TI1-Kurzübersicht Prüfung WS05/06

Sven Eckelmann

13. Juni 2006

Inhaltsverzeichnis

1 Laufzeiten

Algorithmus	Laufzeit
Breitensuche	$O(\|V\|+\|E\|)$
Tiefensuche	$O(\|V\|+\|E\|)$
TopSort	$O(\|V\|+\|E\|)$
Low-Werte	$O(\|V\|+\|E\|)$
Zweifache Komponenten	$O(\|V\|+\|E\|)$
Minimaler Spannbaun (Kruskal)	$O(\|E\|\log\|E\|)$
Wegkompression	$O(\log\|V\|)$
Minimaler Spannbaum nach Prim	$O(\|E\|*\|V\|)$
Prim mit Heap	$O(\|E\|\log\|V\|)$
Dijkstra	$O(\|V\|^2)$
Dijkstra ohne Mehrfache Berechnung	$O(\|E\|\log\|V\|)$
Floyd Warshall	$O(n^3)$
Ford Fulkerson	$O(\|E\|*\|f^*\|)$
Edmond-Karp	$O(\|E\|^2*\|V\|)$
Kürzeste Weg $a \rightsquigarrow b$ (dyn. Programmierung)	$O(n^2*2^n)$
Längste gemeinsame Teilfolge	$O(m*n)$
Kürzeste gemeinsame Oberfolge	$O(m*n)$
Erfüllbarkeitsproblem	$O(2^n*\|F\|)$
Davis-Putnam	$O(2^n*\|F\|)$
Monien, Speckenmeyer für k-KNF	$O(\alpha^n)$
Monien, Speckenmeyer für 3-KNF	$O(1,6181^n)$
2-Sat	polynomiell
kürzester Weg	polynomiell
längster Weg	exponentiell
Minimaler Schnitt	polynomiell
Maximaler Schnitt	exponentiell
Heuristik Maximaler Schnitt	$O(n^2)$
Backtracking für TSP	
branch-and-bound	
TSP mit dynamischem Programmieren	$O(n^2*2^n)$
Lokale Suche bei KNF	$O(1,5^n)$
Hamilton-Kreis	$O(n^2*2^n)$
Eulerscher Kreis	$O(\|V\|+\|E\|)$
TSP über Spannbaum	$O(\|E\|^4+\|V\|)$
Mergesort	$O(n\log n)$
Quicksort	$O(n^2)$
Multiplizieren	$O(n^2)$
Multiplizieren über Divide and Conquer	$O(n^{\log_2 3})$
Matrixmultiplikation	$O(n^3)$
Matrixmultiplikation über Divide and Conquer	$O(n^{\log_2 7})$
Finden von Dreiecken im Graph	$O(n^{\log_2 7})$
Max-2-Sat	$O(2^{\frac{n}{3}\log_2 7})$
Selection	$O(n^2)$
LinSelection	$O(n)$

2 Allgemeine Eigenschaften eines Graphen

2.1 Kantenanzahl

2.1.1 gerichtet

$G = (V, E), |V| = n$
$|K| \leq n(n-1) = n^2 - n = O(n^2)$

2.1.2 ungerichtet

$G = (V, E), |V| = n$
$|K| \leq \binom{n}{k} = \frac{n^2-n}{2} = O(n^2)$

2.2 Eingangsgrad

$Egrad(u) = |\{(v,u)|(v,u) \in E\}|$
$0 \leq Egrad(v) \leq n-1$

2.3 Ausgangsgrad

$Agrad(v) = |\{(v,u)|(v,u) \in E\}|$
$0 \leq Agrad(v) \leq n-1$

2.4 einfacher Weg

$(v_0, v_1, ..., v_k)$ einfach $\Leftrightarrow |\{v_0, v_1, ..., v_k\}| = k+1$ (alle v_i verschieden)

2.5 geschlossener Weg

$(v_0, v_1, ..., v_k)$ geschlossen $\Leftrightarrow v_0 = v_k$

2.6 Kreis

2.6.1 gerichtet

$k \geq 2$, einfach, geschlossen

2.6.2 ungerichtet

$k \geq 3$, einfach, geschlossen

2.7 Eulerkreis

G ist Eulerkreis $\Leftrightarrow$ G (ungerichtet) ist geschlossener Weg bei dem jede Kante genau einmal genutzt wird

2.8 Hamiltonkreis

H ist Hamiltonkreis von G (ungerichtet) $\Leftrightarrow$ G enthält Weg H der geschlossen ist und bei dem jeder Knoten von G genau einmal genutzt ist

2.9 Zusammenhang

$G = (V, E)$ (ungerichtet) ist zusammenhängend $\Leftrightarrow$ $\forall u, v \in V$ $\exists$ Weg $u \rightsquigarrow v$

2.9.1 Kreis

$G = (V, E)$ zusammenhängend und $|E| = |V| - 1 \Leftrightarrow G$ hat keinen Kreis

2.10 starker Zusammenhang

$G = (V, E)$ gerichtet, so ist G stark zusammenhängend $\Leftrightarrow$ $\forall u, v \in V$ $\exists$ Weg $u \rightsquigarrow v$ und $v \rightsquigarrow u$
Man kann zwischen allen Knoten hin und her gehen

2.11 Zusammenhangskomponente

$H = (W, F)$ ungerichteter Teilgraph von G ($W \subseteq V$, $F \subseteq E$) ist Zusammenhangskomponente von G $\Leftrightarrow$ H ist zusammenhängend, enthält alle Kanten $(u, v) \in E$ mit $u, v \in W$, aber keine Kanten $(u, v) \in E$ mit $u \in W$, aber $(v \notin W)$

2.12 starke Zusammenhangskomponente

$H = (W, F)$ gerichteter Teilgraph von G ($W \subseteq V$, $F \subseteq E$) ist starke Zusammenhangskomponente von G $\Leftrightarrow$ H ist stark zusammenhängend, maximal induziert, aber keine Kanten $(u, v) \in E$ mit $u \in W$, aber $(v \notin W)$

2.12.1 maximal induziert

- enthält alle Kanten $(u, v) \in E$ mit $u, v \in W$ (induziert)
- existiert kein weiterer stark zusammenhängender Teilgraph, der diesen als Teilgraph besitzt

2.13 Zweifach zusammenhängend

G zweifach zusammenhängend $\Leftrightarrow \forall u, v \in V, v \neq u$ gibt es zwei disjunkte Wege zwischen u und v in G.

