Schranken

Baumdarstellung eines vergleichsbasierten Sortierers (Beispiel InsertionSort, $n=3$)

InsertionSort(Array A)

1. **for** $j \leftarrow 2$ **to** $\text{length}[A]$ **do**
2. $\text{key} \leftarrow A[j]$
3. $i \leftarrow j-1$
4. **while** $i > 0$ and $A[i] > \text{key}$ **do**
5. $A[i+1] \leftarrow A[i]$
6. $i \leftarrow i-1$
7. $A[i+1] \leftarrow \text{key}$



Untere Schranken

Beobachtungen

- Die Tiefe eines Sortierbaums ist eine untere Schranke für die Worst-Case Laufzeit bei Eingabegröße n
- Jeder Sortierbaum ist ein Binärbaum
- Der Sortierbaum hat für jede Ausgabereihenfolge mindestens ein Blatt
- Es gibt $n!$ Ausgabereihenfolgen

Überlegung

- Wie tief ist ein Binärbaum mit $n!$ Blättern mindestens?
- Ein Binärbaum der Tiefe k hat höchstens 2^k Blätter (vollständiger Binärbaum)
- Umgekehrt: Ein Binärbaum mit $n!$ Blättern hat mindestens Tiefe $\log(n!)$

Untere Schranken

Stirling Formel

- Für alle natürlichen Zahlen n gilt:

$$\sqrt{2\pi n} \left(\frac{n}{e}\right)^n \leq n! \leq \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \cdot e^{\frac{1}{12n}}$$

Satz 78

- Ein Binärbaum mit $n!$ Blättern hat Tiefe $\Omega(n \log n)$.

$$\log(n!) \geq \log\left(\sqrt{2\pi n} \left(\frac{n}{e}\right)^n\right) = \log(\sqrt{2\pi n}) + n(\log\left(\frac{n}{e}\right)) = \Omega(n \log n)$$

Untere Schranken

Korollar 79

- Jeder vergleichsbasierte Sortieralgorithmus hat eine Laufzeit von $\Omega(n \log n)$.

Beweis

- Die Baumdarstellung eines vergleichsbasierten Sortieralgorithmus bei Eingabegröße n hat $n!$ Blätter und somit Tiefe $\Omega(n \log n)$.
- Die Tiefe der Baumdarstellung gibt eine untere Schranke für die Worst-Case Laufzeit des Algorithmus, da der Algorithmus bei entsprechender Eingabe alle Vergleiche des längsten Astes durchführt und für jeden Vergleich $O(1)$ Zeit benötigt.
- Somit folgt das Korollar.

Untere Schranken

Wie kann man untere Schranken für andere Probleme zeigen?

- Will zeigen, dass jeder Algorithmus für Problem A Laufzeit $\Omega(f(n))$ hat
- Ich weiss, dass jeder Algorithmus für Problem B Laufzeit $\Omega(f(n))$ hat

Generelle Beweisidee

- Baue Algorithmus C der Problem B löst und dabei einen optimalen Algorithmus für Problem A als Unterprogramm benutzt
- Zeige: Die Laufzeit des Algorithmus ist $o(f(n))$ +Laufzeit für Problem A
- Dann hat jeder Algorithmus für Problem A eine Laufzeit von $\Omega(f(n))$
- (Wäre dies nicht so, dann gäbe es einen Algorithmus mit Laufzeit $o(f(n))$ für Problem A. Somit kann ich Problem B mit Algorithmus C in $o(f(n))$ Zeit lösen. Widerspruch, da die Laufzeit für Problem B $\Omega(f(n))$ ist)

Untere Schranken

Satz 80

Die Berechnung der konvexen Hülle einer Punktmenge von n Punkten in der Ebene benötigt $\Omega(n \log n)$ Zeit.

