1 Asymptotische Effizienz

1.1 Wichtige Reihen

Geometrische Reihe

Die geometrische Reihe hat die Form $s_n = \sum_{k=0}^{n} q^k$.

Es gilt:

1. $s_n = \frac{q^{n+1}-1}{q-1}$, falls $q \neq 1$
2. $\lim_{n\to\infty} s_n = \frac{1}{1-q}$, falls $|q| < 1$

Harmonische Reihe

Die harmonische Reihe hat die Form $h_n = \sum_{k=1}^{n} \frac{1}{k}$

Es gilt die Annäherung $h_n \approx \ln n$

1.2 Komplexitätsklassen

$O(f)$

$$g \in O(f) \Leftrightarrow \exists c > 0, n_0 : \forall n \geq n_0 : 0 \leq g(n) \leq c \cdot f(n)$$

alternative Definition:

$$g \in O(f) \Leftrightarrow \limsup_{n\to\infty} \frac{g(n)}{f(n)} = c \geq 0, c \neq \infty$$

$\Omega(f)$

$$g \in \Omega(f) \Leftrightarrow \exists c > 0, n_0 : \forall n \geq n_0 : c \cdot f(n) \leq g(n)$$

alternative Definition:

$$g \in \Omega(f) \Leftrightarrow \liminf_{n\to\infty} \frac{g(n)}{f(n)} = c > 0$$

$\Theta(f)$

$$g \in \Theta(f) \Leftrightarrow \exists c_1, c_2 > 0, n_0 : \forall n \geq n_0 : c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$$

alternative Definition:

$$g \in \Theta(f) \Leftrightarrow \lim_{n\to\infty} \frac{g(n)}{f(n)} = c, 0 < c < \infty$$

$o(f)$

$$g \in o(f) \Leftrightarrow \forall c > 0, n_0 : \forall n \geq n_0 : 0 \leq g(n) < c \cdot f(n)$$

$\omega(f)$

$$g \in \omega(f) \Leftrightarrow \forall c > 0, n_0 : \forall n \geq n_0 : c \cdot f(n) < g(n)$$

1.3 Rekursionsgleichungen

Substitutionsmethode

Methode zum überprüfen des Ergebnisses! Die Lösung muss vorher durch Raten, mehrfaches Einsetzen oder Rekursionbäume gefunden werde.

Methode: Sei $T'(n)$ mutmaßliche Lösung der Gleichung $T(n) = a \cdot T(b) + f(n)$. Einsetzen der mutmaßlichen Lösung $T'(n)$ in $T(n)$, also $T(n) = c \cdot (a \cdot T(b) + f(n))$. Gibt es ein $c > 0$, sodass $T(n) \leq T'(n)$, dann ist die Lösung korrekt.

Erraten der Lösung durch Iteration

Mehrfaches Einsetzen in die Rekursionsgleichung ermöglicht häufig das Erkennen von Mustern und damit eine mögliche Lösung.

Erraten der Lösung durch Rekursionsbäume

1. Baum aufstellen:
   (a) Wurzel: zu analysierende Kosten $T(n)$
   (b) Blätter: Basisfälle
   (c) Die Kosten eines Knotens setzen sich zusammen aus
      i. die nichtrekursiven Kosten (Kosten des Knotens selbst)
      ii. Kosten der Kinder
2. Aufsummieren der Kosten in einer Ebene (Gleichung abhängig von der Höhe h)
3. Gesamtkosten: Summe über alle Ebenen (inklusive Blätter)
4. Beweis durch Substitution!

1.4 Mastertheorem

Grundproblem hat die Form $T(n) = b \cdot T(\frac{n}{c}) + f(n)$ mit $b \geq 1, c > 1$

Sei $E = \log_c b$.