2.14 Artikulationspunkt

v ist Artikulationspunkt von $G \Leftrightarrow G \backslash \{v\}$ ist nicht zusammenhängend.
v ist Artikulationspunkt von $G \Leftrightarrow v$ gehört ≥ 2 zweifachen Zusammenhangskomponenten

3 Breitensuche

Gibt es einen Weg von u nach v im gerichteten Graph $G = (V, E)$?
Menge der Knoten $\{v | col[v] = schwarz\}$ nach BFS sind von s erreichbar.

3.1 Algorithmus BFS

```
/** Initialisierung */
for each u in V
   col[u] = weiß;
col[s] = grau;                   // s Startknoten
Q = {s};                         // Initialisierung zu Ende.
/** Abarbeitung der Schlange */
while (Q != {Ø}){                 // Testbar mit tail= head
   u = Q[head];                  // u wird bearbeitet (expandiert)
   for each v in Adj[u] {
      if (col[v] == weiß) {      // v wird entdeckt
         col[v] = grau;
         v in Schlange tun;
      }                          //Schlange immer grau.
   }
   u aus Q raus;
   col[u] = schwarz;             // Knoten abgearbeitet
}
```

3.2 BFS umgangssprachlich

1. Schiebe Startknoten in Schlange
2. Nehme ersten Knoten aus Schlange (markiere Schwarz) - wenn keine Knoten in Schlange Abbruch
3. Schiebe alle erreichbaren, unbearbeiteten Knoten in Schlange (markiere Grau)
4. Fange von 2. an bis keine Knoten mehr in Schlange

4 Topologische Sortierung

Ist $G = (V, E)$ gerichteter Graph: Sortierung von V so, dass alle Kanten gehen von links nach rechts in der Sortierung.
G hat topologische Sortierung $\Leftrightarrow$ G hat keinen Kreis

4.1 Algorithmus TopSort

```
1./** Initialisierung */
   Egrad[v] auf 0, Q = {Ø};
   for each v in V {
      Gehe Adj[v] durch,
      zähle für jedes gefundene u
      Egrad[u] = Egrad[u]+1;
   }
2. /** Einfügen in Schlange */
   for each u in V {
      if Egrad[u] = 0
          Q = Q U {u}
   }
3. /** Array durchlaufen */
   for i = 1 to n {
      if Q = {Ø}
          "Ausgabe Kreis"; return;

   4. /** Knoten aus Schlange betrachten*/
      v[i] = Q[head];
      Lösche Q[head] aus Q;

   5. /** Adjazenzliste durchlaufen*/
      for each u in Adj[v[i]] {
          Egrad[u] = Egrad[u]-1;
          if Egrad[u] = 0
            Q = Q U {u}
      }
   }
```

4.2 TopSort umgangssprachlich

1. Bestimme Egrad jedes Knotens (am besten Adjazenzliste durchgehen und bei jeder Möglichkeit diesen zu erreichen +1 hochzählen)
2. Schiebe alle mit Egrad = 0 in Schlange
3. von i=1 bis Anzahl der Knoten
   (a) Ist Schlange leer wurde Kreis gefunden -> Abbruch mit Fehlermeldung

(b) Sonst nehme ersten aus Schlange und setze ihn im Sortierungsfeld an i-te Stelle

(c) Verringere bei allen von diesem Knoten erreichbare Knoten den Egrad um 1 (ähnlich wie bei Initialisisierung)

(d) Sollte dabei Egrad bei Knoten 0 werden, schieben sie in Schlange

5 Tiefensuche

- d[1...n] - Entdeckzeit
- f[1...n] - Beendezeit
- pi[1...n] - Tiefensuchwald (pi[u] - Knoten über den u entdeckt wurde)
- col[1...n] - aktuelle Farbe des Knotens

5.1 Algorithmus DFS

```
DFS(G)
/** 1. Initialisierung */
for each u in V{
   col [u]: = weiß;
   pi[u]: = nil;
}
time: = 0;
/** 2. Hauptschleife,
* Aufruf von DFS-visit
* nur, wenn col[u] = weiß
*/
for each u in V{
   if (col[u] == weiß)
      {DFS-visit}
}

DFS-visit(u)
1. col[u] = grau;        //Damit ist u entdeckt.
2. d[u] = time; time = time + 1;
3. for each v in Adj[u]{ //u wird bearbeitet
4. if(col[v] == weiß){   //(u, v) untersucht
       pi[v] = u;
5.                       //v entdeckt
6.     DFS-visit(v);
      }
   }                     // Die Bearbeitung von u ist zuende
   /** Sind alle Knoten aus Adj[u] grau oder schwarz,
   * so wird u direkt schwarz.
   */
```

```
7. col[u]: = schwarz;
8. f[u]: = time;
   time: = time +1;      //Zeitzähler hoch bei Entdecken+Beenden
```

5.2 DFS umgangssprachlich

1. Schaue dir von jedem Knoten alle erreichbaren, unbesuchten Knoten an und besuche sie sofort (a)
   (a) Suche nach von diesem Knoten erreichbaren, unbesuchten Knoten und besuche ihn sofort
   (b) Wenn keine unbesuchten Knoten mehr von aktuellen Knoten erreichbar, gehe Weg wieder Knoten für Knoten zurück und schaue ob jeweils weitere Knoten von diesen besuchbar sind (a)

5.3 Kantenklassifikation

1. Baumkanten $\Leftrightarrow (u,v) \in E_\Pi(\Pi[v] = u)$
   Kante, die im Tiefensuchbaum enthalten ist
2. Rückwärtskanten $\Leftrightarrow (u,v) \notin E_\Pi$ und v Vorgänger von u in G_Π
   Kante, die auf einen Vorgänger von u im Tiefensuchbaum zeigen würde
3. Vorwärtskanten $\Leftrightarrow (u,v) \notin E_\Pi$ und v Nachfolger von u in G_Π
   Kante, die auf einen Nachfolger von u im Tiefensuchbaum zeigen würde
4. Kreuzkanten $\Leftrightarrow (u,v) \notin E_\Pi$ und v weder Vorgänger noch Nachfolger von u in G_Π
   Kante, die auf einen Knoten in einem anderen Weg im Tiefensuchbaum zeigen würde

5.4 Weiße Weg

v wird über u entdeckt $\Leftrightarrow$ zum Zeitpunkt $d[u]$ existiert ein (weißer) Weg in G.

