Algorithmen und Datenstrukturen - Zusammenfassung

Fiona Pichler

January 7, 2019

Fehler gefunden? $\Longrightarrow$(pichlerf@student.ethz.ch)

Contents

1 Kostenmodelle

1.1 O-Notation

Die obere Schranke des Wachstums einer Funktion.
Wenn g ∈ O(f) ist, wächst sie höchstens so schnell wie f.
$\lim_{x\to\infty} \frac{g(n)}{f(n)} = 0$, dann ist g ∈ O(f)

1.2 Ω-Notation

Die untere Schranke des Wachstums einer Funktion. Wenn g ∈ Ω(f) ist, so wächst sie mind. so schnell wie f.
$\lim_{x\to\infty} \frac{g(n)}{f(n)} = \infty$ dann ist g ∈ Ω(f)

1.3 Θ-Notation

Wenn g ∈ Ω(f) ∩ O(f) ist, so ist sie in Θ(f). Sie wächst also gleich schnell wie f.
$\lim_{x\to\infty} \frac{g(n)}{f(n)} = C > 0 \in \mathbb{C}$ Dann ist g ∈ Ω(f)

2 Induktionsbeweis

Induktionshypothese: Hier steht was wir beweisen wollen.

Induktionsanfang Hier zeigen wir, dass unsere Hypothese für mind. ein n stimmt.

Induktionsschritt Hier zeigen wir, dass wenn die Hypothese für n stimmt, sie auch für n+1 stimmt.

3 Teleskopieren

Wird für rekursive Formeln genutzt. Wir setzen die Rekursionsformel so lange ein, bis wir ein Muster erkennen und die Formel nicht-rekursiv schreiben können.

4 Suchverfahren

4.1 Binäre Suche

Wir beginnen in der Mitte und beginnen wieder in der Mitte von der rechten Menge, wenn x grösser ist, oder in der Links falls x kleiner ist.

Binäre Suche — BINARY-SEARCH$(A = (A[1], \ldots, A[n]), b)$

```
1 left ← 1; right ← n                                  ▷ Initialer Suchbereich
2 while left ≤ right do
3     middle ← ⌊(left+right)/2⌋
4     if A[middle] = b then return middle              ▷ Element gefunden
5     else if A[middle] > b then right ← middle−1      ▷ Suche links weiter
6     else left ← middle+1                             ▷ Suche rechts weiter
7 return "Nicht vorhanden"
```

Laufzeit: O(log n)

4.2 Interpolations Suche

Funktioniert gleich wie die Binäre Suche, doch statt immer
middle = [left + $\frac{1}{2}$(right-left)]
nehmen wir p = $\frac{b-A[left]}{A[right]-[left]} \in [0,1]$
Laufzeit: O(log log n), Ω(n)

4.3 Exponentielle Suche

Die rechte Grenze ist r=1. Diese verdoppeln wir bis $r > boderr > n$. wir suchen dann mit der Binären Suche auf dem Array von 1,..,min(r,n)

EXPONENTIELLE SUCHE

```
EXPONENTIAL-SEARCH(A = (A[1], ..., A[n]), b)
1 r ← 1                                          ▷ Initiale rechte Grenze
2 while r ≤ n and b > A[r]                       ▷ Finde rechte Grenze
3     r ← 2 · r                                  ▷ Verdopple rechte Grenze
4 return BINARY-SEARCH((A_1, ..., A[min(r, n)]), b) ▷ Weiter mit binärer Suche
```

diese Suche ist gut, wenn p $\ll$ n.
Laufzeit: O(log p), Ω(n)

4.4 Untere Schranke

Der Beweis, dass jedes Vergleichs basierte Suchverfahren auf sortierten Arrays eine Laufzeit von Ω(log n) hat. Beweisbar mit einem Entscheidungsbaum.

4.5 Suchen in unsortierten Arrays

Es wird linear gesucht und einfach jeder Eintrag geprüft.
Laufzeit: Θ(n)

5 Sortierverfahren

5.1 Bubblesort

Die ersten zwei Elemente werden verglichen, vertauscht, dann die nächsten zwei, so viele Male, bis das Array sortiert ist.

