Algorithmen

1. Sonstiges

1.1 Analyse

1.1.1 Induktion

1.1.2 Stochastik

1.1.3 amortisierte Analyse

1.2 Algorithmen

1.2.1 Median-Bestimmung

1.2.2 Match-count

1.2.3 induzierte Teilstrings

1.2.4 kanonische Linearisierung zyklischer Strings

1.2.5 größte Teilungsfreie Teilmenge

1.2.6 k-Wiederholdungen

1.2.7 längstes disjunktes maximales Paar

2. Notationen

2.1 Klein Omega-Notation

wie groß omega nur asymptotisch gebunden, s. auch o-Notation

2.2 o-Notation

wie groß O nur asymptotisch gebunden, gut zu merken über $2n=o(n^2)$, aber $2n^2$ ungleich $o(n^2)$, weil die Ungleichung 0 kleiner gleich f(n) kleiner c*g(n) für alle c ab einem bestimmten n gelten muss, und ausserdem kleiner und nicht kleiner gleich sein muss!

2.3 Groß Omega-Notation

asymptotische untere Grenze: f(n) ist asymptotisch oberhalb von c*g(n), also f(n)=gross-omega(g(n))

2.4 O-Notation

asymptotische obere Grenze: f(n) ist asymptotisch unterhalb von c*g(n), also f(n)=O(g(n))

2.5 Etha-Notation oder so

asymptotisch eng gebunden, d.h. f(n) ist asymptotisch genau zwischen c1*g(n) und c2*g(n), also ist f(n)=Etha(g(n))

3. Sortierung

3.1 Quicksort

3.1.1 normal

3.1.2 randomisiert

3.2 Counting-Sort

3.3 Radix-Sort

4. Hashing

4.1 geschlossenes Adressieren

4.1.1 Divisionsmethode

4.1.2 Multiplikationsmethode

4.2 offenes Adressieren

Wenn Platz besetzt ist, neuen Platz suchen

4.2.1 Lineares Sondieren

4.2.2 Quadratisches Sondieren

4.2.3 Double Hashing

4.3 universelles Hashing

4.4 voll-dynamisches Hashing

dynamisches Anpassen der Hashtabelle => O(1+n/m) wie bisher, nur dass n/m generell kleiner ist, hinzu kommt Aufwand für Tabellenvergrößerung: amortisierte Analyse => besser: O(1)

5. String-Matching

5.1 klassisch

5.1.1 klassisch

- **Z-Agorithmus**
- **Bayer-Moore**
- **Bad-character-Regel**
- **Strong-Good-Suffix-Regel**
- **Knuth-Morris-Pratte**
- **Aho-Corasick-Algorithmus**
- **mit Wildcards**
- **zweidimensional**

5.1.2 seminumerisch

- **Shift-AND**
- **Karp-Rabin-Fingerprint**

5.2 Suffix-Bäume

5.2.1 Konstruktion

- **Ukkonen**
- **3 Regeln**
- **Suffix-Links**
- **skip and count**

5.2.2 Anwendung

- ***Exaktes Matching***
- **implizite Kantenlabels**
- **Musterdatenbank**
- **Longest-Common-Substring**
- **All-Pairs-Präfix-Suffix-Matching**
- ***maximale Palindrome***
- **lca-Anfragen (longest-common-ancestor)**
- **lce (longest common extension)**
  » Siehe auch: : lca-Anfragen (longest-common-ancestor)

5.3 Suffix-Arrays

5.3.1 exaktes Matching

5.3.2 approximatives Matching

6. dynamische Programmierung

im Gegensatz zu divide-and-conquer (top-down): bottom-up

6.1 Global-Alignment (approximatives Matching)
Needleman-Wunsch-Algorithmus

6.2 Local-Alignment (optimales Alignment)
Smith-Waterman

6.3 All-Pairs-Shortest-Paths
Floyd-Warschall

» Siehe auch: : Floyd-Warshall (all-pairs shortest paths)

7. Graphen

7.1 Dijkstra (single source-shortest paths)
[Cormen] S. 527

7.1.1 Gegeben
ein gerichteter und gewichteter Graph und einer seiner Knoten s von dem aus wir alle kürzesten Wege zu allen anderen Knoten berechnen wollen

7.1.2 Gesucht
von s (Source) aus alle kürzesten Wege zu allen anderen Knoten

7.1.3 Lösung
Trick ist: Man berechnet sukzessive kürzeste Wege zu den nächsten Knoten und benutzt diese als gesicherte Ausgangsbasis für die Berechnung der nächstnächsten Knoten...