Es gilt:

1. $f(n) \in O(n^{E-\varepsilon})$ für ein $\varepsilon > 0 \Rightarrow T(n) \in \Theta(n^E)$
2. $f(n) \in \Theta(n^E) \Rightarrow T(n) \in \Theta(n^E \cdot \log n)$
3. $f(n) \in \Omega(n^{E+\varepsilon})$ für ein $\varepsilon > 0$ und $b \cdot f(\frac{n}{c}) \leq d \cdot f(n)$ für $d < 1$ und n hinreichend groß $\Rightarrow T(n) \in \Theta(f(n))$

2 Heaps

Eigenschaften

- Binärbaum

- Max-Heap: Schlüssel des Knotens ist größer oder gleich der Schlüssel seiner Kinder (Min-Heap analog)
- alle Ebenen bis auf die unterste sind komplett gefüllt
- Blätter befinden sich auf höchstens zwei Ebenen
- Blätter der untersten Ebene sind linksbündig geordnet

Arrayeinbettung

- Wurzel liegt in $a[0]$
- das linke Kind von $a[i]$ liegt in $a[2i+1]$
- das rechte Kind von $a[i]$ liegt in $a[2i+2]$

heapify

- Voraussetzung: linker und rechter Teilbaum sind Heaps

Methode:

1. Vergleiche Schlüssel des ausgewählten Knotens mit Schlüsseln der Kinder
   (a) falls aktueller Schlüssel maximal: fertig
   (b) ansonsten: tausche aktuellen mit maximalem Knoten, führe Heapify auf dem geänderten Teilbaum aus

buildheap

Führe heapify auf den immer letzten Knoten mit Kind aus, auf den heapify noch nicht ausgeführt wurde.
Man wandert also von rechts unten nach links, dann die nächste Ebene von rechts nach links usw.

3 Suchen

3.1 Lineare Suche

Einfaches Durchlaufen des Arrays von vorne nach hinten.
Zeitkomplexität: $O(n)$

3.2 Bilineare Suche

Wie lineare Suche, beginne aber gleichzeitig Suche von hinten. Gleiche Komplexität!

3.3 Binäre Suche

Suche im sortierten Binärbaum.
Zeitkomplexität: $O(\log n)$

4 Sortieren

4.1 Dutch-National-Flag

Es gibt vier Regionen:

- rote Elemente (Position 0 bis r)
- weiße Elemente (Position $r+1$ bis $u-1$)
- blaue Elemente (Position b bis n)
- unbekannte Elemente (Position u bis $b-1$)

Am Anfang ist das gesamte Array unbekannt, d.h. $r=-1, u=0, b=n+1$.

Methode:

So lange $u < b$: Nimm Element u, untersuche die Farbe, falls:

1. rot: tausche an die Position r, setze $r=r+1, u=u+1$
2. weiß: setze $u=u+1$
3. blau: tausche an die Position $b-1$, setze $b=b-1$

Eigenschaften:

- $B(n)=A(n)=W(n) \in \Theta(n)$
- nicht stabil
- in-place

4.2 Insertionsort

1. sortiert von vorne nach hinten
2. 1. Teil des Arrays ist sortiert, 2. ist unsortiert

Methode:

Nehme das aktuelle Element, füge es an die richtige Stelle im sortierten Teil ein, verschiebe alle folgenden Einträge des sortierten Arrays nach hinten.

Eigenschaften:

- $B(n) \in \Theta(n)$
- $A(n)=W(n) \in \Theta(n^2)$
- stabil
- in-place

4.3 Selectionsort

Methode:

1. Finde minimales Element des unsortierten Teils
2. tausche es an die erste Stelle
3. rekursives selectionsort auf den unsortierten Teil

Eigenschaften

- $B(n) = A(n) = W(n) \in \Theta(n^2)$
- nicht stabil (es gibt stabile Varianten)
- in-place

4.4 Mergesort

Methode:

1. Teile das Array in zwei Teile
2. Rufe rekursiv mergesort auf, außer Arraylänge beträgt 1
3. Füge Arrays zusammen:
   (a) Vergleiche die ersten Elemente der beiden Teilarrays und füge das kleinere in das Ergebnisarray ein
   (b) weiter bis beide Teilarrays vollständig eingefügt wurden
4. Schritt (3) rekursiv bis zur obersten Ebene (bis das gesamte Array zusammengefügt ist)

Eigenschaften

- $B(n) = A(n) = W(n) \in \Theta(n \cdot \log n)$
- stabil
- Speicherbedarf: $\Theta(n)$

4.5 Heapsort

Heapsort sortiert einen Heap durch rekursiven Aufruf von heapify.