Beweis

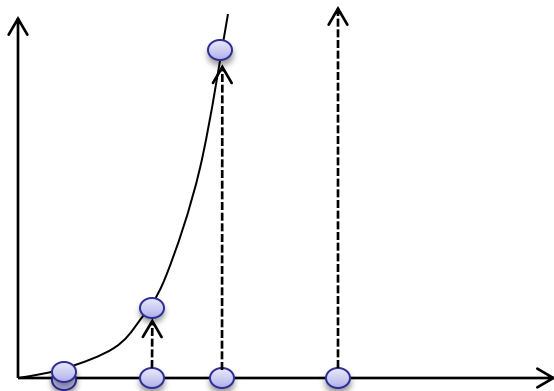
- Annahme: Ich kann die konvexe Hülle von n Punkten in der Ebene in $f(n)=o(n \log n)$ Zeit berechnen
- Dann sei Algorithmus FastHull ein Algorithmus der dies tut
- Zeige: Man kann dann Algorithmus FastSort konstruieren, der in $o(n \log n)$ sortiert
- Dies ist aufgrund unserer unteren Schranke nicht möglich (streng genommen gilt dies natürlich nur für vergleichsbasierte Algorithmen; wir schummeln also hier ein wenig)

Untere Schranken

Beweis

FastSort(A)

1. Initialisiere Feld B für n Punkte
2. **for** $i \leftarrow 1$ **to** n **do**
3. Für Zahl $x=A[i]$ Schreibe Punkt (x,x^2) in Feld B
4. Berechne konvexe Hülle mit FastHull(B)
5. Gib Punkte in der Reihenfolge aus, in der sie auf der Hülle auftreten

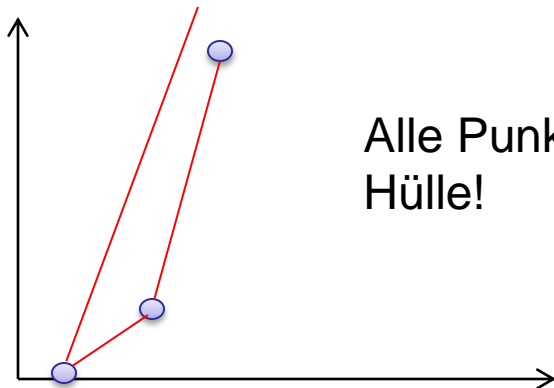


Untere Schranken

Beweis

FastSort(A)

1. Initialisiere Feld B für n Punkte
2. **for** $i \leftarrow 1$ **to** n **do**
3. Für Zahl $x=A[i]$ Schreibe Punkt (x,x^2) in Feld B
4. Berechne konvexe Hülle mit FastHull(B)
5. Gib Punkte in der Reihenfolge aus, in der sie auf der Hülle auftreten



Untere Schranken

Beweis

FastSort(A)

1. Initialisiere Feld B für n Punkte
 2. **for** $i \leftarrow 1$ **to** n **do**
 3. Für Zahl $x=A[i]$ Schreibe Punkt (x,x^2) in Feld B
 4. Berechne konvexe Hülle mit FastHull(B)
 5. Gib Punkte in der Reihenfolge aus, in der sie auf der Hülle auftreten
- Die Laufzeit von FastSort ist $O(n)+f(n)$
 - Hat also FastHull eine Laufzeit von $O(n \log n)$, so hat auch FastSort eine solche Laufzeit
 - Da wir wissen, dass jeder (vergleichsbasierte) Sortieralgorithmus Laufzeit $\Omega(n \log n)$ hat, kann dies nicht sein und FastHull (und jeder andere vergleichsbasierte Algorithmus zur Berechnung der konvexen Hülle) hat Laufzeit $\Omega(n \log n)$.

Untere Schranken

3SUM

- Sei S eine Menge von n Integers. Gibt es 3 unterschiedliche Zahlen in S , die sich zu 0 aufsummieren?

