Effiziente kombinatorische Algorithmen

gelesen von Prof. Wanka
im Wintersemester 2015/2016
an der FAU Erlangen

DRAFT - inoffizielles Skript - DRAFT

Mitschrift von Alexander Raß
Nachtrag: Zusatzthemen nach Videoaufzeichnung des Wintersemesters 2016/2017

16. Juni 2020

Inhaltsverzeichnis

0 Einige Definitionen und Bezeichnungen

Definition 0.1 *Sei $V = \{v_1, \ldots, v_n\}$ eine endliche Menge und $E \subseteq \mathcal{P}_2(V) = \{\{u, v\} \mid u, v \in V, u \neq v\}$.*

- *Dann heißt das geordnete Paar $G = (V, E)$ ein* **(endlicher, schlichter, ungerichteter) Graph**, *wobei V die* **Knotenmenge** *und E die* **Kantenmenge** *von G heißen.*
  Ist $e = \{u, v\} \in E$, so heißen u und v **benachbart** (adjazent) *in G und e und u bzw. e und v* **inzident**.

- *$d_G(u) = |\{e \in E \mid e \text{ ist mit } u \text{ inzident}\}|$ ist der* **Grad** *von u.*
  $\triangle(G) = \max\{d_G(u) \mid u \in V\}$ ist der **Grad** *von G.*

- *Statt $e = \{u, v\}$ schreiben wir auch: $u \overset{e}{-} v$. Eine Kantenfolge $w = (e_1, e_2, \ldots, e_k)$ mit $e_i \in E$, $1 \leq i \leq k$, heißt* **Weg** *von u nach v, falls $e_i = \{v_{i-1}, v_i\}$, $u = v_0$, $v = v_k$ und $u = v_0 \overset{e_1}{-} v_1 \overset{e_2}{-} v_2 \ldots v_{k-1} \overset{e_k}{-} v_k = v$.*
  Ist $u = v$, so heißt w ein **Kreis**.
  Enthält w keinen Knoten mehr als einmal, so heißt w **einfacher Weg**.
  Enthält w bis auf die beiden Enden keinen Knoten mehr als einmal, so ist w ein **einfacher Kreis**, *wenn $u = v$.*

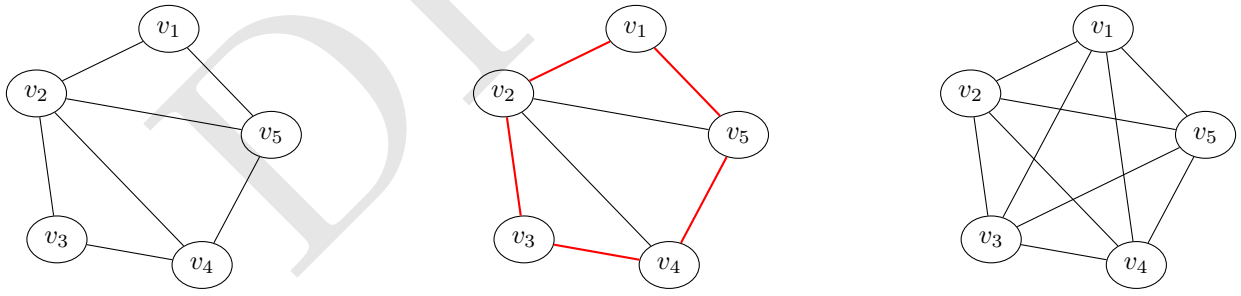
- *$G = (V, E)$ heißt* **zusammenhängend** (zhgd, connected), *falls zwischen je zwei Knoten u und v ein Weg existiert.*

- *G ist ein* **Wald** (forest), *falls G keine einfachen Kreise enthält.*
  G heißt **Baum** (tree), *falls G ein zusammenhängender Wald ist.*

- *Seien G und H Graphen. H heißt* **Teilgraph** (subgraph) *von G, falls $V(H) \subseteq V(G)$ und $E(H) \subseteq E(G)$.*
  H heißt **aufspannender Teilgraph** (spanning subgraph) *von G, falls H Teilgraph von G ist und $V(H) = V(G)$.*
  Ist ein aufspannender Teilgraph ein Wald (Baum), so heißt H **aufspannender Wald (Baum)** *von G.*

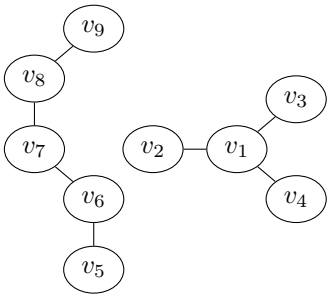
Beobachtungen:

$$\sum_{u \in V} d_G(u) = 2 \cdot |E| \qquad |E| \leq \binom{|V|}{2} = \frac{|V| \cdot (|V| - 1)}{2}$$

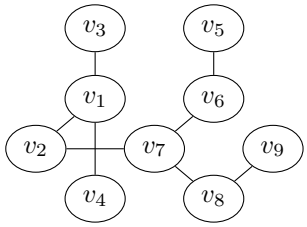


(a) Einfacher Graph (b) Einfacher Kreis im Graphen (c) Vollständiger Graph K_5

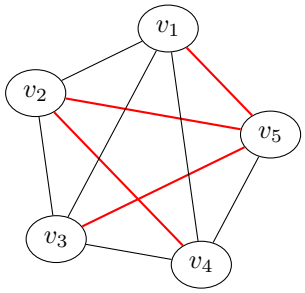
Abbildung 1: Beispielgraphen



(a) Wald



(b) Baum



(c) aufspannender Baum des K_5

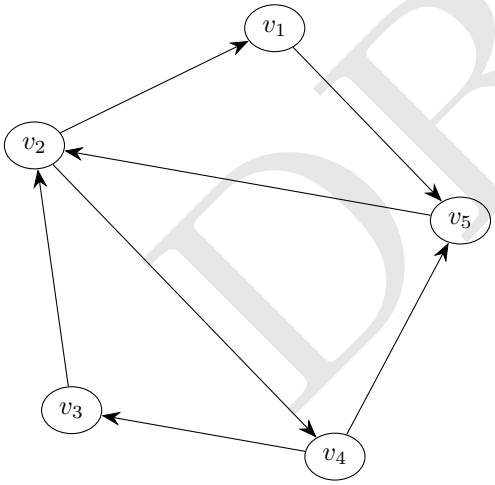
Abbildung 2: Wald und Baum

Definition 0.2 $V = \{v_1, \ldots, v_n\}$, $E \subseteq V \times V \setminus \{(v, v) \mid v \in V\}$.

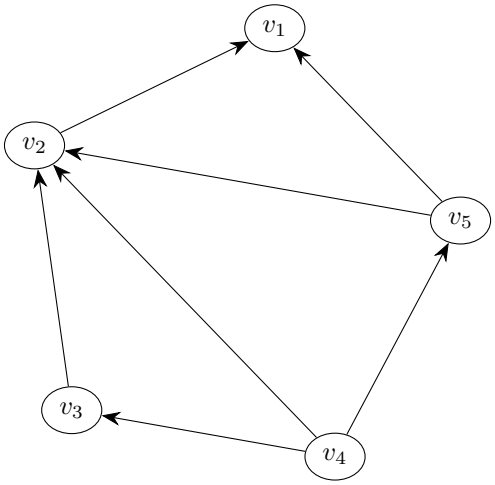
- $G = (V, E)$ *ist ein* **gerichteter Graph** (directed graph) *oder kurz* **Digraph**. *Begriffe für Digraphen sind analog zu den ungerichteten Graphen.*
- $d_G^+(v) = |\{e \in E \mid e = (v, w) = v \xrightarrow{e} w\}|$ *heißt* **Ausgangsgrad** (outdegree).
  $d_G^-(v) = |\{e \in E \mid e = (w, v) = w \xrightarrow{e} v\}|$ *heißt* **Eingangsgrad** (indegree).
- $G = (V, E)$ *Digraph. G heißt* **stark zusammenhängend** (strongly connected), *falls für je zwei Knoten $u, v \in V$ gilt: Es gibt einen einfachen gerichteten Weg von u nach v* und *von v nach u.*
  Falls G ungerichtet zusammenhängend ist, ist G **schwach zusammenhängend** (weakly connected).

Beobachtung:

$$\sum_{v \in V} d_G^-(v) = \sum_{v \in V} d_G^+(v) = |E|$$



(a) Stark zusammenhängend



(b) Schwach zusammenhängend

Abbildung 3: Gerichtete Graphen

1 Depth-First Search (DFS)

Sei $G = (V, E)$ ein Digraph.
Ziel: Systematisches Durchsuchen des Digraphen durch folgendes algorithmisches Schema.
$\mathcal{S}$ ist eine Teilmenge von V und enthält die bereits besuchten Knoten.