5.5 Kreis

$G = (V,E)$ (gerichtet) hat Kreis $\Leftrightarrow$ DFS(G) ergibt Rückwärtskante (Kante auf grauen Knoten)

6 Starke Zusammenhangskomponente

6.1 Starke Komponenten

1. DFS(G) und speichere Knoten nach absteigender Beendezeit $V = (v_1, v_2, ..., v_n)$
2. Drehe Kanten in G um (G^U)

3. Rufe DFS(G^U) auf und rufe Knoten in DFS entsprechend absteigender Beendezeit auf
   Jeder Aufruf von DFS-visit in DFS gibt neue starke Komponente an

7 Zweifache Zusammenhangskomponente

7.1 Algorithmus l-Werte

```
Modifikation von DFS-visit(u):

MDFS-visit(u)                    //hier col[u] = weiß
.
.
.
l[u]=d[u]
.
.
.
for each v in Adj[u] do{
   if (col[v] == weiß){
      pi[v] = u; MDFS-visit(v);
      l[u] = MIN{l[u], l[v]}
   }
   if (col[v] == grau) and (pi[u] = v){
      l[u] = MIN{l[u], d[v]}     //d[v]! nicht l[v], keine Iteration.
   }
}
```

7.2 l-Werte umgangssprachlich

Normale Tiefensuche, aber:

1. Wenn man Knoten besucht, nimmt man Entdeckzeit als l-Wert des Knotens
2. Trifft man auf grauen Knoten bestimmt man neuen l-Wert des Knotens aus dem Minimum des l-Wertes des aktuellen Knoten und eigenem l-Wert
3. Wenn man wieder Rückwärts geht, bestimmt man den neuen l-Wert als Minimum des l-Wertes des zuletzt besuchten Knotens (von dem man kommt) und dem dem eigenen l-Wertes

7.3 Algorithmus Zweifache Komponenten

```
1. DFS(G)                            //modifiziert für l-Werte

2. DFS(G)                            //in derselben(!) Reihenfolge wie 1. mit
   NDFS-visit(u)                     //col[u] = w
```

```
.
.
.
for each v in Adj[v]{
   if (col[v] == w){
      pi[v] = u, v auf Keller;
      NDFS-visit(v);
      if (l[v] >= d[u]){
         Ausgabe bis v inklusive,
         u noch hinzu ausgeben
      }
   }
}
```

7.4 Zweifache Komponenten umgangssprachlich

1. Tiefensuche zur Bestimmung der l-Werte
2. Tiefensuche mit selben Reihenfolge
   (a) Findet man weißen Knoten, schiebt man ihn auf Stack und besucht ihn (wie normale Tiefensuche)
   (b) Geht man wieder zurück schaut man ob der l-Wert von Sohn $\geq$ eigener Entdeckzeit ist. Wenn ja hat man Artikulationspunkt (der aktuelle Knoten u) gefunden und gibt Stack bis inklusive v (dem Sohn von u von dem wir kommen) aus und zusätzlich den Artikulationspunkt.

8 Minimaler Spannbaun

8.1 Algorithmus von Kruskal

```
1. F = Ø, P = {{1}, ..., {n}} // P ist die Partition
2. E nach Kosten sortieren    // geht in O(E log |E|)=O(|E| log|V|),
                              // E<=V2 , log|E|=Olog|V|
3. while(|P|>=1){              // solange P={{1, ..., n}}
4.   {v, w} = kleinstes (erstes) Element von E
     {v, w} aus E löschen
5.   Testen, ob F U {v, w} einen Kreis hat

6.   if({v, w} induziert keinen Kreis){
        Wv = die Menge mit v aus P;
        Ww = die Menge mit w aus P;
        Wv und Ww in P vereinigen
        F = F U {{v, w}}
     }
  }
```

1. Sortiere Kanten nach Kosten

2. Nimm billigste Kante aus den Kanten
3. Verbindet die Kante nicht zwei Knoten aus der selben Menge (also bildet keinen Kreis), füge die Kante hinzu und verbinde die zwei Mengen zu denen sie gehören zu einer
4. Mache Weiter bei 2. bis es nur noch eine Menge gibt

8.2 Union-Find

Datenstruktur (zum Beispiel als Bäume deren Knoten auf die Vaterknoten zeigen und die Wurzel auf sich selbst) zum Verwalten von Partitionen von Mengen.

8.2.1 Operationen

- Init(S) - Erstellt für jeden Knoten eine eigene Klasse/Menge
- Union(r, s) - Vereinigt beide Klassen/Mengen zu einer Klasse mit Repräsentant r
- Find(x) - Bestimmt Repräsentant der zu x gehört (im Baum wird Kanten bis Wurzel nach oben gegangen)

8.2.2 Union-By-Size

- Union(r, s) - hängt den niedrigeren Baum unter die Wurzel des höheren Baum. Falls notwendig wird, um r als Wurzel zu erhalten, s und r vertauscht

8.2.3 Algorithmus Wegkompression

Es wird versucht den nötigen Weg bei Find zu verkürzen.

```
Find(v)
1. v auf Keller tun        // etwa als array implementieren
2. while (P[v]!=v){         (wie Schlange)
3.    v = P[v];
      v auf Keller tun
   }
4. v ausgeben;
5. for (w auf dem Keller){
      P[w] = v;            //Können auch Schlange, Liste oder
   }                       //sonstwas nehmen, da Reihenfolge egal
```

1. Vom Ausgangsknoten aus werden Knoten auf Stack gespeichert bis man bei Wurzel angelangt ist
2. Alle Knoten im Stack erhalten als Vaterknoten die Wurzel des Baumes

8.3 Algorithmus nach Prim

```
1. Wähle irgendeinen Startknoten r
   Q = V\{r}                       // Q enthält immer die Knoten,
   F=Ø                             // die noch bearbeitet werden müssen.
2. while (Q != Ø {
3.   M = {{v, w}| v in V\ Q, w in Q}

                                   // M = Menge der Kanten mit
                                   // genau einem Knoten in Q
     {v, w} = eine Kante von minimalen Kosten in M
4.
     F = F U{{v, w}};
     Q = Q ohne den einen Knoten aus {v, w}, der auch zu Q gehört
```