Methode:

1. buildHeap
2. tausche letztes und erstes Element des Arrays
3. heapify des Baums bis auf das letzte (gerade getauschte) Element
4. weiter bei (2), bis nur noch ein Knoten im Baum ist

Eigenschaften

- $B(n) = A(n) = W(n) = O(n \cdot \log n)$
- nicht stabil
- in-place

4.6 Quicksort

Methode:

1. Partitioniere das Array, sodass
   - alle Elemente links des Pivotelements kleiner als das Pivotelement sind
   - alle Elemente rechts des Pivotelements größer als das Pivotelement sind
2. Rufe quicksort rekursiv auf die beiden Partitionen auf

Partitionierung:

Es gibt drei Bereiche: „$<$ Pivot“, „$\geq$ Pivot“, „ungeprüft“

1. Verschiebe linke Grenze nach rechts, solange das zusätzliche Element „$<$ Pivot“ ist
2. Verschiebe rechte Grenze nach links, solange das zusätzliche Element „$\geq$ Pivot“ ist
3. vertausche die beiden Elemente
4. weiter bei (1), bis „ungeprüft“ leer ist

Eigenschaften

- $B(n) = A(n) \in \Theta(n \cdot \log n)$
- $W(n) \in \Theta(n^2)$
- Platzbedarf: $\Theta(\log n)$
- nicht stabil (gibt stabile Varianten)

5 Binäre Suchbäume

Ein binärer Suchbaum (BST) ist ein Binärbaum mit Schlüsseln, wobei der Schlüssel eines Knotens

- mindestens so groß ist, wie jeder Schlüssel im linken Teilbaum
- höchstens so groß ist, wie jeder Schlüssel im rechten Teilbaum

5.1 Sortieren

Eine Inorder-Traversierung ergibt alle Werte in sortierter Reihenfolge.

5.2 Suche

Traversiere Baum.

1. falls gesuchter Schlüssel gleich dem aktuellen Schlüssel: fertig.
2. falls gesuchter Schlüssel kleiner als aktueller Schlüssel: durchsuche linken Teilbaum.
3. falls gesuchter Schlüssel größer als aktueller Schlüssel: durchsuche rechten Teilbaum.

5.3 Einfügen

1. Suche geeignete Stelle, wie Suche, außer dass bei gleichem Schlüsselwert weiter abgestiegen wird.
2. Füge an gefundenen Knoten ohne Kinder an

5.4 Minimum

Das Minimum eines BST-Baums ist das Element an der linkesten Stelle (Traversierung nach links bis kein Kindknoten mehr vorhanden ist).

Komplexität: $\Theta(h)$

5.5 Nachfolger finden

Gesucht ist der Nachfolger des Knotens k bei der Inorder-Traversierung.

1. rechter Teilbaum existiert: Nachfolger ist der kleinste Knoten im rechten Teilbaum
2. sonst: Nachfolger ist der jüngste Vorfahre, dessen linker Teilbaum den Knoten k enthält.

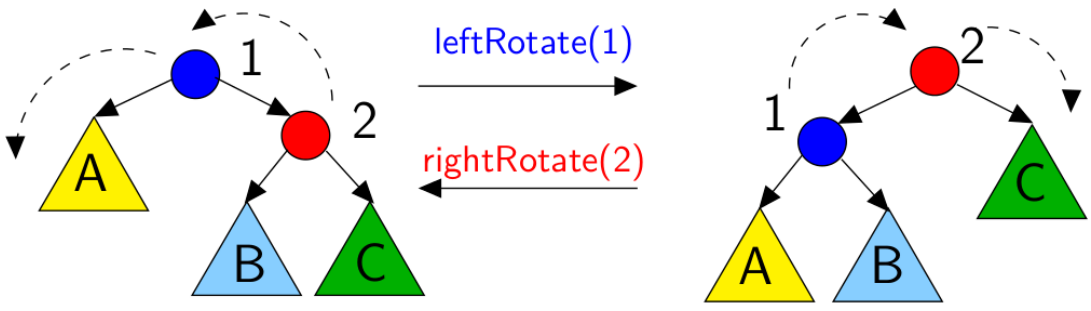
Komplexität: $\Theta(h)$

5.6 Löschen

1. Knoten hat keine Kinder: Lösche den Knoten
2. Knoten hat ein Kind: Schneide den Knoten aus (Vater und Kind miteinander verbinden)
3. Knoten hat zwei Kinder: Finde Nachfolger, lösche Nachfolger aus dem Baum, ersetze Knoten durch Nachfolger

Komplexität: $\Theta(h)$

5.7 Rotationen



5.8 AVL-Bäume

In AVL-Bäumen haben alle Knoten ein zusätzliches Datenfeld, in der die Höhe des Teilbaums gespeichert wird. Nach jeder kritischen Operation wird der Baum durch Rotationen ausbalanciert.