Algorithm 1: Algorithmisches Schema zum Durchsuchen von Digraphen

```
1 Starte in einem beliebigen, aber festen Knoten s ∈ V;
2 S:={s};
3 Markiere alle Kanten e ∈ E als UNBENUTZT;
4 while noch eine Kante u -e-> v UNBENUTZT mit u ∈ S do
5 |   Wähle so einen Knoten u ∈ S und eine UNBENUTZTe Kante u -e-> v;
6 |   S:=S ∪ {v};
7 |   Markiere e als BENUTZT;
8 end
```

Lemma 1.1 *Gestartet in $s \in V$ eines Digraphen $G = (V, E)$ gilt für die Knoten in $\mathcal{S}$ nach Terminierung*

$$\mathcal{S} = \{v \in V \mid \text{es gibt einen Weg von } s \text{ nach } v\} \ .$$

BEWEIS:
„$\subseteq$“: offensichtlich
„$\supseteq$“: $v \in V$ und v ist von s aus erreichbar in G.
Das heißt es gibt $v_0, \ldots, v_k \in V$ sodass $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k = v$.
Durch Induktion über i wird gezeigt, dass $v_i \in \mathcal{S}$, $0 \leq i \leq k$.
$i = 0 : s = v_0 \surd$
Zu zeigen: $v_i \in \mathcal{S} \Rightarrow v_{i+1} \in \mathcal{S}$
Annahme: $v_i \in \mathcal{S}$ und $v_{i+1} \notin \mathcal{S}$
$\Rightarrow$ Nach Terminierung ist $v_i \xrightarrow{e} v_{i+1}$ *UNBENUTZT*
im Widerspruch zum Abbruchkriterium in while Schleife. □

In ungerichteten Graphen: $u \overset{e}{-} v$ wird ersetzt durch $u \xrightarrow{e'} v$ und $v \xrightarrow{e''} u$.
Einige Details des algorithmischen Schemas sind unspezifiziert geblieben:

- Kodierung des Graphen?
- Wie ist $\mathcal{S}$ dargestellt?
- In welcher Reihenfolge werden $u \in \mathcal{S}$ und *UNBENUTZT*e Kanten $u \to v$ innerhalb der while Schleife ausgewählt?

Kodierung eines (Di-)Graphen $G = (V, E)$:

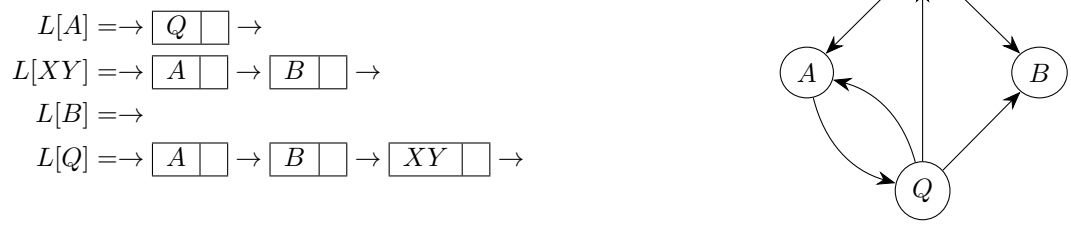


Abbildung 4: Repräsentation eines Graphen durch Inzidenzlisten

Repräsentation durch Inzidenzlisten:
Speicherplatz: $O(|V| + |E|)$; ist linear in der Größe des Graphen.
Repräsentation durch Adjazenzmatrix:
Speicherplatz: $O(|V|^2)$

Organisation des Kantendurchlaufs innerhalb der while Schleife:
$\mathcal{S}$ wird durch Array dargestellt.
$\forall s \in V : \mathcal{S}[v] \in \{BESUCHT, NICHT\_BESUCHT\}$
Initialisierung:
$\mathcal{S}[s] := BESUCHT$
$\forall v \neq s : \mathcal{S}[v] := NICHT\_BESUCHT$
Auswahl von $u \in \mathcal{S}$ und einer *UNBENUTZT*en Kante $u \xrightarrow{e} v$ geschieht mittels einem Keller (Stack) und der Inzidenzliste $L[u]$ in Verbindung mit den Kantenmarkierungen.

Algorithm 2: Algorithmisches Schema für eine Tiefensuche

```
1  Starte in einem beliebigen, aber festen Knoten s ∈ V;
2  for v ∈ V do
3  |  if v ≠ s then
4  |  |  S[v]:=NICHT_BESUCHT
5  |  else
6  |  |  S[v]:=BESUCHT
7  |  end
8  end
9  Lege s auf den Keller;
10 Markiere alle Kanten e ∈ E als UNBENUTZT;
11 while Keller nicht leer do
12 |  Nehme obersten Knoten u vom Keller;
13 |  if ∃ UNBENUTZTe Kante u -e-> v then
14 |  |  Markiere e als BENUTZT;
15 |  |  Lege u auf den Keller zurück;
16 |  |  if S[v] = NICHT_BESUCHT then
17 |  |  |  S[v] := BESUCHT;
18 |  |  |  Lege v auf den Keller;
19 |  |  end
20 |  end
21 end
```

Dieses Vorgehen nennt man Tiefensuche (Depth-First Search).

Der Algorithmus terminiert nach $O(|V| + |E|)$ Schritten.

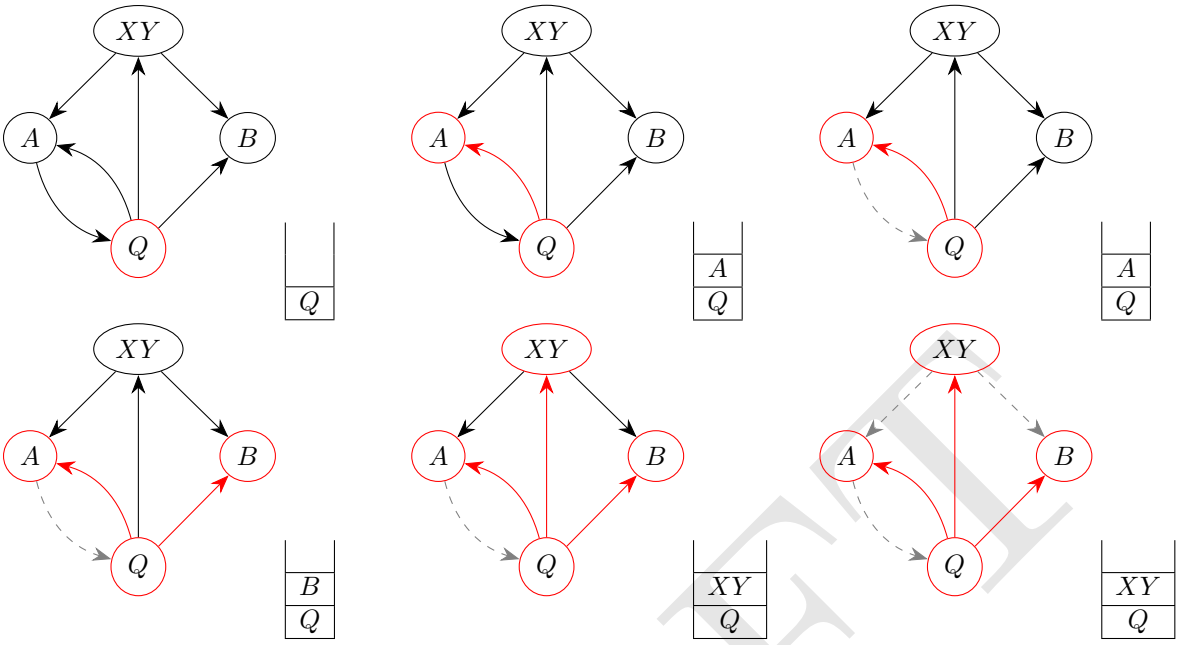
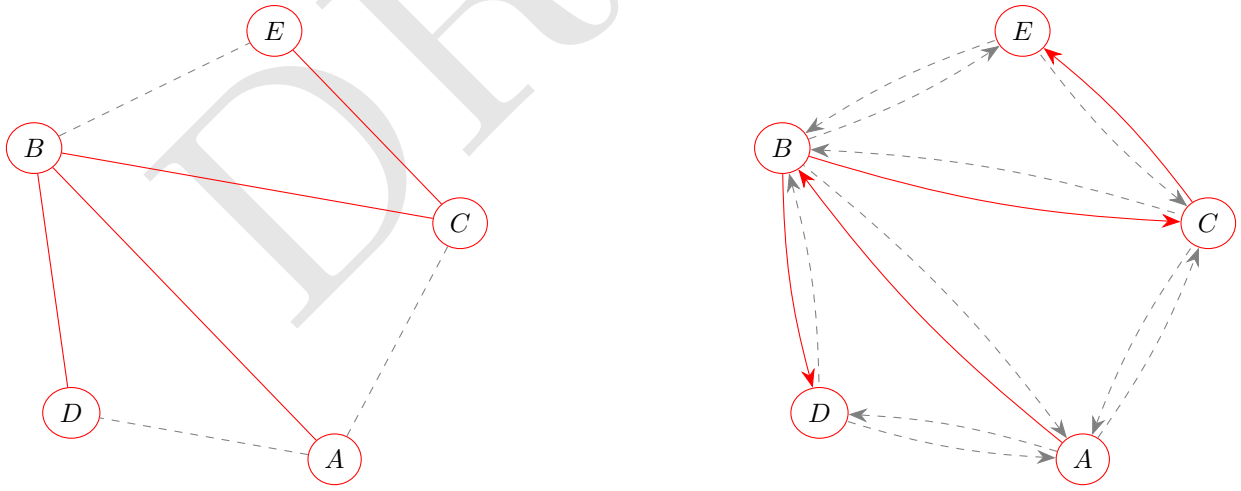