BUBBLESORT

```
BUBBLESORT(A = (A[1], ..., A[n]))
1 for j ← 1 to n − 1 do
2     for i ← 1 to n − 1 do
3         if A[i] > A[i + 1] then Vertausche A[i] und A[i + 1].
```

Laufzeit:
worst case: $\Theta(n^2)$
best case: O(n)

5.2 Sortieren durch Auswahl (Selectionsort)

Wir suchen das grösste Element und tauschen es mit dem Hintersten.
Wir suchen das Element auf dem Array ohne die hinterste Stelle, und vertauschen es mit dem Zweithintersten...

SortierenDurchAuswahl

```
SelectionSort(A)
1 for k ← n downto 2 do          ▷ k Schlüssel sind noch nicht sortiert
2     Berechne den Index i des maximalen Schlüssels in A[1], ..., A[k].
3     Vertausche A[i] und A[k].  ▷ A[k] enthält (n − k + 1)-grössten Schl.
```

Laufzeit:
worst case: $\Theta(n^2)$
best case: $O(n^2)$

5.3 Sortieren durch Einfügen (Insertionsort)

Wir fügen Elemente in ein bereits Sortiertes Array ein.
Das erste Element ist schon sortiert, das zweite kommt dazu, usw.
Dank Binärer Suche finden wir den Platz für das einzufügende Element schnell.

SortierenDurchEinfügen

```
InsertionSort(A = (A[1], ..., A[n]))
1 for k ← 1 to n − 1 do          ▷ k Schlüssel sind bereits sortiert
2     Benutze binäre Suche auf A[1], ..., A[k], um die Position i
      zu finden, an die A[k + 1] eingefügt werden muss.
3     b ← A[k + 1]               ▷ Speichere A[k + 1] in b zwischen
4     for j ← k downto i do      ▷ Verschiebe A[i], ..., A[k]
5         A[j + 1] ← A[j]        ▷ auf A[i + 1], ..., A[k + 1]
6     A[i] ← b                   ▷ Speichere A[k + 1] an A[i]
```

Laufzeit:
worst case: $\Theta(n^2)$
best case: $O(n)$

5.4 Heapsort

Der Heap kann als binärer Baum visualisiert werden.

Wiederherstellung des Heaps

```
RESTORE-HEAP-CONDITION(A, i, m)
 1 while 2 · i ≤ m do                          ▷ A[i] hat linken Nachfolger
 2     j ← 2 · i                               ▷ A[j] ist linker Nachfolger
 3     if j + 1 ≤ m then                       ▷ A[i] hat rechten Nachfolger
 4         if A[j] < A[j + 1] then j ← j + 1   ▷ A[j] ist grösserer Nachfolger
 5     if A[i] ≥ A[j] then STOP                ▷ Heap-Bedingung erfüllt
 6     Vertausche A[i] und A[j]                ▷ Reparatur
 7     i ← j                                   ▷ Weiter mit Nachfolger
```

Heapsort

```
HEAPSORT(A)
 1 for i ← ⌊n/2⌋ downto 1 do
 2     RESTORE-HEAP-CONDITION(A, i, n)
 3 for m ← n downto 2 do
 4     Vertausche A[1] und A[m]
 5     RESTORE-HEAP-CONDITION(A, 1, m − 1)
```

Laufzeit:
worst case: O(n log n)
best case: O(n log n)

5.5 Mergesort

Wir verschmelzen zwei sortierte Teilfolgen.

Verschmelzen der Teilfolgen

```
MERGE(A, left, middle, right)
 1 B ← new Array[right-left+1]                 ▷ Hilfsarray zum Verschmelzen
 2 i ← left; j ← middle+1; k ← 1               ▷ Zeiger zum Durchlaufen
 3 ▷ Wiederhole, solange beide Teilfolgen noch nicht erschöpft sind
   while (i ≤ middle) and (j ≤ right) do
 4     if A[i] ≤ A[j] then B[k] ← A[i]; i ← i + 1
 5     else B[k] ← A[j]; j ← j + 1
 6     k ← k + 1
 7 ▷ Falls die erste Teilfolge noch nicht erschöpft ist, hänge sie hinten an
   while i ≤ middle do B[k] ← A[i]; i ← i + 1; k ← k + 1
 8 ▷ Falls die zweite Teilfolge noch nicht erschöpft ist, hänge sie hinten an
   while j ≤ right do B[k] ← A[j]; j ← j + 1; k ← k + 1
 9 ▷ Kopiere Inhalte von B nach A zurück
   for k ← left to right do A[k] ← B[k − left + 1]
```

Laufzeit:
worst case: Θ(r-l)
best case: Θ(r-l)
Speicherplatz:
Θ(n)

5.5.1 Rekursives 2-Wege-Mergesort

Wir implementieren Mergesort rekursiv.