6 Rot-Schwarz-Bäume

Eigenschaften:

1. Jeder Knoten ist entweder rot oder schwarz.
2. Die Wurzel ist schwarz.
3. Jedes Blatt (also null) ist schwarz.
4. Ein roter Knoten hat nur schwarze Kinder.
5. Für jeden Knoten enthalten alle Pfade, die an diesem Knoten starten und in einem Blatt enden, die gleiche Anzahl schwarzer Knoten. Die Schwarz-Höhe jeden Knotens ist eindeutig.

6.1 Schwarz-Höhe

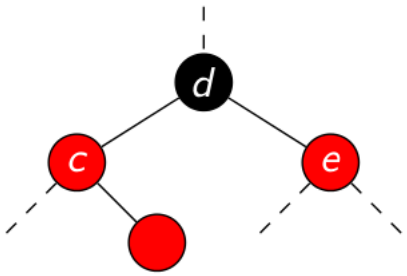
Die Schwarz-Höhe ist die Anzahl schwarzer Knoten bis zu einem Blatt. Der Knoten selbst wird nicht mitgezählt.

6.2 Einfügen

Komplexität: $O(\log n)$

Der neu eingefügte Knoten ist immer rot. Ist der Vaterknoten schwarz, so sind wir fertig. Ist er rot, so muss eine Rot-Rot-Verletzung aufgelöst werden.

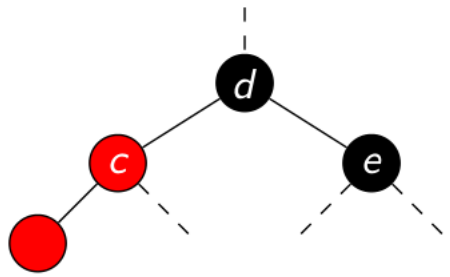
Fall 1: Onkel ist rot



Lösung:

1. Färbe c und e schwarz
2. Färbe d rot
3. Löse eventuelle Rot-Rot-Verletzung zwischen d und dem Vaterknoten von d.

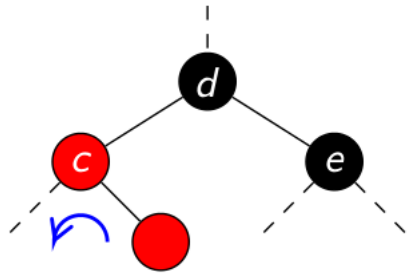
Fall 2: Onkel ist schwarz, neuer Knoten liegt außen



Lösung:

1. rotiere um d nach rechts
2. färbe d rot
3. färbe c schwarz

Fall 3: Onkel ist schwarz, neuer Knoten liegt innen



Überführe zu Fall 2 durch Linksrotation um c.

6.3 Löschen

Komplexität: $O(\log n)$

Methode:

1. Finde den Nachfolger
2. Entferne den Nachfolger
3. Behebe entstandene Farbverletzung
4. Ersetze den Knoten durch seinen Nachfolger, übernehme die Farbe des Knotens

7 Hashing

7.1 Counting Sort

Komplexität: $\Theta(n)$

Histogramm

Jeder Eintrag gibt an, wie häufig der Index des Eintrages im Eingabe-Array vorkommt. Gibt es z.B. die 0 zweimal in der Eingabe, so ist $h[0] = 2$.

Positionen

Die Positionen berechnen sich aus dem Histogramm durch simples aufaddieren, also $p[i] = \sum_{k=0}^{i} h[k]$.

Methode:

1. Erstelle Histogramm
2. Erstelle Positionen anhand des Histogramms
3. Kopiere Eingabe- in Ausgabearray anhand der Positionen, beginne mit letztem Arrayeintrag $i = e[n]$ der Eingabe:
   (a) Dekrementiere $p[i]$
   (b) Kopiere i nach $a[p[i]]$

7.2 Verkettung

Idee: Bei Kollision werden alle Schlüssel mit dem gleichen Hashwert in einer Liste gespeichert. Jedes neue Element mit dem gleichen Hashwert werden an den Anfang der Liste geschrieben.