8.4 Prim umgangssprachlich

1. Nehme irgendeinen Knoten und entferne ihn aus der Menge der noch nicht bearbeiteten Knoten
2. Solange es noch nicht bearbeitete Knoten gibt
   (a) Suche Kante mit geringsten Kosten, die aus dem Bereich der bearbeiteten Knoten in den Bereich der noch nicht bearbeiteten Knoten zeigt
   (b) Speichere die Kante und entferne den Knoten aus dem Bereich der noch nicht bearbeiteten Knoten

8.5 Heap

(hier binärer Heap als Vorrangwarteschlange - totale Ordnung muss existieren)

8.5.1 Eigenschaften

- Vaterknoten ist immer kleiner als Kinderknoten (Wurzel hat größte Priorität/kleinsten Wert)
- Alle Ebenen sind bis auf letzte voll gefüllt

8.5.2 Minimum löschen

1. Entferne Minimum
2. Setze für leeren Platz den letzten Knoten (der in letzten Ebene unten Rechts ist)
3. Tausche mit kleinerem Kind bis Heapeigenschaft wieder hergestellt

8.5.3 Einfügen

1. Füge an ersten freien Platz (rechts von letzten Knoten) neuen Knoten ein
2. Tausche solange mit Vater bis wieder Heapeigenschaft hergestellt

8.6 Algorithmus Prim mit Q in Heap

```
1. Wähle Startknoten r.
   for each v in Adj[r]{
       key[v]=k({r,v});
       kante[v]=r}
   Für alle übrigen v in V{
       key[v]=inf; kante[v]=inf}
   Füge V\{r} mit Schlüsselwerten key[v] in heap Q ein.
2. while Q != Ø{
3.     w = Min; DeleteMinQ ;
       F = FU{{kante[w],w}}
4. for each u in Adj[w] die auch in Heap Q{
       if K({u,w})<key[u]{
         key[u] = K({u,w});
         kante[u] = w;
         Q anpassen}
       }
   }
```

8.7 Prim mit Heap umgangssprachlich

1. Nehme irgendeinen Knoten
2. Speichere alle adjazente Knoten v dieses Knotens r mit den Kosten der Kante (r, v) im Heap
3. Speichere alle restlichen Kanten mit Kosten ∞ in Heap
4. Solange es noch Knoten im Heap gibt
   (a) Nimm Kante mit minimalen Kosten aus Heap
   (b) Speichere die Kante und entferne den Knoten aus dem Bereich der noch nicht bearbeiteten Knoten
   (c) Alle Knoten u die adjazent zu neu gefundenem Knoten w, noch im Heap und deren Kosten für Kante (u, w) kleiner als Kosten (key) des Knotens u im Heap sind, speichere neue Kosten für Knoten u im Heap und passe ihn an (heapify).

9 Kürzeste Weg

9.1 Algorithmus Dijkstra

Für Graphen ohne Kreise

```
1. D[s] = 0, S = {s}, Q=V\{s}
2. for (i = 1 to n - 1){                //Suchen noch n-1 kürzeste Wege
3.    M = {{v, w}|v in S, w in Q}
4.    for each w in Q adjazent zu S{
5.       D[w] = Min{D[v] + K(v, w)|(v,w) in M}}
6.    w = ein w in Q mit D[w] minimal;
7.    S = S + {w}, Q = Q\{w}}
```

9.2 Dijkstra umgangssprachlich

1. Füge s in Menge der bearbeiteten Knoten (S) ein, Setze Abstand zu s (D[s]) auf 0 und entferne es aus der Menge der noch zu bearbeitenden Knoten (Q)
2. $(n-1)$x mache
   (a) Finde Knoten w in der Menge der noch nicht bearbeiteten Knoten auf den eine Kante aus der Menge der bearbeiteten Knoten zeigt und dessen Distanz zu s Minimal ist (D[v]+K(v, w) minimal)
   (b) Füge s in S ein und entferne s aus Q

9.3 Algorithmus Dijkstra ohne mehrfache Berechnung desselben D[w]

```
1. D[1] = 0; D[v] = inf für v in V\{s}; Q = V\{s}; S = {s};
2. for i = 1 to n - 1 {
3.    w = ein w in Q mit D[w] minimal;
4.    S = S + {w}; Q = Q\{w};
5.    for each v in Adj[w]{      //D[v] aussparen fur v adjazent zu w
6.       if D[w] + K(w, v) < D[v]{
            D[v] = D[w] + K(w, v) }}
```

9.4 Algorithmus kürzeste einfache Weg

```
KW (W, u, v)                  //u, v in W, W = nach b betrachtete Knotenmenge
1. if (u == v)
      return 0;
2. l = inf;
3. for each w in Adj[u] die auch in W{
      l' = KW(W\ {u}, w, v);
      l' = K(u, w) + l'j ;
     if(l' < l)
```

```
                //Hier pi[w] = u gibt kürzeste Wege selbst
        l = l';
   }
4. return l  //Ausgabe inf, wenn Adj[u] ver W = Ø.
```

9.5 KW umgangssprachlich

1. Entferne Startknoten aus der Liste der noch zu benutzenden Knoten
2. Teste rekursiv über welchen neue Startknoten in der Adjazenzliste von unserem bisherigen Startknoten, der aber auch in der Liste der noch zu benutzenden Knoten sein muss, den kürzenden Weg von unserem bisherigen Startknoten zu unserem Zielknoten hat $(\min(K(u,w)+KW(W\backslash\{u\},w,v)\forall w \in W \bigcap Adj[u]))$.
3. Ist der Startknoten der Zielknoten, dann gebe 0 zurück. Wenn Startknoten $\neq$ Zielknoten ist, aber keine Knoten mehr in den noch benutzbaren Knoten W sind, dann gebe ∞ zurück.