Rekursives Mergesort

```
MERGESORT(A, left, right)
 1 if left < right then
 2     middle ← ⌊(left + right)/2⌋        ▷ Mittlere Position
 3     MERGESORT(A, left, middle)          ▷ Sortiere vordere Hälfte von A
 4     MERGESORT(A, middle + 1, right)     ▷ Sortiere hintere Hälfte von A
 5     MERGE(A, left, middle, right)       ▷ Verschmilz Teilfolgen
```

Laufzeit:
worst case: Θ(n log n)
best case: Θ(n log n)

5.5.2 Reines 2-Wege-Mergesort

Statt rekursiv, sortieren wir im Array selbst. Wir starten mit Teilfolgen der Länge 1 und mergen weiter.

Reines 2-Wege-Mergesort

```
STRAIGHTMERGESORT(A)
 1 length ← 1                            ▷ Länge bereits sortierter Teilfolgen
 2 while length < n do                   ▷ Verschmilz Folgen d. Länge length
 3     right ← 0                         ▷ A[1], ..., A[right] ist abgearbeitet
 4     while right+length < n do         ▷ Es gibt noch mind. zwei Teilfolgen
 5         left ← right+1                ▷ Linker Rand der ersten Teilfolge
 6         middle ← left+length−1        ▷ Rechter Rand der ersten Teilfolge
 7         right ← min(middle+length, n) ▷ Rechter Rand der zweite Teilfolge
 8         MERGE(A, left, middle, right) ▷ Verschmilz beide Teilfolgen
 9     length ← length·2                 ▷ Verdoppelte Länge der Teilfolgen
```

Laufzeit

Wie rekursives Mergesort führt reines 2-Wege-Mergesort immer $\Theta(n \log n)$ viele Schlüsselvergleiche und -bewegungen aus.

Laufzeit:
worst case: Θ(n log n)
best case: Θ(n log n)

5.5.3 Natürliches 2-Wege-Mergesort

Wir suchen die längsten bereits sortierten Teilfolgen und mergen diese.

Natürliches 2-Wege-Mergesort

```
NaturalMergesort(A)
 1 repeat
 2     right ← 0                    ▷ Elemente bis A[right] sind bearbeitet
 3     while right < n do           ▷ Finde und verschmilz die nächsten Runs
 4         left ← right+1           ▷ Linker Rand des ersten Runs
 5         middle ← left            ▷ A[left], ..., A[middle] ist bereits sortiert
 6         while (middle < n) and A[middle + 1] ≥ A[middle] do
 7             middle ← middle+1    ▷ Ersten Run um ein Element vergrössern
 8         if middle < n then       ▷ Es gibt einen zweiten Run
 9             right ← middle+1     ▷ Rechter Rand des zweiten Runs
10             while (right < n) and A[right + 1] ≥ A[right] do
11                 right ← right+1  ▷ Zweiten Run um ein Element vergrössern
12             Merge(A, left, middle, right)
13         else right ← n           ▷ Es gibt keinen zweiten Run
14 until left = 1
```

Laufzeit:
worst case: Θ(n log n)
best case: Θ(n log n)

5.6 Quicksort

Wir nehmen ein Pivotelement p,(zum Beispiel das letzte) und durchsuchen das array von links nach rechts nach einem Element das grösser ist als p, dann suchen wir von recht nach links eines, das kleiner ist als p. Wir vertauschen die beiden, und fahren wie zuvor fort, bis das grössere Element weiter rechts steht als das kleinere. Wir tauschen das grössere mit p und das Array ist sortiert. Gegeüber mergesort, braucht Quicksort weniger Speicherplatz.