Füllgrad

Der Füllgrad gibt an, wie lange die Listen bei Hashing mit Verkettung im Schnitt sind.

Der Füllgrad ist definiert durch

$$\alpha(n, m) = \frac{n}{m}$$

wobei

- n der Anzahl der möglichen Schlüssel
- m der Anzahl der Hashtabellenpositionen

entspricht.

Unter der Annahme, dass m proportional zu n ist, gilt:

$$\alpha(n, m) = \frac{n}{m} \in \frac{O(m)}{m} = O(1) \tag{1}$$

Komplexität

Suchen $A(n) \in \Theta(1 + \alpha)$

Einfügen $A(n) \in \Theta(1)$

Löschen $A(n) \in \Theta(1 + \alpha)$

Sortieren $A(n) \in O(n)$ (mit (1) vorausgesetzt)

7.3 Hashingfunktionen

Divisionsmethode

$$h(k) = k \mod m$$

m sollte prim und nicht zu nah an einer Zweierpotenz gewählt werden.

Multiplikationsmethode

$$h(k) = \lfloor m \cdot (k \cdot c \mod 1) \rfloor$$

mit $c \in (0, 1)$.

Knuth empfiehlt $c \approx \frac{\sqrt{5}-1}{2} \approx 0.62$

Vorgehen:

1. Wähle $m = 2^p$, $s \in (0, 2^w)$, $c = \frac{s}{2^w}$, wobei w der Schlüssellänge entspricht
2. Berechne $k \cdot s$
3. Teile durch 2^w, verwende nur Nachkommastellen
4. Multipliziere mit 2^p und verwende nur den ganzzahligen Anteil

Universelles Hashing

Gegeben ist eine Menge H von Hashingfunktionen. Es wird zufällig eine Funktion aus H gewählt.

H ist universell $\Leftrightarrow$ der Anteil der Funktionen aus H, so dass k und k' kollidieren, ist $\frac{|H|}{m}$.

H ist universell $\Leftrightarrow$ Wahrscheinlichkeit einer Kollision von k und k' ist $\frac{1}{|H|} \cdot \frac{|H|}{m} = \frac{1}{m}$

7.4 Offene Adressierung

Grundprinzip

Bei einer Kollision wird der Schlüssel an anderer Position der Hashtabelle gespeichert. Die Position ist abhängig von der Sondierungsmethode.

Löschen

Bei offener Adressierung werden gelöschte Einträge als gelöscht markiert, jedoch nicht gelöscht. Hintergrund: Bei der Suche muss weitergesucht (d.h. der nächste Eintrag mit gleichem Hash betrachtet) werden. Ist der Eintrag leer, so bricht die Suche dagegen ab.

Einfügen

Einfügen betrachtet als gelöscht markierte Einträge als leer und überschreibt diese.

Lineares Sondieren

$$h(k,i) = (h'(k) + i) \mod m$$

wobei

- k ist der Schlüssel
- i ist der Index der Sondierungssequenz
- h' ist eine Hashfunktion

Nachteil: Lange Folgen von nichtleeren Einträgen werden noch länger (Clustering)

Quadratisches Sondieren

$$h(k,i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \mod m$$

wobei

- k ist der Schlüssel
- i ist der Index der Sondierungssequenz
- h' ist eine Hashfunktion
- $c_1, c_2 \neq 0$ sind Konstanten

Clustering wird vermieden, jedoch tritt sekundäres Clustering auf: $h(k,0) = h(k',0) \Rightarrow h(k,i) = h(k',i)\ \forall i$

Doppeltes Hashing

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m$$

wobei h_1 und h_2 Hashfunktionen sind.

Doppeltes Hashing führt zu einer besseren Verteilung. Sind h_2 und m relativ prim, wird die gesamte Hashtabelle abgesucht.

Komplexität

erfolglose Suche $A(n) \in O(\frac{1}{1-\alpha})$

erfolgreiche Suche $A(n) \in O(\frac{1}{\alpha} \cdot \ln \frac{1}{1-\alpha})$

8 Graphen

8.1 Grundlagen

Im folgenden: $G = (V, E), |V| = n, |E| = m$.