9.6 Kürzeste einfache Weg mit Setzen

Umsetzung kurzesten einfachen Weg mit Hilfe der Dynamischen Programmierung

```
KW(V, a, b){
1. for i = 1 to n {
2.    for each W <= V, b in W, |W| = i{
3.       for each v in W{
4.          Setze(W, v);
         }
      }
   }
}

Setze(W, v){
1. if (v == b) {
      T[W, v] = 0;
      pi[W, v] = b;
      return

   T[W, v] = inf; W‘ = W\{v}
2. for each u in Adj[v] und in W {
   l = K(v, u)+T(W‘, b);
   if l < T[W, v] {
      T[w, v] = l;
      Pi[W, v] = u;
   }
}
```

9.7 Kürzeste Weg umgangssprachlich

1. Erstelle Matrix T mit bis zu 2^n Zeilen und n Spalten
   $T[W,v]$ gibt nun minimale Kosten für den Weg $v \rightsquigarrow b$ durch Knoten der Menge W an (b unser Zielknoten)
2. Initialisiere $T[b,b] = 0$, $T[b,v] = \infty$ für $v \neq b$
3. Für $|W| = 2$ (also 2 Knoten in W, von dem ein Knoten $\neq b$ ist) $T[W,b] = 0$ und $T[W,v] = K(v,b)$ für $v \neq b$
4. Gehen weitere Zellen Zeile für Zeile durch
   (a) T[W, b] = 0
   (b) für $v \neq b$ $T[W,v] = \min\{K(v,u) + T[W\backslash\{v\},u] \ \forall u \in W\}$
   Versuchen also v als Startknoten und ein u aus $W\backslash\{v\}$ als nächsten Knoten einzufügen. Aus den Gesamtsummen (also mit dem restlichen Weg bis b) wird dann die günstigste Variante ausgesucht und eingetragen.

9.8 Algorithmus Floyd Warshall

Für Graphen mit Kreisen > 0

```
1. T[u, v] = 0, T[u, v] = K(u, v) für (u, v) in E
   T[u, v] = inf für u = v, (u, v) in E
2. for i = 1 to n{
      for each (u, v) in V{        Alle geordneten Paare.
         T[u, v] = Min{T[u, v], T[u, i] + T[i, v]}
   }
```

1. Erstelle Matrix (ähnlich Adjazenzmatrix) mit Kosten als Einträgen (sonst ∞)
2. Erstelle für alle Knoten ($w \in V$) alle Einträge neu mit $T[u,v] = \min(T[u,v], T[u,w] + T[w,v])$
   Testen ob aktueller Weg oder Weg über anderen Knoten günstiger ist

10 Flüsse in Netzwerken

10.1 Min-Cut-Max-Flow

f ist maximaler Fluss $\Leftrightarrow G_f$ hat keinen Erweiterungspfad $\Leftrightarrow G_f$ Es gibt einen Schnitt S, T , so dass $|f| = K(S,T)$

10.2 Algorithmus von Ford und Fulkerson

```
1. f (u, v) = 0 für alle u, v in V
2. while Es gibt Weg s -> t in Gf {
3. W = ein Erweiterungspfad in Gf
4. g = Fluss in Gf mit g(u, v) = Kf (W ) für alle u -> v in W , wie oben
```

```
5. f = f + g }
Gib f als maximalen Fluss aus.
```

10.3 Ford Fulkerson umgangssprachlich

1. Suche Erweiterungspfad (Weg von Quelle zur Senke mit minimaler Kapazität $\neq 0$
2. Bestimme Minimum
3. Subtrahiere das Minimum entlang des Erweiterungspfades von den Kapazitäten
4. Erstelle Kanten in Gegenrichtung zum aktuellen Erweiterungspfad mit Kapazität des bestimmten Minimums
5. Addiere gefundenen Fluss (Erweiterungspfad mit Fluss auf Minimum beschränkt) zu bisher gefundenem Fluss
6. Wiederhole diese bis kein Erweiterungspfad mehr gefunden werden kann

10.4 Edmond Karp-Strategie

Da der Algorithmus von Ford und Fulkerson pseudopolynomiell ist (da $|f*|$ maximaler Fluss), wird zur Auswahl des Erweiterungspfades die Breitensuche (für Kanten mit Kapazität > 0) genutzt. Damit immer von der Kantenanzahl kürzeste Wege.

11 Kombinatorische Suche

11.1 Algorithmus Erfüllbarkeitsproblem

```
Erzeuge hintereinander alle Belegungen a(0...0, 0...1, 0...10, ..., 1...1)
  Ermittle a(F)
  Ist a(F) = 1, return "F erfüllbar duch a"
return "F unerfüllbar."
```

11.2 Erfüllbarkeitsproblem umgangssprachlich

1. Erzeuge alle Belegungen
2. Erfüllt eine Belegung F, dann ist F erfüllbar, sonst nicht

11.3 Algorithmus Davis-Putnam

```
DP(F){
1. if F offensichtlich wahr         // leere Formel
       return F erfüllbar           // ohne Klausel
2. if F offensichtlich unerfüllbar  // Enthält leere Klausel
```

```
       return unerfüllbar           // oder Klausel x und !x
3. Wähle eine Variable x von F gemäß einer Heuristik.
   Wähle (b1 , b2 ) mit b1 , b2 = 0 oder b1 =0, b2 =1
4. H:=F-x=b1 ; a[x]:=b1 ;
   if (DP(H) == H) erfüllbar
     return "F erfüllbar"
5. H:=F-x=b2 ; a[x]:=b2
// Nur, wenn in 4. "unerfüllbar"
return DP(H)
}
```

11.4 Davis-Putnam umgangssprachlich

1. Versuche über Backtracking Belegung zu finden, die die Formel erfüllt. Dabei soll die Formel vereinfacht werden (Klauseln werden erfüllt - Fallen weg oder nur Literale Fallen weg und werden gestrichen)
2. Versuche dabei über eine Heuristik den Baum etwas zu vereinfachen, da in aktuell gewählten Zustand über Pure-literal-rule oder unit-clause-rule eine Belegung für ein Literal direkt gegeben sein könnte
3. Sollte dies nicht möglich sein, wähle eine Belegung (ein Literal mit 1 belegen) und teste ob sie erfüllbar ist, sonst wähle die Alternative (das selbe Literal mit 0 belegen) und schaue ob sie damit erfüllbar ist.