Abbildung 5: Leicht verkürzte Darstellung eines Durchlaufs einer Tiefensuche

Entfernt man aus der abschließenden Graphdarstellung aus Abbildung 5 die gestrichelten Kanten, die nicht benutzt wurden, so erhält man den Tiefensuchbaum, der zu der gezeigten Tiefensuche gehört.
In Abbildung 6 ist das Ergebnis der Tiefensuche in einem ungerichteten Graphen dargestellt. Die Reihenfolge der Kanten in den Inzidenzlisten ist dabei alphabetisch bezüglich Zielknoten und es wurde bei der Tiefensuche mit Knoten A begonnen.



(a) Ungerichtete Darstellung des Graphen

(b) Ungerichtete Kanten werden ersetzt durch je zwei gerichtete Kanten

Abbildung 6: Tiefensuche im ungerichteten Graphen

Definition 1.1 *Ein Graph H heißt* **(maximale) Zusammenhangskomponente** *des Graphen G, falls H ein Teilgraph von G ist, zusammenhängend ist und G keinen H echt enthaltenden zusammenhängenden Graphen enthält.*

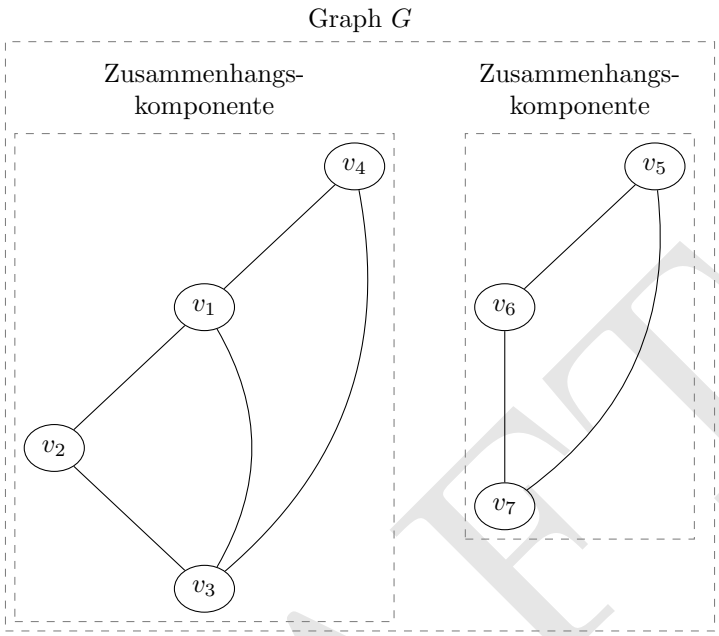


Abbildung 7: Graph mit zwei Zusammenhangskomponenten

Satz 1.1 *Mit dem Tiefensuche-Algorithmus kann ein Graph $G = (V, E)$ in Zeit $O(|V| + |E|)$ in seine maximalen Zusammenhangskomponenten zerlegt werden.* □

1.1 Einschub - Breitensuche (Breadth-Firts Search - BFS)

Algorithm 3: Breitensuche im Graphen, welche auch die *bfnr* berechnet

```
   /* bfnr enthält eine Numerierung der Knoten nach Reihenfolge in der sie besucht
      werden                                                                    */
 1 Starte in einem beliebigen, aber festen Knoten s ∈ V;
 2 for v ∈ V do
 3 |  if v ≠ s then
 4 |  |  S[v]:=NICHT_BESUCHT
 5 |  else
 6 |  |  S[v]:=BESUCHT
 7 |  end
 8 end
 9 T:=∅;
10 zaehler:=1;
11 Füge s an das Ende der Warteschlange (Queue) hinzu;
12 Markiere alle Kanten e ∈ E als UNBENUTZT;
13 while Warteschlange nicht leer do
14 |  Nehme vordersten Knoten u von Warteschlange;
15 |  bfnr[u]:=zaehler;
16 |  zaehler:=zaehler + 1;
17 |  while ∃ UNBENUTZTe Kante u --e--> v do
18 |  |  Markiere e als BENUTZT;
19 |  |  if S[v] = NICHT_BESUCHT then
20 |  |  |  S[v] := BESUCHT;
21 |  |  |  T:=T ∪ {(u, v)};
22 |  |  |  Füge v an das Ende der Warteschlange (Queue) hinzu;
23 |  |  end
24 |  end
25 end
26 return T, s, bfnr;
```

Das Vorgehen in Algorithmus 3 nennt man Breitensuche (Breadth-First Search).
Das Resultat des Durchlaufs einer Breitensuche zum Beispiel 1.1 ist in Abbildung 8 dargestellt. Bei der Breitensuche kann es wie in Abbildung 8 ersichtlich vorkommen, dass gestrichelte Kanten auch zwei verschiedene Äste verbinden.

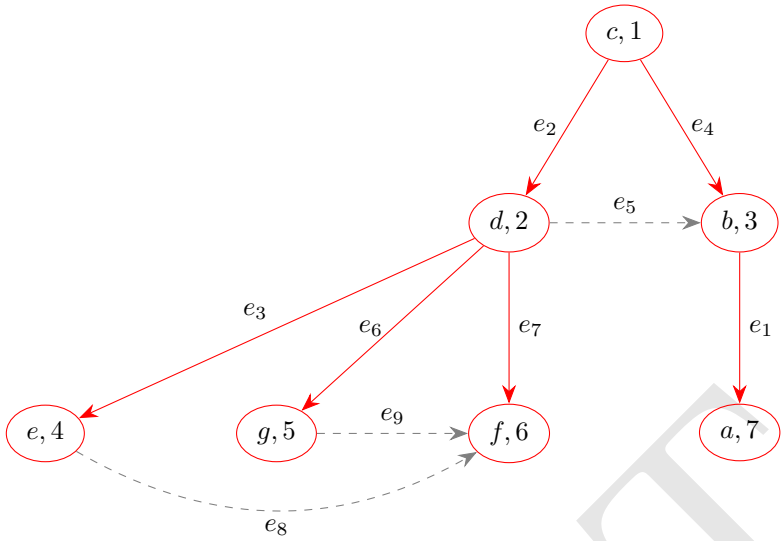


Abbildung 8: Breitensuchbaum, Knoten sind mit *bfnr* abgebildet, c ist Startknoten der BFS