Aufteilungsschritt

```
Partition(A, l, r)
 1 i = l
 2 j = r − 1
 3 p = A[r]
 4 repeat
 5     while i < r and A[i] < p do i = i + 1
 6     while j > l and A[j] > p do j = j − 1
 7     if i < j then Vertausche A[i] und A[j]
 8 until i ≥ j
 9 Tausche A[i] und A[r]
10 return i
```

Hauptalgorithmus

```
Quicksort(A, l, r)
1 if l < r then                  ▷ Array enthält mehr als einen Schlüssel
2     k = Partition(A, l, r)     ▷ Führe Aufteilung durch
3     Quicksort(A, l, k − 1)     ▷ Sortiere linke Teilfolge rekursiv
4     Quicksort(A, k + 1, r)     ▷ Sortiere rechte Teilfolge rekursiv
```

Laufzeit:
worst case: $\Theta(n^2)$
best case: O(n log n)
Speicherplatz: O(log n)

5.7 eine untere Schranke für vergleichsbasierte Sortierverfahren

Der Beweis, dass vergleichsbasierte Sortierverfahren nicht besser werden können als Ω(n log n)

6 Dynamische Programmierung

Definition der DP-Tabelle
Welche Dimension hat die Tabelle? Was ist die Bedeutung jedes Eintrags?

Berechnung eines Eintrags
Welche Einträge hängen nicht von anderen Einträgen ab?
Wie berechnet sich ein Eintrag aus den anderen Einträgen?

Berechnungsreihenfolge
In welcher Reihenfolge müssen die Einträge berechet werden?

Auslesen der Lösung
Wie lässt sich die Lösung aus der Tabelle finden?

6.1 Subset Sum Problem

Problemstellung
Wir haben eine Menge von n Elementen und müssen die so aufteilen, dass wir zwei Mengen mit der Selben Summe bekommen.
z:=$\frac{1}{2}\sum_{i=1}^{n} a_i$ Wir suchen eine Auswahl I der Menge 1,...,n, sodass
$\sum_{i\in I} a_i$ = z.

Definition der DP-Tabelle
Zweidimensionale Tabelle (0 bis n) x (0 bis z)
Jeder Eintrag enthält entweder "wahr" oder "falsch"
T[k,s] gibt an, ob es eine Auswahl I der ersten k Gegenstände1,...,k gibt, sodass $\sum_{i\in I} a_i$ = s gilt.

Berechnung eines Eintrags
T[0,0] = wahr
T[0,s] = falsch $\forall s \in [1, ..., z]$

$$T[k,s] = \begin{cases} T[k-1,s] & falls\ s < a_k \\ T[k-1,s] \vee T[k-1, s-a_k] & falls\ s \geq a_k \end{cases}$$

Berechnungsreihenfolge
Sie Einträge werden mit aufsteigendem k und bei gleichem k mit aufsteigendem s berechnet.

Auslesen der Lösung
Eine Auswahl der Grösse n mit $\sum_{i\in I} a_i$ = z exisstiert,
wenn T[n,z]= wahr.
um I zu erhalten starten wir bei T[k,s].
wenn T[k,s]=T[k-1,s], dann T[k,s]$\notin$ I
wir fahren mit T[k-1,s] fort.
wenn $T[k,s] = T[k-1, s-a_k]$,dann$T[k,s] \in$ I
wir fahren mit $T[k-1, s-a_k]$ fort, bis k=0.

6.2 Rucksackproblem

Problemstellung
Wir haben eine Menge von n Elementen. Jedes Element i hat einen Nutzen v_i und ein Gewicht w_i. Wir haben eine Gewichtsgrenze W.
Wir suchen eine Summe S, sodass $\sum_{i \in S} w_i \leq W.$ und $\sum_{i \in S} v_i =$ maximal.

Definition der DP-Tabelle

Zweidimensionale Tabelle maxWert(0 bis i) x (0 bis w) maxWert[i,w] gibt den maximal erreichbaren Nutzen an, wenn nur von den Elementen 0,...,i ausgewählt werden kann und das Gewicht höchstens w beträgt.

Berechnung eines Eintrags

maxWert[0,w] $\leftarrow 0$
Für i ¿ 0
maxWert[i,w] $\leftarrow \begin{cases} max\{maxWert[i-1,w], maxWert[i-1,w-w_i]+v_i\} & falls\ w \geq w_i \\ maxWert[i-1,w] & sonst \end{cases}$

Berechnungsreihenfolge
Wir berechnen die Einträge nach aufsteigenden i von (0,...,n) bei gleichem i mit aufsteigendem w (0,...W)

Auslesen der Lösung
maxWert[n,W] speichert den maximal erreichbaren Nutzen einer Auswahl. Die Auswahl kann wie in SubsetSum-Problem gelöst werden.

7 Abstrakte Datentypen

7.1 Stapel/Stack

bei einem Stapel hat man immer nur Zugriff auf das oberste Element, und legt neue Objekte oben drauf.