Vermutung

- 3SUM kann nicht in $o(n^2)$ Laufzeit gelöst werden

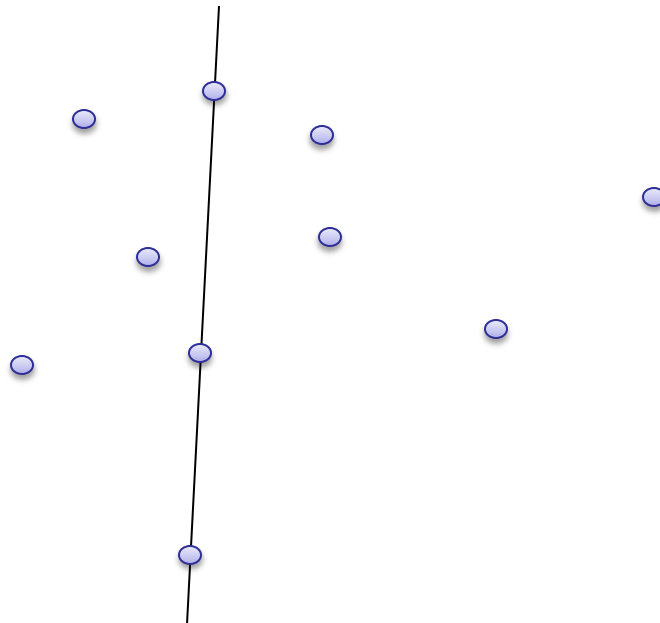
Kommentar

- Die Laufzeit hängt immer auch vom genauen Rechenmodell ab
- In bestimmten Rechenmodellen gibt es Algorithmen, deren Laufzeit etwas besser ist als $O(n^2)$

Untere Schranken

Kolinearitätsproblem

- Seien n Punkte in der Ebene gegeben. Gibt es 3 Punkte, die auf einer Linie liegen?

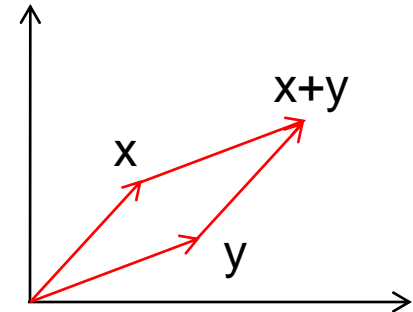


Untere Schranken

Wie testet man Kolinearität von 3 Punkten?

- Seien $x=(x_1, x_2)$ und $y=(y_1, y_2)$ zwei Punkte/Vektoren.
- Dann gibt

$$\det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1$$



- die Fläche des von den beiden Vektoren aufgespannten Parallelograms an
- Sind die Vektoren x und y linear abhängig, so hat das Parallelogramm Fläche 0
- Hat man nun drei Punkte a,b,c, so kann man Kolinearität testen, indem man testet, ob b-a und c-a linear abhängig sind

Untere Schranken

Satz 81

- Sei $f(n)$ eine untere Schranke für die Worst-Case Laufzeit des besten Algorithmus für 3SUM. Dann hat auch das Kolinearitätsproblem eine Laufzeit von $\Omega(f(n))$.