Algorithm 4: modifizierte BFS im Graphen, welche eine topologische Sortierung berechnet *topsortnr*

```
   /* topsortnr enthält eine Numerierung der Knoten, welche eine topologische
      Reihenfolge darstellt                                                    */
   /* in_degree berechnet zu jedem Knoten u die Anzahl der Kanten, die von einem noch
      nicht bearbeiteten Knoten v nach u führen.                               */
 1 for v ∈ V do
 2 |   topsortnr[v]:=0;
 3 |   in_degree[v]:=d⁻_G(v);
 4 |   if in_degree[v] = 0 then
 5 |   |   Füge v an das Ende der Warteschlange (Queue) hinzu;
 6 |   end
 7 end
   /* zaehler sei die Anzahl der bearbeiteten Knoten                           */
 8 zaehler:=0;
 9 while Warteschlange nicht leer do
10 |   Nehme vordersten Knoten u von Warteschlange;
11 |   zaehler:=zaehler + 1;
12 |   topsortnr[u]:=zaehler;
   |   /* Entferne alle Kanten die in u starten.                               */
13 |   for Kante e ∈ E : e = (u → v) do
14 |   |   in_degree[v]:=in_degree[v] − 1;
15 |   |   if in_degree[v] = 0 then
16 |   |   |   Füge v an das Ende der Warteschlange (Queue) hinzu;
17 |   |   end
18 |   end
19 end
   /* Falls keine topologische Sortierung möglich ist, so ist der zaehler ≠ |V|. */
20 return zaehler, topsortnr ;
```

1.2 DFS für (ungerichtete) Graphen und die schnelle Berechnung der zweifachen Zusammenhangskomponenten

Sei $G = (V, E)$ immer ein zusammenhängender Graph.
Nachfolgende rekursive Version der DFS numeriert die Knoten in der Reihenfolge, in der sie erstmals besucht werden (von $1, \ldots, |V|$). Das Ergebnis der Numerierung wird in dem Array *dfnr* gespeichert. Gleichzeitig wird eine Kantenmenge T berechnet. In der Menge T werden die gerichteten Kanten des Tiefensuchbaums abgelegt.

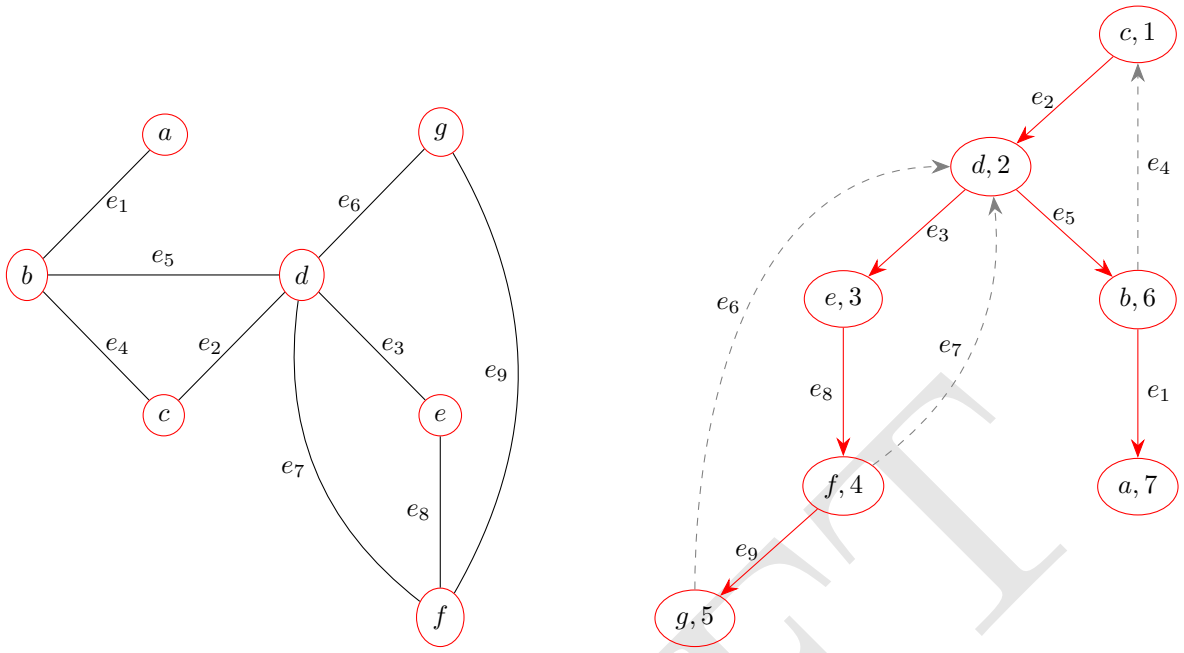
Algorithm 5: Tiefensuche für ungerichtete Graphen (Hopcroft/Tarjan)

```
1  Procedure Suche(v : V)
2  |  dfnr[v]:=zaehler;
3  |  zaehler:=zaehler + 1;
4  |  Markiere v als BESUCHT;
5  |  for w ∈ L[v] do
6  |  |  if w ist NICHT_BESUCHT then
7  |  |  |  T:=T ∪ {(v, w)};
8  |  |  |  Suche(w);
9  |  |  end
10 |  end
11 Algorithm DFS_u()
12 |  T:=∅;
13 |  zaehler:=1;
14 |  for v ∈ V do
15 |  |  Markiere v als NICHT_BESUCHT
16 |  end
17 |  Wähle r ∈ V;
18 |  Suche(r);
19 |  return T, r, dfnr;
```

Beispiel 1.1 *$G = (V, E)$ mit $V = \{a, b, c, d, e, f, g\}$ und*
$E = \{a \overset{e_1}{-} b, c \overset{e_2}{-} d, d \overset{e_3}{-} e, b \overset{e_4}{-} c, b \overset{e_5}{-} d, d \overset{e_6}{-} g, d \overset{e_7}{-} f, e \overset{e_8}{-} f, f \overset{e_9}{-} g\}$.
Die Kanten liegen aufsteigend nach Kantenindex sortiert in den jeweiligen Inzidenzlisten:

$L[a] = \rightarrow \boxed{b\,|\,} \rightarrow$
$L[b] = \rightarrow \boxed{a\,|\,} \rightarrow \boxed{c\,|\,} \rightarrow \boxed{d\,|\,} \rightarrow$
$L[c] = \rightarrow \boxed{d\,|\,} \rightarrow \boxed{b\,|\,} \rightarrow$
$L[d] = \rightarrow \boxed{c\,|\,} \rightarrow \boxed{e\,|\,} \rightarrow \boxed{b\,|\,} \rightarrow \boxed{g\,|\,} \rightarrow \boxed{f\,|\,} \rightarrow$
$L[e] = \rightarrow \boxed{d\,|\,} \rightarrow \boxed{f\,|\,} \rightarrow$
$L[f] = \rightarrow \boxed{d\,|\,} \rightarrow \boxed{e\,|\,} \rightarrow \boxed{g\,|\,} \rightarrow$
$L[g] = \rightarrow \boxed{d\,|\,} \rightarrow \boxed{f\,|\,} \rightarrow$

Das Ergebnis der Tiefensuche DFS_u gestartet im Knoten c ist in Abbildung 9 visualisiert.
Im Tiefensuchbaum steht in jedem Knoten der Knotenname und die dfnr.
Die roten durchgezogenen Kanten sind die Kanten des Tiefensuchbaums
$T = \{e_2, e_3, e_8, e_9, e_5, e_1\}$.
Die grauen gestrichelten Kanten sind die unbenutzten Rückkanten
$B = E \setminus T = \{e_7, e_6, e_4\}$.



(a) Graph G

(b) Tiefensuchbaum, Knoten sind mit *dfnr* abgebildet

Abbildung 9: Beispielgraph - DFS_u wird bei c gestartet

Gestrichelte Kanten von einem Knoten v zeigen immer zu einem Knoten, der auf dem direkten Weg von v zur Wurzel liegt. Die gestrichelten Kanten springen also nur auf Ästen des Baums T zurück. Es gibt keine Kanten zwischen Ästen. Dies ist eine fundamentale Eigenschaft der Tiefensuche in **ungerichteten** Graphen!

Lemma 1.2 *Sei G ein zusammenhängender Graph. Dann berechnet DFS_u in Zeit $O(|V|+|E|)$ die Funktion* dfnr *und eine Zerlegung von E in T und $B = E \setminus T$, wobei gilt:*

i) Der (gerichtete) Graph $H = (V, T)$ ist ein Baum mit Wurzel r, wobei $dfnr[r] = 1$ ist.

ii) Ist $(v \xrightarrow{e} w) \in B$ („Rückkante“, $dfnr[v] > dfnr[w]$), so ist v in H Nachfolger von w („v liegt auf einem Ast von w“).