11.4.1 Pure literal rule

x ist ein pures Literal (taucht entweder nur normal oder nur negiert auf). Wenn es nur normale Auftaucht, Belegung = 1, sonst Belegung = 0.

```
if es gibt ein x in F , so dass !x nicht in F {
    H := F |x = 1, a[x] := 1; return DP(H);
}
if !x in F aber x nicht in f {
    H := F |x = 0; a[x] = 0; return DP(H);
}
```

11.4.2 Unit clause rule

Existiert Klausel mit nur einem Literal. Belege Literal so, dass Klausel erfüllt ist.

```
if es gibt eine Einerklausel (x) in F {
    H := F|x = 1; a[x] := 1; return DP(H);
}
if (!x) in F {
    H := F|x = 0; a[x] := 0; return DP(H);
}
```

11.5 Erfüllbarkeitsäquivalente Umformung in 3-KNF

Hier als Beispielformel: $(x+\overline{y}) \Leftrightarrow (y * z)$

1. Schreibe Formel als Baum mit Variablen = Blätter und Operationen innere Knoten
2. Operationen erhalten neue Variablen N_1 bis N_l
3. Schreibe immer Vaterknoten mit seinen Kindern als $N_1 \Leftrightarrow (x+\overline{y})$
   Erhalten also $F = (N_1 \Leftrightarrow (x+\overline{y})) * (N_2 \Leftrightarrow (y+z)) * (N_3 \Leftrightarrow (N_2 \Leftrightarrow N_3))$
4. Formen einzelne Klauseln weiter um in 3-KNF
   Beispielsweise:
   - $N_1 \Leftrightarrow (x+\overline{y}) = (N_1 \Leftarrow (x+\overline{y})) * (N_1 \Rightarrow (x+\overline{y}))$
   - $N_1 \Leftarrow (x+\overline{y}) = (\overline{N_1} + (x+\overline{y})) * (N_1 + \overline{(x+\overline{y})}) = ...$

11.6 Algorithmus Monien/Speckenmeyer

```
1. if F offensichtlich wahr return "erfüllt"
2. if F offensichtlich unerfüllbar return "unerfüllbar"
3. Wähle eine kleinste Klausel C,
   C=l1 + ... + li in F                         //li Literal, x oder !x
4. Betrachte die Belegungen l1 = 1; l1 = 0, l2 = 1; ...;l1 = 0, l2 = 0, ..., ls = 1
5. if (eine der Belegungen autark in F){
       b := eine autarke Belegung; return MoSP(F|b )}
6. Teste = MoSP(Fi) für i=1, ..., s              // Hier ist keine der
   und Fi jeweils durch Setzen einer der         // Belegungen autark
   obigen Belegungen; return "erfüllbar", wenn ein Aufruf
    "erfüllbar" ergibt, "unerfüllbar" sonst.
```

12 Traveling-Salesman-Problem

12.1 Algorithmus Backtracking für TSP

```
1. if M stellt Rundreise dar
     return (M, Kosten von M)
2. if M hat keine Rundreise <= inf
     return (M, inf)                // Etwa eine Zeile voller inf,
                                    // eine Spalte voller inf
3. Wähle (u,v), u=v mit M(u,v) <= inf,
  wobei in Zeile von u oder Spalte von v mindestens ein Wert = inf
4. M':=M modifiziert, so dass u -> v gewählt
5. M”:= M modifiziert, dass M(u,v) = inf
6. Führe TSP(M') aus
  Führe TSP(M”) aus
7. Vergleiche die Kosten;
  return (M, K), wobei M die Rundreise der kleineren Kosten ist.
```

12.1.1 Modifizieren der Matrix bei Wahl $u \to v$

1. Streiche Zeile u (bis auf Zelle (u,v)) indem man Elemente auf ∞ setzt
2. Streiche Spalte v (bis auf Zelle (u,v)) indem man Elemente auf ∞ setzt

12.1.2 Modifizieren der Matrix bei nicht Wahl $u \to v$

1. Streiche Zelle (u,v) indem man es auf ∞ setzt

12.2 Schranken

- $S_1(M)$ = minimale Kosten einer Rundreise unter M
- $S_2(M)$ = Summe aller bisher gewählten Kanten
- $S_3(M) = \sum_{v\in V} \min\{M(u,v) + M(v,w) : u,w \in V\}$
- $S_4(M) = \sum_{v\in V} \min\{M(u,v) : u \in V\} + \sum_{u\in V} \min\{M(u,v) - \min\{M(w,v) : w \in V\} : v \in V\}$
    1. Bilde Summe der Minima der Zeilen
    2. Reduziere Zeilen um ihr Minimum
    3. Addiere zur ersten Summe die Summe der Minima der Spalten der reduzierten Matrix

12.3 Offizielles Branch-and-bound

Wie Branch and bound, aber rechne die untere Schranke aus und schaue ob sie schlechter als bisher berechnete Kürzeste Rundreise ist. Wenn ja, dann breche den Pfad ab und gehe wieder einen Aufruf zurück.

12.4 Algorithmus dynamische Programmierung für TSP

```
1. for i=2 to n TSP(k, 0 ..., 0) := M(k,1)
2. for i=2 to n-2 {
     for all S <= {2, ..., n}, |S| = i {
     for all k \in {2, ..., n}\ S{
        TSP(k,S) = Min{M(k,s) + TSP(s, S\ {s})}
     } }}
```

13 Divide-and-Conquer

13.1 Algorithmus Mergesort

```
Mergesort(A[1, ..., n]){
```

```
1. if (n==1) oder (n==0) return A;
2. B1 = Mergesort (A[1, ..., n/2]);
3. B2 = Mergesort (A[n/2+1, ..., n]); //2. + 3. divide-Schritte
4. return "Mischung von B1 und B2 "   //Mischung bilden, conquer-Schritt
}
```

13.2 Algorithmus Quicksort

```
1. if (n == m) return
2. a = ein A[i];
3. Lösche A[i] aus A.
   for j=1 to n {
4.     if (A[j] <= a){
5.                                 //A[j] als nächstes Element zu B1 ;
         break;
         }
6. if (A[j] > a){
       A[j] zu B2 }
   }
7. A = B1 a B2
8. if (B1 != Ø) Quicksort (B1 ) // B1 als Teil von A
9. if (B2 != Ø) Quicksort (B2 ) // B2 als Teil von A
```

13.3 Algorithmus Multiplikation

Einteilen der Zahlen in obere und untere Hälfte

1. $a_0 = a[n, ..., \frac{n}{2} + 1]$
2. $a_1 = a[\frac{n}{2}, ..., 1]$

$a * b \;=\; a_1 * b_1 * 2^n + (a_0 * b_0 + a_1 * b_1 + (a_1 - a_0) * (b_0 - b_1)) * 2^{\frac{n}{2}} + (a_0 * b_0)$ 2^x kann durch geeignete Shiftoperationen implementiert werden.

13.4 Algorithmus Matrixmultiplikation

Einteilung der Matrizen in 4 Quadranten, die rekursiv verarbeitet werden.