Von s ausgehend betrachtet man die direkt verbundenen Knoten (Pfeil von s zu den entsprechenden Knoten) und schreibt zu den Knoten jeweils die Länge des Weges=Größe des Gewichtes der Kante! Von diesen Knoten u nehme man jenen mit dem kürzesten Wege und entfernt diesen aus der Knotenmenge (wie auch schon s entfernt wurde). Dann relaxiert man von Neuem, aber diesmal die Kanten von u ausgehend und trägt wieder die Länge des Weges (summiert mit der Länge von s nach u) ein und so weiter...

7.1.4 Laufzeit
Laufzeit: O(|V|^2): sich jeweils den kürzesten Weg zu einem Knoten (die Auswahl von u) in jedem Schritt kostet |V| und dieser Schritt wird |V| mal gemacht => |V|^2

7.2 Flüsse in Netzwerken
[Cormen], S. 579

7.2.1 Max-Flow-Min-Cut

7.2.2 Ford-Fullerson-Max-Flow

7.3 bipartite Graphen

7.4 Kruskal (minimum spanning trees)
[Cormen], S. 505

7.4.1 Gegeben
ein gewichteter, ungerichteter Graph

7.4.2 Gesucht
der minimale Aufspannbaum (mininmal spanning tree)

7.4.3 Lösung
naja, ihr kennt das ja alle aus Inf B, nicht wahr, darum nur ganz kurz:
Zuerst packt man jeden Knoten in eine eigene Menge und ernennt diesen zum Repräsentaten der Menge. Nun nimmt man alle Kanten, sortiert diese nach Gewichten und nimmt der Reihe nach immer wieder die kürzeste (und entfernt diese dann aus dieser Kantenmenge) Kante (u,v). Dann schaut man sich den Repräsentaten von u und von v an, und wenn diese den gleichen haben, sind die beiden Knoten schon durch (egal wieviele) Kanten verbunden. Andernfalls verbindet man diese über die Kante (u,v) und vereinigt die beiden Knotenmengen von u und v und ernennt einen zum Repräsentaten (ist egal welchen, wahrscheinlich entweder v oder u oder halt einen anderen Knoten dieser Menge). Die Kante, mit der nun verbunden wurde, nimmt man in den MST auf! Wenn es nur noch eine einzige Knotenmenge gibt (d.h. alle Knoten haben den gleichen Repräsentaten), dann ist man zu Ende, der MST ist erschaffen worden...

7.4.4 Laufzeit
Die Menge der Kanten ist E, daher ist die Laufzeit O(|E|*lg|E|), eigentlich noch +|V| für die Initialisierung, aber da es prinzipiell bei zusammenhängenden Graphen |V|-1 Kante geben muss (und wir von einem zusammenhängenden ausgehen) und ein Graph mit |V|-1 Kanten schon ein MST ist (da man keine Kante wegnehmen könnte, andernfalls hätten wir keinen zusammenhängenden Graphen mehr), haben wir mindestens |V| Kanten oder mehr! Deswegen fällt bei asymptotischer Betrachtung das +|V| weg! Der Restterm kommt - so erkläre ich mir es zumindest - von der Sortierung der Kanten, wie wir am Anfang der Vorlesung sahen, ist die beste Sortierung (Quicksort) ohne Einschränkungen der zu sortierenden Menge n*lg n! (Mal abgesehen davon braucht man für die Fusion der Knotenmenge jeweils alpha(E,V)

wobei dieses alpha die reverse Ackermann-Funktion ist und das asymptotisch lg|E| ist, aber naja...)

7.5 Floyd-Warshall (all-pairs shortest paths)

[Cormen] S. 558

7.5.1 Gegeben

ein gerichteter und gewichteter Graph

7.5.2 Gesucht

von allen Knotenpaaren jeweils der kürzeste Weg zueinander

7.5.3 Lösung

Anzahl der Knoten=n

Man baut eine Matrix D(0):n*n und füllt diese wie eine Adjazenzmatrix

Anschließend baut man noch für k=1 bis n rekursiv eine neue Matrix D(k) und berechnet die Einträge aus der vorherigen Matrix, dafür nimmt man das minimum von dij(k-1) und dik(k-1)+dkj(k-1). In der Matrix D(n) sind die gewünschten kürzesten Strecken! Bei der Minimierung ist die Überlegung, dass zwischen den Knoten i und j jeweils (die durchnumerierten Knoten) 1 bis k-1 auf dem Weg liegen können. (Bei D(0) also keine-> nur direkte Verbindungen.) Bei der nächsten Matrix wird k schließlich immer um eins erhöht, d.h. ein Knoten mehr kann dazwischenliegen (und zwar genau der mit der Nummer k) und die Strecke von i nach j über k ist von i nach k und dann von k nach j => die Summe in der Minimierung; dann wird der kürzere von dem summierten Weg (über den Knoten k also) und dem anderen (ohne Knoten k) genommen.