BEWEIS:

i) $\surd$

ii) $(v \to w) \in B \overset{\text{z.z.}}{\Rightarrow} v$ ist Nachfolger von w in $H(V, T)$
Annahme: v ist kein Nachfolger von w in H.
Situation: Als die Liste $L[w]$ (Inzidenzliste des Knotens w) durchlaufen wurde, war v noch als *NICHT_BESUCHT* markiert, aber $v \in L[w]$, muss daher betrachtet worden sein und hätte spätestens dann als *BESUCHT* markiert werden müssen $\Rightarrow$ ↯

□

Definition 1.2 *$G = (V, E)$ zusammenhängender Graph.*
$v \in V$ heißt **Artikulationspunkt** (AP, Zerfällungspunkt, articulation point) *von G, falls es Knoten $x, y \in V$ gibt mit $x \neq v \neq y$, $x \neq y$, sodass jeder Weg in G zwischen x und y den Knoten v enthält.*
G heißt **zweifach zusammenhängend** (biconnected)*, falls G keinen Artikulationspunkt enthält.*

Sei $V' \subseteq V$, $G' = G|_{V'}$ (der durch V' auf G induzierte Teilgraph) heißt **zweifache Zusammenhangskomponente** (ZHK), *falls G' zweifach zusammenhängend ist und es kein $V'' \subseteq V$ mit $V' \subsetneq V''$ gibt, sodass $G|_{V''}$ zweifach zusammenhängend ist.*

Alternative (äquivalente) Definition eines Artikulationspunkts:
$G = (V, E)$ zusammenhängender Graph. $v \in V$ ist ein Artikulationspunkt, wenn der Restgraph $G|_{V \setminus \{v\}}$ in mehrere Zusammenhangskomponenten zerfällt (mindestens zwei).
Ziel: Finde die zweifachen Zusammenhangskomponenten in linearer Zeit $O(|V| + |E|)$.
Lösungsansatz:
(a) Enthält der Eingabegraph G keinen Artikulationspunkt, so ist G zweifach zusammenhängend.
(b) Ist v ein Artikulationspunkt, so besteht $G \setminus \{v\}$ aus zusammenhängenden Teilgraphen $G_i = (V_i, E_i)$ mit $\bigcup_i V_i = V \setminus \{v\}$. Gehe mit allen G_i zu (a).

Seien *dfnr*, T, B, r von DFS_u zum zusammenhängenden Eingabegraphen G berechnet.
Zu $v \in V$:
tief(v) (engl. lowpoint): kleinste Zahl *dfnr*$[u]$, die erreichbar ist durch einen gerichteten Weg von v aus in $H = (V, T)$ gefolgt von **höchstens** einer Rückkante aus B.
In Abbildung 10 wird der Tiefensuchbaum aus Beispiel 1.1 dargestellt mit berechnetem Wert *tief*.

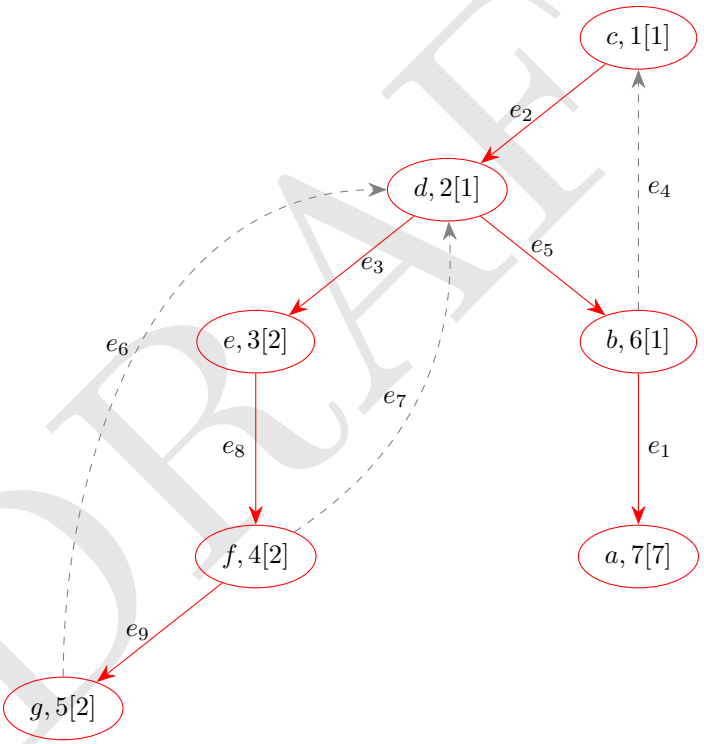


Abbildung 10: Beispielgraph nach Tiefensuche DFS_u aus Beispiel 1.1 mit *dfnr* (nach Komma) und *tief* (in eckigen Klammern [])

Offensichtlich: *tief*$(v) \leq$ *dfnr*$[v]$. In Abbildung 10 ist der Knoten d ein Artikulationspunkt, weil *tief*$(e) \not<$ *dfnr*$[d]$

Lemma 1.3 *Seien* dfnr, B, T, r *von DFS_u zu $G = (V, E)$ berechnet. Falls Baumkante $(u \to v) \in T$ existiert mit* $\text{dfnr}[u] > 1$ *und* $\text{tief}(v) \geq \text{dfnr}[u]$, *so ist u ein Artikulationspunkt.*

BEWEIS: Sei S die Menge der Knoten in $H = (V, T)$ auf dem Weg von r zu u, u ausgenommen. V_v sei die Menge der Knoten, die zum Teilbaum gehören, der an v hängt, inklusive v. Wegen Lemma 1.2 gibt es keine Kante $e \in E$, die zwei Knoten aus V_v und $V \setminus (S \cup \{u\} \cup V_v)$ verbindet. Das heißt alle Kanten e inzident zu Knoten aus V_v, die nicht Knoten aus V_v verbinden, sind inzident zu Knoten in $S \cup \{u\}$.
Annahme: u ist kein Artikulationspunkt.
Dann gibt es einen Weg von v über Baumkanten zu einem Knoten v' in V_v und von dort aus eine Rückkante zu einem Knoten $w \in S$. $\Rightarrow$ *tief*$(v) \leq$ *dfnr*$[w] <$ *dfnr*$[u]$ ↯ □

Lemma 1.4 *Seien dfnr, B, T, r von DFS_u zu $G = (V, E)$ berechnet. Sei u ein Artikulationspunkt von G und $u \neq r \Rightarrow$ es gibt eine Baumkante $(u \to v) \in T$ mit* $\text{tief}(v) \geq \text{dfnr}[u]$.

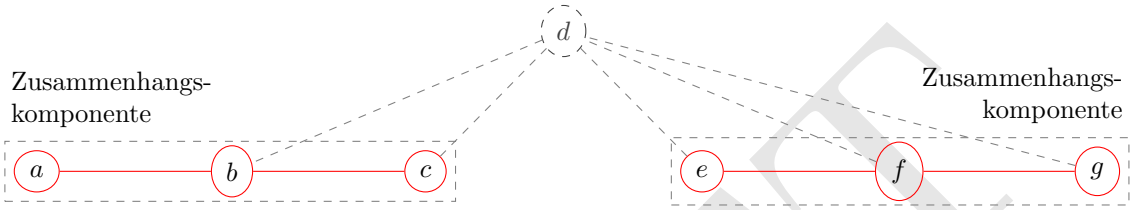


Abbildung 11: Maximale Zusammenhangskomponenten nach dem Entfernen des Knotens d aus dem Graphen zum Beispiel 1.1

BEWEIS: $G \setminus \{u\}$ besteht aus den maximalen Zusammenhangskomponenten $G_1, \ldots, G_m, m \geq 2$, und jeder Weg in G zwischen Knoten aus $V(G_i)$ und $V(G_j)$, $i \neq j$, enthält u.
Ohne Beschränkung der Allgemeinheit $r \in V(G_1)$.
Jede Suche von DFS_u von r aus gelangt über Baumkanten zu u. Sei $u \to v$ erste Baumkante mit $v \notin G_1$.
Ohne Beschränkung der Allgemeinheit $v \in V(G_2)$.
Da keine Kanten zwischen $V(G_2)$ und $V \setminus (V(G_2) \cup \{u\})$ gibt, gilt: *tief*$(v) \geq$ *dfnr*$[u]$ □

Lemma 1.5 *Voraussetzungen wie üblich.*
Wurzel r ist Artikulationspunkt $\Leftrightarrow$ $\underbrace{d_H^+(r)}_{\text{Ausgangsgrad der Wurzel im Tiefensuchbaum}} \geq 2$

Definition 1.3 *Sei $G = (V, E)$ ein zusammenhängender Graph. Seien $s_1, \ldots, s_p$ alle Artikulationspunkte von G und $C_1, \ldots, C_m$ die zweifachen Zusammenhangskomponenten von G.*
Dann heißt $\tilde{G}$ mit $V(\tilde{G}) = \{s_1, \ldots, s_p\} \cup \{C_1, \ldots, C_m\}$ und $E(\tilde{G}) = \{s_i - C_j \mid s_i$ Knoten in $C_j\}$ **Superstrukturgraph**.