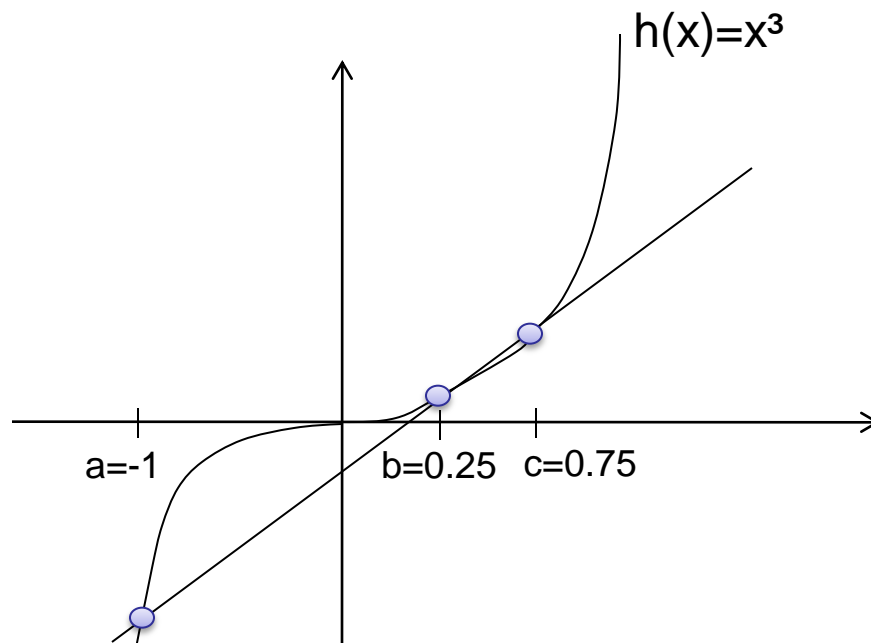
Beweis

- Sei Kolinear ein optimaler Algorithmus für das Kolinearitätsproblem mit Laufzeit $g(n)$.
- Wir entwerfen zunächst einen Algorithmus 3SUM-Fast für 3SUM mit Laufzeit $g(n)+O(n)$, der Algorithmus Kolinear benutzt.

Untere Schranken

3SUM-Fast(S)

1. $P \leftarrow \emptyset$
2. Für jede Zahl $x \in S$ füge Punkt (x, x^3) zu Punktmenge P hinzu
3. **return** Kolinear(P)



Untere Schranken

Behauptung

- 3SUM-Fast löst das 3SUM Problem in Laufzeit $g(n)+O(n)$.

Beweis

- Die Laufzeit folgt sofort, da nur die n Zahlen aus S in n Punkte umgeformt werden müssen und dann Kollinear aufgerufen wird.
- Wir müssen zeigen, dass die Punkte (a,a^3) , (b,b^3) und (c,c^3) genau dann kollinear sind, wenn $a+b+c=0$ ist (d.h. 3SUM erfüllt ist). Dabei sind a,b,c unterschiedliche Zahlen aus S

Untere Schranken

Behauptung

- 3SUM-Fast löst das 3SUM Problem in Laufzeit $g(n)+O(n)$.

Beweis

- Um zu überprüfen, ob die 3 Punkte kollinear sind, rechnen wir die Determinante von $(a-c, a^3-c^3)$, $(b-c, b^3-c^3)$ aus

$$\det \begin{pmatrix} b-a & b^3-a^3 \\ c-a & c^3-a^3 \end{pmatrix} = (b-a)(c^3-a^3) - (c-a)(b^3-a^3)$$
$$= -(a-b)(a-c)(b-c)(a+b+c)$$

- Da a, b und c unterschiedliche Zahlen sind, wird dieses Polynom genau dann 0, wenn $a+b+c=0$ ist.

Untere Schranken

Beweis von Satz 81 (fortgesetzt)

- Wir nehmen nun an, dass $f(n)$ eine untere Schranke für die Laufzeit des besten 3SUM Algorithmus ist. Ist $f(n) = O(n)$, so müssen wir nichts zeigen, da man sich für das Lösen des Kolinearitätsproblems alle Eingabepunkte angucken muss
- Sei also $f(n) = \omega(n)$
- Angenommen, es gibt einen Algorithmus für das Kolinearitätsproblem mit Laufzeit $g(n) = o(f(n))$
- Dann hat Algorithmus 3SUM-Fast eine Laufzeit $g(n) + O(n) = o(f(n))$.
- Widerspruch zur Annahme, dass $f(n)$ eine untere Schranke für die Laufzeit des besten 3SUM Algorithmus ist.
- Also hat jeder Algorithmus für das Kolinearitätsproblem Laufzeit $\Omega(f(n))$

Untere Schranken

Die 1.000.000\$ Frage

- Zeigen Sie, dass es keine Konstante c gibt, so dass das Rucksackproblem in $O(n^c)$ Zeit gelöst werden kann
- Dabei dürfen die Eingabebezahlen exponentiell in n groß sein!

Untere Schranken

Zusammenfassung

- Für einige (sehr wenige) Probleme können wir untere Schranken beweisen
- Die Schranken können vom gewählten Modell abhängen
- Will man zeigen, dass ein Problem A schwer ist und hat man bereits eine untere Schranke für ein anderes Problem B, so kann man einen Algorithmus entwickeln, der Problem A mit Hilfe von Problem B löst und so die Schwierigkeit der Probleme zueinander in Relation setzen
- Es ist i.a. sehr schwer untere Schranken zu zeigen