Operationen

PUSH(x,S): legt das Objekt x zuoberst auf den Stack S.
POP(S): gibt das oberste Objekt von Stack S zurück und entfernt es.
TOP(S): gibt das oberste Objekt von Stack S nur zurück.
ISEMPTY(S): gibt "true" zurück wenn S leer ist, sonst "false"
EMPTYSTACK: Liefert einen leeren Stack zurück

7.2 Schlange/Queue

bei einer Schlange hat man immer nur Zugriff auf das vorderste Objekt, aber legt neue Objekte immer zuhinterst ab.

Operationen

ENQUEUE(x,Q): fügt das Objekt x hinten an Q an.
DEQUEUE(Q): gibt das vorderste Objekt zurück und entfernt es
FRONT(Q): gibt das vorderste Objekt zurück
ISEMPTY(Q): gibt "true" zurück falls Q leer ist, sonst "false"
EMPTYQUEUE: Liefert eine leere Schlange zurück

7.3 Priority Queue

eine Priority Queue ist wie eine Queue, aber die Objekte haben eine Priorität, nach der wir die Objekte einfügen und auslesen können.

Operationen

INSERT(x,p,Q): fügt das Objekt x hinten an Q an.
EXTRACTMAX(Q): gibt das Objekt mit höchster Priorität zurück und entfernt es
FRONT(Q): gibt das vorderste Objekt zurück
ISEMPTY(Q): gibt "true" zurück falls Q leer ist, sonst "false"
EMPTYQUEUE: Liefert eine leere Schlange zurück

7.4 Multistack

wie ein Stack aber mit einer Operation mehr

Operationen

PUSH(x,S): legt das Objekt x zuoberst auf den Stack S.
POP(S): gibt das oberste Objekt von Stack S zurück und entfernt es.
TOP(S): gibt das oberste Objekt von Stack S nur zurück.
ISEMPTY(S): gibt "true" zurück wenn S leer ist, sonst "false"
EMPTYSTACK: Liefert einen leeren Stack zurück
MULTIPOP(k,S): gibt die obersten k Objekte des Stacks zurück und löscht diese.

7.5 Wörterbuch/Dictionary

besteht aus Schlüsseln in einem geordnetem Universum

Operationen

INSERT(k,D): fügt den Schlüssel k ins Universum D ein. Fehlermeldung falls k bereits vorhanden.
DELETE(k,D): Löscht den Schlüssel k im Universum D. Fehlermeldung falls k nicht vorhanden.
SEARCH(k,D): gibt "true" zurück falls k in D vorhanden ist, "false" sonst.

8 Hashing

Bei einem Wörterbuch, könnte man ein Objekt mit k 100 im Array T an stelle T[100] speichern, doch das wäre eine riesen Platzverschwendung, wenn unser Wörterbuch nur 3 Worte/Objekte enthält.
Hashing wandelt diese Idee ab.
Wir nehmen ein Array T[m], für k$\in$ 0,..,m-1] speichern wir das Objekt in T[k],
für k $\geq$ m stellen wir uns vor, die es gäbe noch mehr Tabellen und berechnen für k die Position wenn sich die Tabelle vervielfachen würde.
Es wurde erwiesen, dass wenn m als Primzahl gewählt wird, die Wahrscheinlichkeit einer Kollision der k minimal wird.

8.1 Universelles Hashing

eine universelle Hashklasse ist eine Menge H $\subseteq h \in H : hK \to 0, ..., m-1$ von Hashfunktionen, so dass:
$\forall k_1, k_2 \in K$ mit $k_1 \neq k_2 : \frac{|h \in H : h(k_1) = h(k_2)|}{|H|} \leq \frac{1}{m}$
die Hashfunktion wird zufällig aus H gewählt.

8.2 Hashverfahren mit Verkettung der Überläufer

Wir speichern am Platz T[k] alle Elemente mit gleicher Hashadresse als Liste.

8.3 Offenes Hashing

Statt eine Liste zu erstellen, Speichern wir die Objekte direkt in der Hashtabelle mit Hilfe einer Sondierungsfunktion s(j,k).

9 Selbstanordnung

Wir sortieren unsere Listen ständig um, um die Suchzeit für häufig gebrauchte Objekte zu verkleinern.

Frequency Count:
Wir zählen für jeden Schlüssel die Häufigkeit des gebrauchs.
Nach jeder Suche werden die Schlüssel erneut nach der Häufigkeit geordnet.