Satz 1.2 *Zu zusammenhängenden Graphen G ist der zugehörige Superstrukturgraph $\tilde{G}$ ein Baum.* □

Falls ein Graph Artikulationspunkte besitzt, so hat $\tilde{G}$ **mindestens zwei** Blätter (Knoten mit Grad 1). Den Blättern entsprechen zweifache Zusammenhangskomponenten. Ist C eine zweifache Zusammenhangskomponente, die ein Blatt im Superstrukturgraph $\tilde{G}$ ist, so ist C mit dem Restgraphen durch genau einen Artikulationspunkt u verbunden.

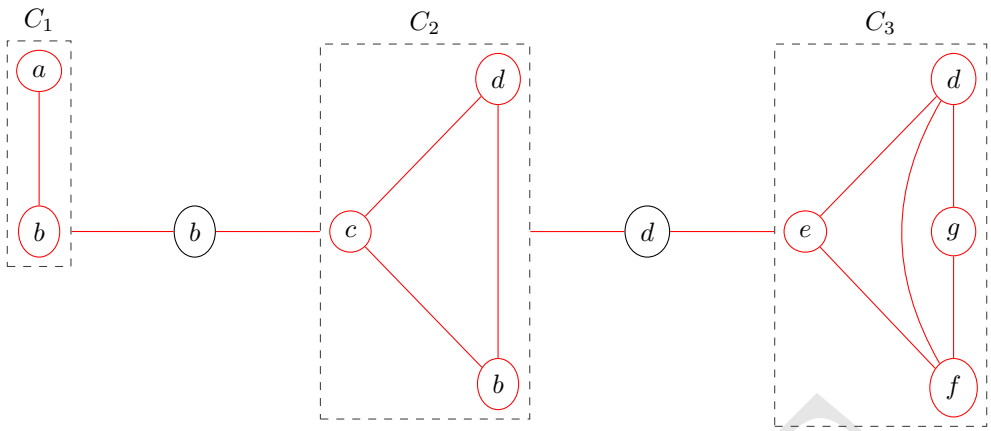


Abbildung 12: Superstrukturgraph zu dem Graphen aus Beispiel 1.1

2 Fälle:

- Startknoten r ist kein Artikulationspunkt:
  Dann wird DFS_u von r über Baumkanten zu einer ersten zweifachen Zusammenhangskomponente C gelangen, die ein Blatt von $\tilde{G}$ ist. C wird über Artikulationspunkt u, gefolgt von $(u \to v) \in T$ betreten und anschließend über „$v \leftarrow u$“ (Rekursionsende) rückwärts verlassen. Ist zum Zeitpunkt des Verlassens von C über „$v \leftarrow u$“ *tief*(v) berechnet, so kann mit Lemma 1.3 u als Artikulationspunkt identifiziert werden. Zwischen erstem Eintreten in C über $u \to v$ und Verlassen von C über „$v \leftarrow u$“ lässt sich die Rechnung von DFS_u auffassen als Rechnung auf eigenem Graphen C gestartet in u.
  Das heißt mit Lemma 1.5: C enthält nur eine Baumkante $u \to v$, die von u ausgeht. Merkt man sich die zwischen Eintreten und Verlassen besuchten Knoten und Kanten auf Stacks mit den Bezeichnungen $BESICHTIGTE\_KNOTEN$ beziehungsweise $BESICHTIGTE\_KANTEN$, so lässt sich zum Zeitpunkt, da u als Artikulationspunkt erkannt wird, die Knoten- und die Kantenmenge von C mit Hilfe der Stacks ausgeben: Von $BESICHTIGTE\_KNOTEN$ gib alles oberhalb von u aus (u wird auch „hingeschrieben“, aber auf dem Keller gelassen). Von $BESICHTIGTE\_KANTEN$ gib alles oberhalb von $u \to v$ einschließlich $u \to v$ aus. Von u wird die Tiefensuche fortgesetzt und eine nächste zweifache Zusammenhangskomponente C' von G, die Blatt von dem Teilgraphen ist, der durch die Knoten $(V \setminus V(C)) \cup \{u\}$ induziert wird, gefunden. Und so weiter...
  (Die Komponente, die den Startknoten r enthält, wird auch ausgegeben, weil als Kriterium für die Erkennung zur Ausgabe einer Komponente die Bedingung aus Lemma 1.4 benutzt wird [auch für r])
- Startknoten r ist ein Artikulationspunkt:
  Alle zweifachen Zusammenhangskomponenten, die r nicht enthalten, sind bereits wie im ersten Fall gefunden und ausgegeben. Wenn zu Baumkante $(r \to v) \in T$ die Tiefensuche von v aus beendet ist und mindestens eine zweite, noch nicht besuchte Kante $(r \to u)$ existiert, so ist r nach Lemma 1.5 ein Artikulationspunkt. Dann nimm jeweils alle Knoten bis zu v und alle Kanten bis zu $(r \to v)$ von den Stacks, $(r \to v)$ ist die erste Kante oben auf dem Stack, die Baumkante ist. Die ausgegebenen Knoten inklusive r und Kanten bilden eine zweifache Zusammenhangskomponente. Wiederhole dies, bis alle Kanten besucht worden sind.

Berechnung von *tief*(v):
Falls v Blatt des Tiefensuchbaums ist

$$\mathit{tief}(v) = \min(\{\mathit{dfnr}[u] \mid u = v \text{ oder } v \to u \text{ ist Rückkante}\}) \ .$$

Falls v kein Blatt ist

$$\mathit{tief}(v) = \min(\{\mathit{dfnr}[u] \mid u = v \text{ oder } v \to u \text{ ist Rückkante}\} \cup \{\mathit{tief}(u) \mid (v \to u) \in T\}) \ .$$

Algorithm 6: DFS-2ZK

Input : Zusammenhängender Graph $G = (V, E)$ durch Inzidenzlisten L.
Output: Die zweifachen Zusammenhangskomponenten von G.

```
1  Procedure Suche(v : V)
2      Markiere v als BESUCHT;
3      tief[v]:=dfnr[v]:=zaehler;
4      zaehler:=zaehler + 1;
5      for w ∈ L[v] do
6          if w ist NICHT_BESUCHT then
7              T:=T ∪ {(v → w)};
8              vater[w] := v;
9              lege w auf den Keller BESICHTIGTE_KNOTEN;
10             lege (v → w) auf den Keller BESICHTIGTE_KANTEN;
11             Suche(w);
               /* Test der Bedingung, die v als Artikulationspunkt nachweist          */
12             if tief[w] ≥ dfnr[v] and v ≠ r then
                   /* Ausgabe der 2fachen Zusammenhangskomponente, die r nicht enthält */
13                 gib alle Knoten oben vom Keller BESICHTIGTE_KNOTEN bis w einschließlich aus;
14                 gib v aus                          /* v liegt noch immer auf dem Keller. */
15                 gib alle Kanten oben vom Keller BESICHTIGTE_KANTEN bis einschl. (v → w) aus;
16                 schreibe „Ende einer zweifachen Zusammenhangskomponente“;
17             end
18             tief[v]:=min{tief[v], tief[w]};
19         else if w ≠ vater[v] and dfnr[v] > dfnr[w] then
20             lege (v → w) auf BESICHTIGTE_KANTEN;
21             tief[v]:=min{tief[v], dfnr[w]};
22         end
23     end
24 Algorithm DFS-2ZK()
25     T:=∅;
26     zaehler:=1;
27     for v ∈ V do
28         Markiere v als NICHT_BESUCHT
29     end
30     Wähle Startknoten r ∈ V;
31     Suche(r) ;          /* Alle zweifachen Zusammenhangskomponenten, die r enthalten, */
32                         /* sind noch nicht ausgegeben worden                          */
33     while Keller BESICHTIGTE_KANTEN nicht leer do
34         gib alle Kanten oben auf BESICHTIGTE_KANTEN bis zur ersten Kante (r → v) ∈ T
             einschließlich aus;
35         gib alle Knoten von BESICHTIGTE_KNOTEN bis v einschließlich aus;
36         gib r aus;
37         schreibe „Ende einer zweifachen Zusammenhangskomponente“;
38     end
```