1. A_{11} = Matrix A oben links
2. A_{12} = Matrix A oben rechts
3. A_{21} = Matrix A unten links
4. A_{22} = Matrix A unten rechts

$$
\begin{aligned}
P_1 &= A_{11} * (B_{12} - B_{22}) \\
P_2 &= (A_{11} + A_{11}) * B_{22} \\
P_3 &= (A_{21} + A_{22}) * B_{11} \\
P_4 &= A_{22} * (-B_{11} + B_{21}) \\
P_5 &= (A_{11} + A_{22}) * (B_{11} + B_{22}) \\
P_6 &= (A_{12} - A_{22}) * (B_{21} + B_{22}) \\
P_7 &= (A_{11} - A_{21}) * (B_{11} + B_{21})
\end{aligned}
$$

$$
\begin{aligned}
C_{11} &= (P_5 + P_4 - P_2) + P_6 \\
C_{12} &= P_1 + P_2 \\
C_{21} &= P_3 + P_4 \\
C_{22} &= (P_5 + P_1 - P_3) - P_7
\end{aligned}
$$

13.5 Max-2-KNF

13.5.1 Dreiecke in gerichteten Graph

$$\sum_{i=0}^{n-1} A^3[i,i] = 6 * \#\, Dreiecke$$

13.5.2 Algorithmus (i_1 , i_2 , i_3)-2-Sat

```
01. V1 = {(b1, ... , b_(n/3) )1 |bi in {0, 1}}  //V = V1 U V2 U V3 , Knoten
    V2 = {(b1, ... , b_(n/3) )2 |bi in {0, 1}}  //|V| = 3 * 2^(n/3)
    V3 = {(b1, ... , b_(n/3) )3 |bi in {0, 1}}

02. for each b,c in {0, 1}^(n/3) {              // Typ 1
03.    Betrachte (b,c) als Belegung a von x1, ..., x_(2/3*n)
04.    g:=|{k | Ck vom Typ 1, a(Ck )=1}|
05. if g=i1 Kante (b1 , c2 ) einbauen}
06. for each c,d in {0, 1}^(n/3) {              // Typ 2
07.    (c,d) als Belegung a von x_(n/3 +1), ..., x_n
08.    g:=|{k | Ck vom Typ 2, a(Ck )=1i}|
09.    if g = i2 Kante (c2 , d3 )}
10. Ebenso for each d,b in {0,1}^(n/3)          //Typ 3
        Analog.
11. A: Adjazenzmatrix
12. B:= A A A
13. Ausgabe: # Anzahl Dreiecke
```

13.6 (i_1 , i_2 , i_3)-2-Sat umgangssprachlich

1. Teile Literale gleichmäßig in 3 Gruppen ein
2. Teile Klauseln in 3 Verschiedene Typen ein
   - Typ 1: Klauseln die nur Literale aus Gruppe 1 oder 2 haben
   - Typ 2: Klauseln die nur Literale aus Gruppe 2 oder 3 haben

- Typ 3: Klauseln die nur Literale aus Gruppe 3 oder 1 haben

3. Bilde alle Belegungen für jede Gruppe (das macht $3*2^{\frac{n}{3}}$-Belegungen - alle diese Belegungen bilden nun Knoten in Adjazenzmatrix A)
4. Teste alle Belegungen von Gruppe 1 und 2 mit den Klauseln vom Typ 1 und trage eine ungerichtete Kante zwischen der Belegung der Gruppe 1 und 2 ein, wenn jeweils genau i_1 Klauseln erfüllt werden
5. Teste alle Belegungen von Gruppe 2 und 3 mit den Klauseln vom Typ 2 und trage eine ungerichtete Kante zwischen der Belegung der Gruppe 2 und 3 ein, wenn jeweils genau i_2 Klauseln erfüllt werden
6. Teste alle Belegungen von Gruppe 3 und 1 mit den Klauseln vom Typ 3 und trage eine ungerichtete Kante zwischen der Belegung der Gruppe 3 und 1 ein, wenn jeweils genau i_3 Klauseln erfüllt werden
7. Rechne über Anzahl der Dreiecke aus und gebe sie zurück

13.7 Algorithmus Max-2-Sat

```
1. for 1 <= I1 , I2 , I3 <= M{
2.   E[I1 , I2 , I3 ] := (I1 , I2 , I3 )-2-Sat mit F
3. Suche den Eintrag E[I1 , I2 , I3 ]>0 aus,
     wo I1 + I2 + I3 maximal. Gebe ihn aus
```

13.8 Max-2-Sat umgangssprachlich

1. Teste alle Möglichkeiten bei der $i_1 + i_2 + i_3 \leq$ der Anzahl an Klauseln ist mit (i_1 , i_2 , i_3)-2-Sat
2. Merke dir das Maximum von $i_1 + i_2 + i_3$ bei der (i_1 , i_2 , i_3)-2-Sat als 0 war
3. Gebe das Maximum zurück

13.9 Algorithmus Selection

```
Select(1,n,i)                     //Linker Rand, rechter Rand, 1 <= i <= n
1. if (1==n) {Ausgabe A[1]; return}
2. a:=A[j] für ein gewähltes j //Pivotelement
3. B1 := die Elemente von A[1, ...,n], die <= a sind
   b1 := die # dieser Elemente
   if i <= b1 {                   //Hier b1 >= 1 da i >= 1
       A[1, ..., b1] := B1
   Select(1, b1, i);return}
4. B2 := die Elemente von A, die > a sind
   b2 := die Anzahl dieser Elemente
   A[1, ..., b1]:=B2              //i-b1 >= 1 und i - b1 <= 2
   Select(1, b2, i-b1)             //da i <= b1 + b2 = n, b2 < n, da b1 >= 1
```

1. Wähle irgendein Element als Pivotelement

2. Schreibe alle Elemente die kleiner oder gleich als das Pivotelement sind in B1 und alle die größer als Pivotelement sind in B2

3. ist i kleiner oder gleich als die Anzahl der Elemente in B1, dann ist das gesuchte Element auch das i. größte Element in B1

4. ist i größer als Anzahl der Elemente in B1, dann ist es das gesuchte Element das (i-b1) größte Element in B2