Transpose:
Bei jedem Zugriff auf einen Schlüssel bewegen wir ihn eine Position nach vorne.

Move-to-Front(MTF):
Bei jedem Zugriff auf einen Schlüssel setzen wir ihn an den Anfang der Liste.

10 Natürliche Suchbäume

Ein Natürlicher Suchbaum ist ein binärer Suchbaum. Falls die Objekte sortiert eingefügt werden, haben wir jedoch einfach eine lineare Liste.

11 AVL-Bäume

Nach Adelson-Velski und Landis benannt. Bei einem AVL-Baum darf der linke Teilbaum höchstens 1 höher sein als der rechte und umgekehrt. Objekte werden wie beim netürlichen Suchbaum eingefügt. Danach wird geprüft ob der Baum die AVL-Bedingeung noch erfüllt und nötigenfalls umbalanciert.

12 Graphenbegriffe

12.1 Gerichteter Graph

Besteht aus einer Menge von Knoten $V = \{v_1, ..v_n\}$ und einer Menge von Kanten $E \subseteq V \times V$ jede Kante hat eine Richtung.

12.2 Ungerichteter Graph

Besteht aus einer Menge von Knoten $V = \{v_1, ..v_n\}$ und einer Menge von Kanten $E \subseteq \{\{u, v\} | u, v \in V\}$ Kante u,v=v,u

12.3 Schleife

Eine Kante (v,v) im Falle gerichteter Graphen oder v im Falle ungerichteter Graphen

12.4 Multigraph

Ein Graph der gewisse Kanten mehrmals enthält

12.5 Vollständiger Graph

Ein Graph der alle möglichen Kanten enthält

12.6 Bipartiter Graph

Wenn die Knoten in zwei disjunkte Mengen U und W aufgeteilt werden enthält jede Kanten einen Knoten aus U und einen aus W.

12.7 Gewichteter Graph

Besitzt eine Kantengewichtsfunktion c: E $\rightarrow \mathbb{R}$.
c(e) ist das Gewicht der Kante e.

12.8 Adjazenz

Ein Knoten w heisst adjazent zu einem Knoten v, falls E die Kante (v, w) oder $\{v, w\}$ enthält.

12.9 Vorgänger

Für gerichtete Graphen G = (V,E) Vorgängermenge $N^-(v) := \{u \in V | (u, v) \in E\}$

12.10 Nachfolger

Für gerichtete Graphen G = (V,E)
Nachfolgermenge $N^+(v) := \{w \in V | (v, w) \in E\}$

12.11 Eingangsgrad

$\deg^-(v)$ ist die Kardinalität der Vorgängermenge.

12.12 Ausgangsgrad

$\deg^+(v)$ ist die Kardinalität der Nachfolgermenge.

12.13 Nachbarschaft

Für ungerichtete Graphen G = (V,E) die Menge $N(v) := \{w \in V | \{v, w\} \in E\}$ heisst Nachbarschaft von v.

12.14 Grad

Für ungerichtete Graphen G = (V,E)
deg(v)
Eine Schleife an v erhöht den Grad um 2!

12.15 Weg

Eine Sequenz von Knoten, welche alle durch Kanten verbunden sind.

12.16 Pfad

Ein Weg der keinen Knoten mehrfachbenutzt. Wenn der Pfad im Knoten s startet und im Knoten t endet heisst er st-Pfad.

12.17 Zusammenhang

Ein ungerichteter Graph in dem für jedes Paar zweier Knoten ein Weg existiert heisst zusammenhängend.

12.18 Zyklus

Ein Weg in dem End- und Startknoten übereinstimmen.

12.19 Kreis

Ein Zyklus der keinen Knoten mehr als einmal benutzt.

12.20 Kreisfreier Graph

Ein Graph ohne Kreise.

12.21 Topologische Sortierung

Ein gerichteter Graph G = (V,E) besitzt genau dann eine topologische Sortierung wenn er kreisfrei ist. Eine Topologische Sortierung, ist eine Sortieren, bei der alle Kanten in die selbe Richtung zeigen.

13 Graphen durchlaufen

13.1 Tiefensuche

Wir verfolgen einen Pfad und markieren die Knoten als besucht, bis wir nur noch besuchte Knoten finden, dann gehen wir einen Knoten zurück und versuchen den Pfad dort zu erweitern. Falls dies nicht funktioniert starten wir die Suche bei einem noch nicht besuchten Knoten erneut.