Transponieren

Sei $G = (V, E)$ ein Graph. Dann ist der transponierte Graph $G^T = (V, E')$ mit $(v, w) \in E' \Leftrightarrow (w, v) \in E$

Es werden also die Kanten umgedreht, die Knoten bleiben gleich.

Adjazenzmatrix

Ein Graph kann durch eine $n \times n$-Matrix A ($n = |V|$) dargestellt werden. Es gilt: $a_{ij} = 1 \Leftrightarrow (v_i, v_j) \in E$, ansonsten $a_{ij} = 0$.

Platzbedarf: $\Theta(n^2)$

Adjazenzliste

Ein Graph kann als Array von Adjazenzlisten dargestellt werden. Der i-te Arrayeintrag enthält alle Kanten von G, die von v_i ausgehen.

Platzbedarf: $\Theta(n + m)$

Kondensationsgraph

Fasst man alle Knoten einer starken Zusammenhangskomponente S_i zu einem Knoten zusammen, so erhält man einen Kondensationsgraph $G \downarrow = (V', E')$.

Für die Kanten des Kondensationsgraphen gilt:

$$(V_i, V_j) \in E' \Leftrightarrow i \neq j, \exists v \in V_i, w \in V_j : (v, w) \in E$$

Zwei Knoten sind im Kondensationgraph also genau dann verbunden, wenn im ursprünglichen Graph zwei Knoten beider Zusammenhangskomponenten verbunden waren.

Es gilt: $(G \downarrow)^T = (G^T) \downarrow$

8.2 Sharirs Algorithmus

Algorithmus zum Finden der starken Zusammenhangskomponenten

Methode:

1. normale DFS auf G, schreibe alle Knoten beim Abschließen (schwarz färben) auf einen Stack
2. DFS auf G^T, beginne bei jedem Durchlauf mit dem obersten noch weißen Knoten auf dem Stack, speicher diesen als Leiter.

Komplexität

Zeit: $\Theta(n + m)$

Speicher: $\Theta(n)$

8.3 Kritische-Pfad-Analyse

Methode:

Finde den Pfad mit maximaler Länge vom Start- zum Endpunkt.

8.4 Greedy-Algorithmen

Greedy-Algorithmen sind solche Algorithmen, die in jedem Schritt die momentan beste Lösung wählen.

Greedy ist dann geeignet, wenn die optimale Lösung aus optimalen Teilproblemen besteht.

8.5 Algorithmus von Prim

Algorithmus zur Berechnung von minimalen Spnnbäumen

Methode:

1. Starte mit einem beliebigen Knoten
2. Füge den Knoten dem Baum hinzu, markiere alle adjazenten Knoten mit RAND
3. Wähle unter allen Kanten vom Baum zu Randknoten die mit minimalem Gewicht
4. Weiter mit (2), bis keine Randknoten mehr vorhanden sind

Komplexität

Laufzeit $T(n) \in \Omega(m), T(n) \in O(n^2)$

Speicher $\Theta(n)$

8.6 Bellman-Ford

Algorithmus zur Berechnung von kürzestem Pfad (SSSP)

Methode

1. Initialisiere alle Entferungen mit ∞, den Startknoten mit 0
2. Für alle Kanten $(v, w) \in E$ mit Gewicht $W(v, w)$: Falls $d[w] > d[v] + W(v, w)$, setze $d[w] = d[v] + W(v, w)$
3. Führe Schritt 2 durch, bis keine Änderungen mehr vorkommen, max. jedoch $|V| - 1$ mal.

Komplexität

$T(n) \in O(n^3)$

8.7 Djikstra

SSSP

Methode

1. Wähle den Startknoten
2. Füge den gewählten Knoten mit entsprechender Kante dem Baum hinzu, markiere alle adjazenten Knoten mit RAND
3. Wähle den Knoten mit minimaler Entfernung vom Startknoten
4. Weiter bei (2), bis keine Randknoten mehr vorhanden sind

Komplexität

Zeit $W(n) \in \Theta(|V|^2)$, $T(n) \in \Omega(|E|)$

Platz $O(|V|)$

8.8 Algorithmus von Warshall

Berechnung der transitiven Hülle

Transitive Hülle

Der Graph wird um solche Kanten ergänzt, sodass er transitiv und reflexiv ist.