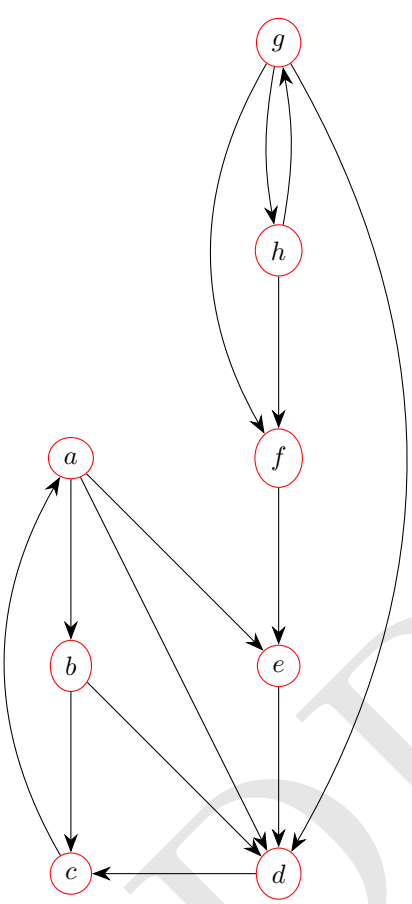
Satz 1.3 *DFS-2ZK berechnet in Zeit $O(|V| + |E|)$ alle zweifachen Zusammenhangskomponenten eines zusammenhängenden Graphen $G = (V, E)$.*

BEWEIS: „Alles, was wir bislang gemacht haben.“ □

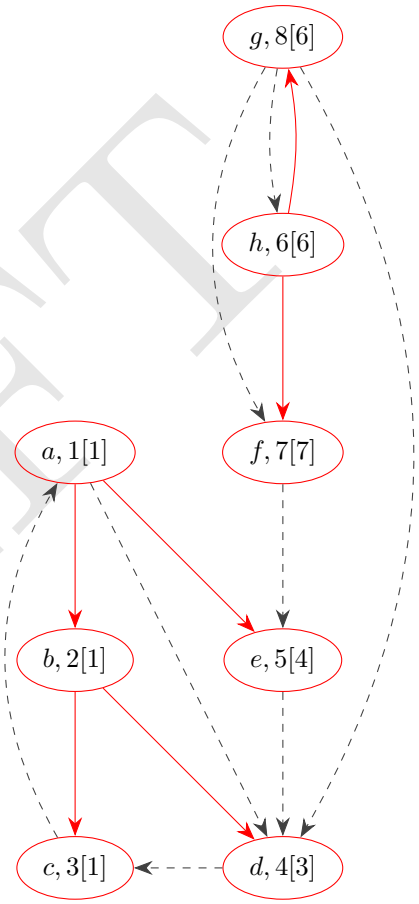
1.3 DFS für Digraphen und schnelle Berechnung der starken Zusammenhangskomponenten

In DFS hier neu:
while es gibt *NICHT_BESUCHT*e Knoten $r \in V$ *do*
    suche(r);

DFS berechnet zu G den Wald $H = (V, T), T \subseteq E$.



(a) Digraph G



(b) Tiefensuchwald H, unbenutzte Kanten $e \in E \setminus T$ sind grau und gestrichelt, Knoten sind mit *dfnr* (nach Komma) und *tief* (in eckigen Klammern []) abgebildet

Abbildung 13: Beispielgraph - *Suche* wird bei a und h gestartet

DFS produziert vier Kantentypen:

(i) Die Kanten aus T sind **Baumkanten** und führen zu jeweils neuen gesuchten Knoten.

(ii) **Vorwärtskanten**: Kanten $(u \to v) \in E \setminus T$ mit $u \to u_1 \to \ldots \to u_n \to v$ ist Weg in H, $n \geq 1$.
(Im Beispiel von Abbildung 13 $(a \to d)$)

(iii) **Rückkanten**: Kanten $(u \to v) \in H \setminus T$ mit $v \to u_1 \to \ldots \to u_k \to u$ ist Weg in H, $k \geq 0$.
(Im Beispiel von Abbildung 13 $(c \to a), (g \to h)$)

(iv) **Crosskanten**: Kanten $(u \to v) \in E \setminus T$, wobei es in H weder einen Weg von u nach v noch von v nach u gibt.
(Im Beispiel von Abbildung 13 $(d \to c), (e \to d), (f \to e), (g \to d), (g \to f)$)

Lemma 1.6 *Ist $u \to v$ Crosskante, so gilt:* $\text{dfnr}[u] > \text{dfnr}[v]$.

Definition 1.4

(a) Sei $G = (V, E)$ ein Digraph, und „$\sim$“ sei Äquivalenzrelation auf V, definiert durch:
$u \sim v \Leftrightarrow$ In G gibt es Wege von u nach v und von v nach u.
Sei $V_1, V_2, \ldots, V_k$ Zerlegung von V unter „$\sim$“ in die Äquivalenzklassen. Dann heißen die durch die V_i induzierten Teilgraphen $G_i = G|_{V_i}$, $1 \leq i \leq k$, **starke Zusammenhangskomponente** *von G.*

(b) Seien $C_1, \ldots, C_k$ die starken Zusammenhangskomponenten von G. Dann heißt der Digraph $\tilde{G}$ mit

$$V(\tilde{G}) = \{C_1, \ldots, C_k\}$$

$$E(\tilde{G}) = \{C_i \to C_j \mid i \neq j, \exists (v_l \to v_m) \in E : v_l \in V(C_i) \text{ und } v_m \in V(C_j)\}$$

der **Superstrukturgraph** *von G.*

Fakt: $\tilde{G}$ ist ein kreisfreier Graph (DAG: directed acyclic graph)

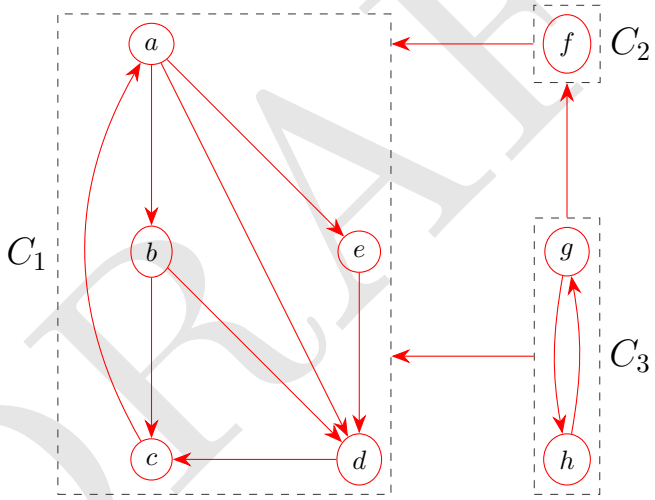


Abbildung 14: Superstrukturgraph zu dem Graphen aus Abbildung 13

DFS, gestartet in Knoten r von G, der zur starken Zusammenhangskomponente C_1 gehört, wird C_1 verlassen, ehe C_1 vollständig durchsucht worden ist, genau dann wenn $D^+_{\tilde{G}}(C_1) \geq 1$. DFS gelangt in diesem Fall zur starken Zusammenhangskomponente C_2 und so weiter bis schließlich DFS in eine erste starke Zusammenhangskomponente C_k gelangt mit $d^+_{\tilde{G}}(C_k) = 0$ (Die muss es geben, da $\tilde{G}$ ein DAG ist). C_k wird vom DFS vollständig durchsucht, bevor C_k „rückwärts“ wieder verlassen wird.

Identifiziere C_k mittels modifizierter Funktion *tief*:
tief(v) : kleinste Zahl *dfnr*$[u]$ eines Knotens u, der in derselben starken Zusammenhangskomponente wie v liegt und der erreichbar ist von v aus auf (gegebenenfalls leerem) Weg über Baumkanten, gefolgt von höchstens einer Rückkante oder Crosskante.

Induktives Berechnungsschema für *tief*, $H = (V, T)$ Tiefensuchwald.