13.10 Algorithmus Linear Selection

```
LS(A,i)                                    //A = A[1, ..., n],
                                           alle Elemente verschieden,
                                           //1 <= i <= n, n durch 10 teilbar
1. if (n <= 1000){
     Sortieren und i-tes Element zurückgeben; return
                                           //Ende der Rekursion
2. for 1 <= j <= n { 5
     Sortiere A[(j - 1)5 + 1, -, j * 5], B[j] = A[(j - 1)5 + 3]
                                           //B[j] = Median des j-ten 5er-Teils von A
   }
3. a:=LS(B, 1/2 * 1/5 * n)                 //Beachte n durch 10 teilbar
                                           //a Pivotelement
4. B1 := die Elemente von A, die <= a sind
   b1 := die # dieser Elemente
   if b1 = i-1{
      Rückgabe von a; return
   }
5. if i <= b1 -1{                     //b1 -1 < n
      Fülle B1 hinter b1 -1 mit inf auf, damit die
      Länge durch 10 teilbar wird.
      Rückgabe LS(B1 ,i);return
   }
6. if i>b1 {
      Dasselbe wie oben mit B2 ; Rückgabe von LS(B2 ,i-b1 );
                                     //b1 >= 1 Wichtig, dass a = A[j]!
      return;
```

Das Pivotelement könnte bei Select immer schlecht gewählt werden (immer entweder das kleinste oder größte Elemente). Dadurch erreicht man Laufzeit von $O(n^2)$. Wenn man möglichst schnell ein mittleres Element findet, reduziert sich das ganze auf $O(n)$. Dazu wird von 5er-Gruppen der Median (das mittlere Element) gebildet und danach wieder den Median dieser Mediane.

14 Weitere Beispiele für dynamische Programmierung

14.1 Kürzeste gemeinsame Oberfolge

1. Erzeugen Matrix T mit $m+1$ (länge 1. Wortes+1) Zeilen und $n+1$ Spalten (länge zweiten Wortes+1)
   $T[u,v]$ ist der Eintrag was die kürzeste Oberfolge ist, wenn u Zeichen des ersten und v Zeichen des zweiten Wortes verarbeitet wurden. Dabei ist $T[0,0]$ unser oberstes linkes Feld
2. Initialisiere $T[i,0] = i \ \forall i \in (0,...,m)$ und $T[0,j] = j \ \forall j \in (0,...,n)$
3. Bearbeite alle weiteren Felder von links nach rechts und von oben nach unten ab
   $T[i,j] = \begin{cases} T[i-1,j-1]+1, & a_i = b_j \\ \min(T[i-1,j]+1, T[i,j-1]+1), & a_i \neq b_j \end{cases}$
4. $T[m,n]$ ergibt Länge der kürzesten gemeinsamen Oberfolge

14.2 Längste gemeinsame Teilfolge

1. Erzeugen Matrix T mit $m+1$ (länge 1. Wortes+1) Zeilen und $n+1$ Spalten (länge zweiten Wortes+1)
   $T[u,v]$ ist der Eintrag was die längste Teilfolge ist, wenn u Zeichen des ersten und v Zeichen des zweiten Wortes verarbeitet wurden. Dabei ist $T[0,0]$ unser oberstes linkes Feld
2. Initialisiere $T[i,0] = 0 \ \forall i \in (0,...,m)$ und $T[0,j] = 0 \ \forall j \in (0,...,n)$
3. Bearbeite alle weiteren Felder von links nach rechts und von oben nach unten ab
   $T[i,j] = \begin{cases} T[i-1,j-1]+1, & a_i = b_j \\ \max(T[i-1,j], T[i,j-1]), & a_i \neq b_j \end{cases}$
4. $T[m,n]$ ergibt Länge der längsten gemeinsamen Teilfolge

15 Komplexität

15.1 ausgewählte Zusammenhänge

$$\begin{array}{rcll} n^k & \in & O(2^n) & k = 0,1,... \\ (\log n)^k & \in & O(n^\varepsilon) & \forall \varepsilon > 0, k = 0,1,... \\ 2^{\frac{n}{2}} & \in & O(2^n) & \\ \binom{n}{k} & \in & \Theta(n^k) & \text{für konstante k} \end{array}$$

15.2 wichtige Formeln

15.2.1 arithmetische Reihe

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

15.2.2 geometrische Reihe

$\sum_{i=0}^{n} q^i = \frac{q^{n+1}-1}{q-1}$

15.2.3 Logarithmen

$\log_b a = \frac{\log a}{\log b}$
$\log_b a = \frac{1}{\log_a b}$
$c^{\log_b a} = a^{\log_b c}$
$\log(a * b) = \log a + \log b$
$\log(\frac{a}{b}) = \log a - \log b$
$\log a^b = b * \log a$

15.3 Master-Theorem

Bei Rekursionsgleichungen der Form $T(n) = \sum_{i=1}^{m} T(a_i n) + \Theta(n^k)$

$$T(n) = \begin{cases} \Theta(n^k), & falls \sum_{i=1}^{m} \alpha_i^k < 1 \\ \Theta(n^k \log n), & falls \sum_{i=1}^{m} \alpha_i^k = 1 \\ \Theta(n^c), & falls \sum_{i=1}^{m} \alpha_i^k > 1 \end{cases}$$

Im Fall $\Theta(n^c)$ kann c über $\sum_{i=1}^{m} \alpha_i^c = 1$ bestimmt werden. Sind alle a_i gleich, so ist $c = -\frac{\log m}{\log \alpha}$.

15.4 Lösung einfacher Rekursionsgleichungen

$k \in \mathbb{N} \backslash \{0\}, d \in \mathbb{R}$
$T(n) = k * T(\frac{n}{2}) + n^d$
Tiefe des Aufrufbaums $\log_2 n$ mit k-Verzweigungen pro Knoten

$$\Rightarrow T(n) = \underbrace{\sum_{i=0}^{(\log_2 n)-1} k^i * \left(\frac{n}{2^i}\right)^d}_{innere\ Knoten} + \underbrace{T(1) * k^{\log_2 n}}_{Blätter}$$

Umformen (hier Beispielsweise für $k = 4$ und $d = 1$) und in Formel für quadratische Reihen einsetzen
$\Rightarrow T(n) = \sum_{i=0}^{(\log_2 n)-1} \frac{2^i * 2^i}{2^i} * n + T(1) * 4^{\log_2 n} = \frac{2^{\log_2 n}-1}{2-1} * n + T(1) * n^2$
$\Rightarrow T(n) = n^2 - n + T(1) * n^2$