Methode

1. Für alle $v \in V$: füge (v, v) den Kanten hinzu
2. Für alle $v \in V$: Falls $(i, v) \in E$ und $(v, j) \in E$: Füge (i, j) hinzu

Komplexität

Zeit $\Theta(n^3)$
Platz $\Theta(n^2)$

8.9 Algorithmus von Floyd

All Pairs Shortest Path (APSP)

Methode

wie Warshall, betrachte jedoch die Distanz:

1. Für alle $v \in V$: setze $d[v, v] = 0$
2. Für alle $v, w \in V$ mit $(v, w) \notin E$: Setze $d[v, w] = \infty$
3. Für alle $v \in V$: Falls $d[i, v] + d[v, j] < d[i, j]$: Setze $d[i, j] = d[i, v] + d[v, j]$

Komplexiät

Zeit $\Theta(n^3)$
Platz $\Theta(n^2)$

Erweiterung

Die Pfade können gespeichert werden, indem zu jedem Knoten der Vorgänger gespeichert wird.

8.10 Flussnetzwerk

c ist die Kapazitätsfunktion, f ein Fluss. s die Quelle (source), t die Senke (target). Seien $u, v \in V$. Es gilt:
Beschränkung $f(u, v) \leq c(u, v)$
Asymmetrie $f(u, v) = -f(v, u)$
Flusserhaltung $\forall u \in V \setminus \{s, t\} : \sum_{v \in V} f(u, v) = 0$

Beschriftung

Wie gewichteter Graph, Kanten sind mit $f(u,v)/c(u,v)$ beschriftet, falls $f(u,v) > 0$, sonst $c(u,v)$

Flüsse zwischen Knotenmengen

Sei $x \in X, y \in Y, X \subseteq V, Y \subseteq V$.

1. $f(x,Y) = \sum_{y \in Y} f(x,y)$
2. $f(X,y) = \sum_{x \in X} f(x,y)$
3. $f(X,Y) = \sum_{x \in X} \sum_{y \in Y} f(x,y)$

Eigenschaften

1. $f(X,X) = 0$
2. $f(X,Y) = -f(Y,X)$
3. $F(X \cup Y, Z) = f(X,Z) + f(Y,Z) - f(X \cap Y, Z)$

Ford-Fulkerson-Methode

1. Wähle beliebigen Pfad $s \to t$, maximiere den Fluss durch diesen Pfad
2. Berechne augmentierenden Pfad

9 Dynamisches Programmieren

Methode

1. Analysiere das Problem, Problem ist geeignet für DP, wenn:
   (a) sich Teilprobleme überlappen
   (b) rekursive Abhängigkeiten zwischen Teilproblemen bestehen
2. Stelle die Rekursionsgleichung (top-down) auf
3. Löse die Rekursionsgleichung bottom-up (z.B. mit Hilfe einer Matrix)

10 Algorithmische Geometrie

10.1 Winkelbestimmung

Lage von x bezüglich y

$$\det(x,y) \begin{cases} > 0 \\ = 0 \\ < 0 \end{cases} \Rightarrow y \text{ liegt } \begin{cases} \text{links von } x \\ \text{auf } x \\ \text{rechts von } x \end{cases}$$

Knick eines Streckenzugs

Sei (p, q, r) ein Streckenzug. Setze $a := q - p, b := r - q$

Es gilt:

$$\det(a,b) \begin{cases} > 0 \\ = 0 \\ < 0 \end{cases} \Rightarrow \angle pqr \begin{cases} > 180° & \text{Linksknick} \\ = 180° & \\ < 180° & \text{Rechtsknick} \end{cases}$$

10.2 Konvexe Hülle

Vorgehen

1. Beginne mit dem Punkt $p_0 = (x, y)$ mit minimalem y, falls nicht eindeutig minimalem x.
2. Sortiere alle Punkte von p_0 ausgehend nach zunehmendem Winkel
   (a) Berechne für jeden Punkt p: $i_p = \frac{x_p - x_{p_0}}{y_p - y_{p_0}}$
   (b) Sortiere alle Punkte nach i_p absteigend
3. Lege p_0, p_1 auf den Stack
4. Falls $i < n$: Lege den nächsten Punkt p_i auf den Stack
5. Berechne mit Knick eines Streckenzugs den Knick von $p_{i-2}p_{i-1}p_i$
   (a) falls Rechtsknick: entferne p_{i-1} vom Stack, weiter bei (5)
   (b) falls Linksknick: weiter bei (4)