Laufzeit: $\Theta(|V|+|E|)$
zusätzlicher Platzbedarf: $\Theta(|E|)$

13.2 Breitensuche

Wir nehmen einen Knoten und besuchen alle seine Nachfolger dann deren Nachfolger usw. Dann starten wir bei noch unbesuchten Knoten nochmals, bis alle Knoten besucht wurden.

Laufzeit: $\Theta(|V|+|E|)$
zusätzlicher Platzbedarf: $\Theta(|V|)$

13.3 Kürzeste Wege

13.3.1 Dijkstra

1. Weise allen Knoten die beiden Eigenschaften "Distanz" und "Vorgänger" zu. Initialisiere die Distanz im Startknoten mit 0 und in allen anderen Knoten mit ∞.
2. Solange es noch unbesuchte Knoten gibt, wähle darunter denjenigen mit minimaler Distanz aus und
   1. Speichere, dass dieser Knoten schon besucht wurde.
   2. Berechne für alle noch unbesuchten Nachbarknoten die Summe des jeweiligen Kantengewichtes und der Distanz im aktuellen Knoten.
   3. Ist dieser Wert für einen Knoten kleiner als die dort gespeicherte Distanz, aktualisiere sie und setze den aktuellen Knoten als Vorgänger.

Laufzeit: $O((|E|)\log |V|)$

13.3.2 Ford

Berechnet alle kürzesten Wege, so oft, bis sich kein Wert mehr verändert.

13.3.3 Bellmann

Berechne sie mit einem Dynamischen Program.

13.3.4 Bellmann-Ford

Algorithmus von Bellman und Ford

Bellman-Ford$(G = (V, E), s)$

```
1  for each v ∈ V\{s} do                          ▷ Initialisiere für alle Knoten die
2      d[v] ← ∞; p[v] ← null                      ▷ Distanz zu s sowie Vorgänger
3  d[s] ← 0; p[s] ← null                          ▷ Initialisierung des Startknotens
4  for i ← 1, 2, ..., |V| − 1 do                  ▷ Wiederhole |V| − 1 Mal
5      for each (u, v) ∈ E do                     ▷ Iteriere über alle Kanten (u, v)
6          if d[v] > d[u] + w((u, v)) then        ▷ Relaxiere Kante (u, v)
7              d[v] ← d[u] + w((u, v))            ▷ Berechne obere Schranke
8              p[v] ← u                           ▷ Speichere u als Vorgänger von v
9  for each (u, v) ∈ E do                         ▷ Prüfe, ob eine weitere Kante
10     if d[u] + w((u, v)) < d[v] then            ▷ relaxiert werden kann
11         Melde Kreis mit negativem Gewicht
```

Laufzeit: O(|E||V|)

13.3.5 Floyd und Warshall

ist ein dynamisches Program:
$d^i_{u,v}$ u,v ∈ V und i ∈ $\{0, ..., n\}$
i steht für die Anzahl Zwischenknoten. speichert die Länge eines kürzesten Weges von u nach v.

$d^0_{u,u} = 0$

Für u ≠ v: $d^0_{u,v} \leftarrow \begin{cases} = w((u,v)) & falls\ (u,v) \in E \\ = \infty & sonst \end{cases}$

$d^i_{u,v} = \min(d^{i-1}_{u,v}, d^{i-1}_{u,v_i} + d^{i-1}_{v_i,v})$

Laufzeit: O($|V|^3$)

Rekonstruktion der Wege
Zu jedem Paar zweier Knoten (u,v) wird der Nachfolger succ[u,v] von u auf einem kürzesten Weg von u nach v gespeichert.
succ[u,v] ← 0 für alle u,v ∈ V mit (u,v) ∉ E. succ[u,v] ← v für alle u,v ∈ V mit (u,v) ∈ E.
falls $d^i_{u,v} \leftarrow d^{i-1}_{u,v_i} + d^{i-1}_{v_i,v}$)
dann wird succ[u,v] ← v für alle u,v ∈ V mit (u,v) ∈ E.succ[u,v_i].