Dazu lassen sich noch die PI Matrizen für jedes k aufstellen, in denen bei dem Eintrag ij immer der letzte Knoten vor j steht, die Matrizen lassen sich ebenso rekursiv berechnen: nimmt man bei obiger Minimierung den alten Weg (ohne k), dann nimmt man auch den alten Eintrag aus der PI Matrix, andernfalls den Eintrag von PIkj(k-). In der letzten Matrix lassen sich so alle Wege der kürzesten Pfade suchen!

Toll ist, dass man die alten Matrizen anschließend nicht mehr braucht => wenig Speicherplatz

7.5.4 Laufzeit

Da man n-mal n*n Matrizen aufstellen muss, braucht man O(n^3)

7.6 topologisches Sortieren

[Cormen], S. 485

aber ich beschreibe hier einen anderen Algorithmus!

7.6.1 Gegeben
ein gerichteter, azyklischer Graph

7.6.2 Gesucht
die topologische Sortierung des Graphen, d.h.: Wenn man ein Haus bauen will, gibt es verschiedene Dinge zu erledigen, z.B. Loch buddeln, Inneneinrichtung, zementieren, Dach drauf, Wände drauf. Da man dies in einer bestimmten Reihenfolge machen sollte (erst die Inneneinrichtung und dann das Loch für den Keller buddeln - naja), verbindet man diese Elemente mit gerichteten Kanten (wie beim Projektmanagement), dass von einem Punkt (Knoten) ausgehende Pfeile eine Vorbedingung für das Ziel des Pfeiles (ein anderer Knoten) sind. Die topologische Sortierung ist einfach die Reihenfolge, in der man die Punkte abarbeiten kann, so dass immer alle Vorbedingungen gewährleistet sind. Bei zyklischen Graphen geht das nicht, weil man dann mind. zwei Knoten hat, die gegenseitig Vorbedingung sind, man kann keinen machen, weil man erst den anderen machen müsste, den man aber nicht machen kann, weil...

Deswegen kann man den topologischen Sortieralgorithmus auch als Entscheidungsalgorithmus nehmen, ob ein azyklischer Graph vorliegt!

7.6.3 Lösung
Man hat zwei Mengen V und U, für alle V berechnet man den indegree (die Anzahl der Pfeile, die auf diesen Knoten gerichtet sind). Nun nimmt man alle Knoten aus V heraus und in U rein, die indegree von 0 haben. Dann berechnet man den indegree der Knoten in V neu und macht nimmt wieder alle mit indegree=0 und so weiter! Die Reihenfolge mit der die Knoten in U aufgenommen wurden ist die topologische Sortierung! Sie ist nicht zwangsläufig eindeutig! Den neuen indegree zu berechnen, macht man zur Verbesserung der Laufzeit über die Kanten, man nimmt einfach die Kanten (u,v) von den Knoten u heraus, die neu in U reinkamen und dekrementiert das indegree von Knoten v jeweils...

7.6.4 Laufzeit
soweit ich mich erinnere ist das O(|V|+|E|): Ich muss auf jeden Fall |V| mal einen Knoten aus V herausnehmen, daher schon mal dieses |V|, dann muss ich insgesamt alle Kanten einmal angefasst haben, um den indegree des Empfängers der Kante (des Kantenpfeils) um 1 zu verringern, daher |E|!

8. Gütefaktor
sind Probleme nicht gut lösbar, macht man eine Approximation. Die Frage ist immer: Wie gut ist die Approximation = Güte einer Approximation. V.a. ist

wichtig, dass man die Güte einer Approximation abschätzen kann, ohne die optimale Lösung (für das Problem) zu kennen.

Sei A die approximative Lösung, OPT die optimale Lösung, P das Problem, und c der Gütefaktor, so gilt immer:

|A(P)| / |OPT(P)| kleiner gleich c

Wobei c logischerweise immer größer als 1 ist, bei 1 wäre die Approximation so gut wie die optimale Lösung (also keine Approximation mehr, glaube ich) und bei kleiner 1 wäre die Approximation besser als die optimale Lösung, das geht ja auch nicht!

Ist z.B. c=2 dann heißt das soviel wie, die optimale Lösung kann max. doppelt so gut sein wie unsere approximative Lösung!

9. Komplexitätstheorie

9.1 P-Probleme

9.1.1 ?Weg von s nach t in G?

9.1.2 ?a und b relativ prim?

9.2 NP-Probleme

9.2.1 Hamilton-Kreis

9.2.2 TSP

- **Christofides**

9.2.3 SAT-Problem

9.2.4 Korrespondenz-Problem

9.3 NP-Vollständigkeit