13.3.6 Johnson

Wir fügen einen neuen Knoten NEU in V ein.
Wir verbinden NEU mit allen anderen Knoten mit Kanten mit Gewicht 0.
Wir führen für jeden Knoten v ∈ V eine Höhe $h(v)$ ein.
Wir definieren eine neue Kantengewichtsfunktion
$\hat{w}((u,v)) := w((u,v)) + h(u) - h(v)$
Die neue Kantengewichtsfunktion wird so gewählt dass $\hat{w}$ stets nicht-negativ ist.
Wir benutzen den Algorithmus von Bellman und Ford ausgehend von NEU zur berechnung aller Höhen $h(v)$.
Für jeden Knoten u ∈ V\NEU starten wir Dijkstras Algorithmus um die Wege zu alen Knoten v∈ V des Graphen zu berechnen. Danach müssen nur noch die gefundenen Distanzen berechnet werden.

Laufzeit: O($|V|^2$ *log* |V||E||V|)

13.4 Minimale Spannbäume

13.4.1 Prim

Wähle Kanten mit minimalen Gewicht so aus, dass alle Knoten zusammenhängend werden. beginne mit beliebigen Startknoten s füge immer die günstigste Kante hinzu, die einen neuen Knoten zur Zusammenhangskomponente von s hinzufügt.

Laufzeit: O(E log V)

13.4.2 Kruskal

wähle n-1 Kanten mit minimalem Gewicht so aus, dass kein Kreis entsteht. Beginne mit leerer Kantenmenge und füge immer die günstigste ante hinzu, die keinen Kreis verursacht.

Laufzeit: O(E log V)

13.4.3 Union-Find

ist eine Datenstruktur.

make(V): erstelle Datenstruktur für leeren Graphen auf V
find(u): gib eindeutigen Namen der ZHK von u zurück
union(u,v): füge Kanteu,v hinzu / vereinige ZHKs von u und v

14 Andere Algorithmen

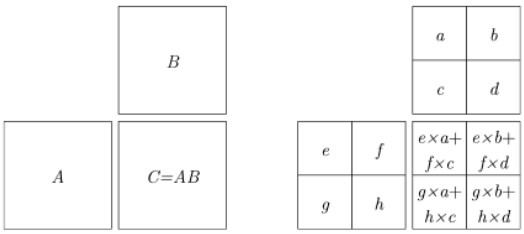
14.0.1 Karatsuba und Ofman

Beschleunigung der Schul-Multiplikationsmethode.
Multiplizieren wir zwei n-stellige Zahlen (wobei wir vereinfachend annehmen, dass $n = 2^k$ eine Zweierpotenz ist), dann können wir diese als
$(10^{n/2}a + b) \cdot (10^{n/2}c + d)$
$= 10^n a \cdot c + 10^{n/2}a \cdot c + 10^{n/2}b \cdot d + b \cdot d + 10^{n/2}(a - b) \cdot (d - c)$
schreiben und rekursiv berechnen.

14.0.2 Strassen

Beschleunigung der Matrixmultiplikation

Wir teilen die Matrizen A und B in je 4 Teilmatrizen auf.



Strassen bemerkte, dass es reicht nur 7 Produkte zu berechnen.
$(f - h) \times (c + d)$, $(e + f) \times d$, $h \times (c - a)$,
$(e - g) \times (a + b)$, $(g + h) \times a$, $e \times (b - d)$,
$(e + h) \times (a + d)$

aus diesen Produkten lassen sich alle Teile von C berechnen.

z.B. $(e \times b) + (f \times d) = ((e + f) \times d) + (e \times (b - d))$

14.0.3 Auswahlproblem, Quicksort

Wir suchen das i-kleinste Element im Array A[n].
Wir wählen ein Pivot.
Wir zählen die Elemente kleiner als p (=) r viele.
Wir teilen A und sortieren dabei die kleineren Elemente in die untere Hälfte ein.
i=r return p.
i<r suche i-tes links
i>r suche (i-r)-tes rechts

Laufzeit:
worst case: $\Theta(n^2)$
average case: $\Theta(n)$

14.0.4 Median der Mediane

Wähle den optimalen Pivot.

Betrachte das Array in 5-er Gruppen.
Bestimme den Median in jeder Gruppe.
A' = Array der Gruppenmediane (Länge$[\frac{n}{5}]$)
Bestimme Median von A' = Pivot p

Doch in der Praxis funktioniert die zufällige Pivotwahl ganz gut.

14.1 Greedy-Algorithmen

Wählen schrittweise den zu diesem Zeitpunkt optimalen Folgezustand aus, was nicht unbedingt das optimale Ergebnis ist.