(i) $v \in V : d_H^+(v) = 0$:

$$tief(v) = \min(\{dfnr[v]\} \cup \{dfnr[u] \mid v \to u \text{ Rückkante oder Crosskante, } u \text{ in gleicher Zusammenhangskomponente wie } v\})$$

(ii) $v \in V : d_H^+(v) \geq 1$:

$$tief(v) = \min(\{dfnr[v]\} \cup \{dfnr[u] \mid v \to u \text{ Rückkante oder Crosskante, } u \text{ in gleicher Zusammenhangskomponente wie } v\} \cup \{tief(u) \mid (v \to u) \in T\})$$

Algorithm 7: DFS-SZK

Input : Digraph $G = (V, E)$ durch Inzidenzlisten L.
Output: Die starken Zusammenhangskomponenten von G.

```
/* Zerlegt den Digraphen G = (V,E) in starke Zusammenhangskomponenten */
Procedure Suche(v : V)
    Markiere v als BESUCHT;
    tief[v]:=dfnr[v]:=zaehler;
    zaehler:=zaehler + 1;
    Lege v auf den Keller K;
    AUF_KELLER[v]:=true;
    for w ∈ L[v] do
        if w ist NICHT_BESUCHT then
            Suche(w);
            tief[v]:=min{tief[v], tief[w]};
        else if dfnr[w] < dfnr[v] and AUF_KELLER[w] then
            tief[v]:=min{tief[v], dfnr[w]};
        end
    end
    if tief[v] = dfnr[v] then
        while AUF_KELLER[v] do
            Nehme obersten Knoten x vom Keller K;
            Gebe x aus;
            AUF_KELLER[x]:=false;
        end
        Schreibe „Ende einer starken Zusammenhangskomponente“;
    end
Algorithm DFS-SZK()
    zaehler:=1;
    K:=empty stack;
    for v ∈ V do
        Markiere v als NICHT_BESUCHT;
        AUF_KELLER[v]:=false;
    end
    while Es gibt Knoten r ∈ V mit r NICHT_BESUCHT do
        Suche(r);
    end
```

Lemma 1.7 *Sei t der erste Knoten, der an der rot markierten Stelle (Zeile 15) des Algorithmus* DFS-SZK *die* if*-Bedingung (*tief$[t]$ = dfnr$[t]$*) erfüllt. Dann bilden genau die Knoten oberhalb von t (inklusive t) auf dem Keller K eine starke Zusammenhangskomponente.*

BEWEIS:

(1) Von t aus sind alle Knoten oberhalb von t auf K über Baumkanten erreichbar, da zu diesem Zeitpunkt die Suche von t aus beendet ist.

(2) zu zeigen: Ist v von t aus über Baumkanten erreichbar, und v auf K, so gibt es einen Weg von v zu t. Einen solchen Weg konstruieren wir jetzt:
Zum Zeitpunkt, zu dem die Suche von t aus beendet ist, ist auch die Suche von v aus beendet und es gilt: $dfnr[v] > tief(v) \geq tief(t) = dfnr[t]$. Sei u Knoten mit $dfnr[u] = tief(v)$. Ist $u = t$, so sind wir fertig. Sonst ist u auf K oberhalb von t, und wir argumentieren für u wie gerade für v. So ergibt sich ein Weg in G von v nach t.
(1)+(2)$\Rightarrow$ Alle Knoten oberhalb von t auf K liegen in derselben starken Zusammenhangskomponente.

(3) zu zeigen: C enthält keine weiteren Knoten.
Annahme: C enthält y, der zum gewählten Zeitpunkt *nicht* oberhalb von t auf K liegt$\Rightarrow$ $dfnr[y] < dfnr[t]$. Alle Knoten $y \in V(C)$, die nicht über t auf K liegen, haben diese Eigenschaft. Mindestens ein solcher Knoten $y' \in V(C)$ ist über Rückkante oder Crosskante erreichbar: $x \to y'$ von x oberhalb von t auf K aus. Da x, y', $t \in V(C)$ sind, gilt: $tief(t) \leq tief(x) \leq dfnr[y'] < dfnr[t]$ ↯

□

Satz 1.4 DFS-SZK *berechnet in Zeit $O(|V| + |E|)$ zum Graphen $G = (V, E)$ die starken Zusammenhangskomponenten.*

BEWEIS: Es wird immer nach Lemma 1.7 eine starke Zusammenhangskomponente korrekt berechnet mit $d^+_{\tilde{G}}(C) = 0$. Der Algorithmus rechnet dann auf dem Rest korrekt weiter. □

2 Flüsse in Netzwerken

Definition 2.1 *Ein* **kombinatorisches Optimierungsproblem** Π *ist charackterisiert durch 4 Komponenten:*

- $\mathcal{D}$*: die Menge der Eingaben (Instanzen)*
- $S(I)$ *für* $I \in \mathcal{D}$*: die Menge der zu* I *zulässigen Lösungen*
- *Die Bewertungsfunktion* $f : S(I) \to \mathbb{N}$ *(*$\mathbb{R}$*)*
- $ziel \in \{\min, \max\}$

Gesucht ist zu $I \in \mathcal{D}$ *eine zulässige Lösung* $\sigma_{OPT} \in S(I)$*, so dass* $f(\sigma_{OPT}) = ziel\{f(\sigma) \mid \sigma \in S(I)\}$.

Definition 2.2 (Flussproblem) *Ein* **Netzwerk** N *ist ein Digraph* $G = (V, E)$ *mit einer* **Kapazitätsfunktion** $c : E \to \mathbb{R}^+$ *und zwei ausgezeichneten Knoten* $s, t \in V$*.* s *ist die* **Quelle** *(source) und* t *die* **Senke** *(sink).* $c(e)$ *ist die Kapazität von* $e \in E$*. Wir schreiben* $N = (G, s, t, c)$.

Bezeichnung: $\overleftarrow{E} = \{i \to j \mid (j \to i) \in E\}$; $\overleftarrow{e} = i \to j$ für $e = (j \to i) \in E$.

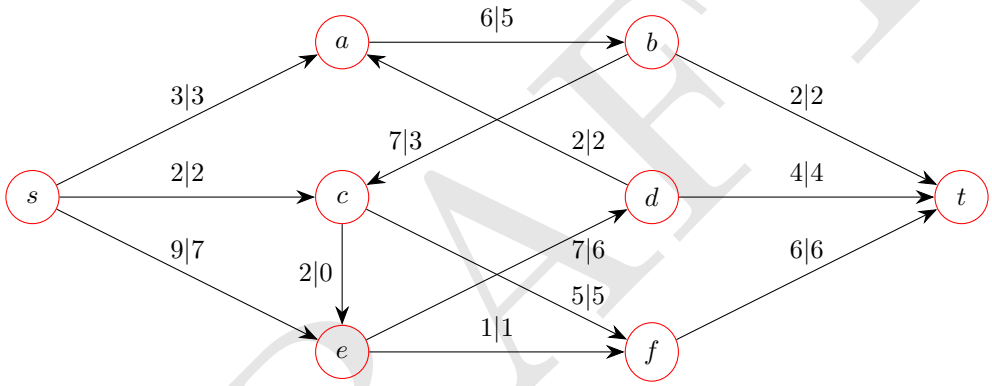


Abbildung 15: Netzwerk $N = (G, s, t, c)$ und Fluss f, die Kapazität und der davon genutzte Fluss (in Klammern) sind an jeder Kante markiert

Definition 2.3 *Sei* N *ein Netzwerk.*

(a) Ein **Fluss** *auf* N *ist eine Funktion*
$f : (E \cup \overleftarrow{E}) \to \mathbb{R}$ *mit*

(i) $0 \le f(e) \le c(e)$ *für* $e \in E$*, und* $f(\overleftarrow{e}) = -f(e)$.

(ii) Konservierungsgesetz:

$$\forall v \in V, s \neq v \neq t : \sum_{u:u\to v} f(u \to v) = \sum_{u:v\to u} f(v \to u)$$

(b) Ist f *ein Fluss auf* N*, so heißt*

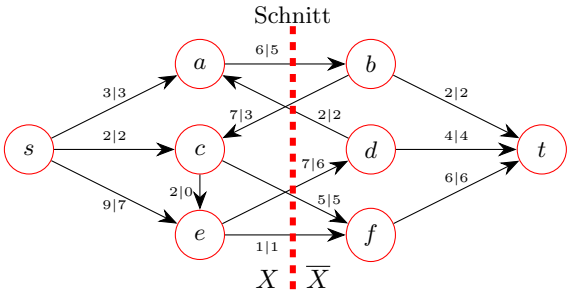
$$|f| := \sum_{v:(s\to v)\in E} f(s \to v) - \sum_{v:(v\to s)\in E} f(v \to s)$$

Wert *des Flusses* f.
Ziel: max.

(c) Ein **s, t-Schnitt** *durch N ist eine Partition von V in X, $\overline{X}$ mit $s \in X$ und $t \in \overline{X}$, $(X \cup \overline{X} = V,\ X \cap \overline{X} = \emptyset)$.*

$$c(X, \overline{X}) = \sum_{u \to v \in E, u \in X, v \in \overline{X}} c(u \to v)$$

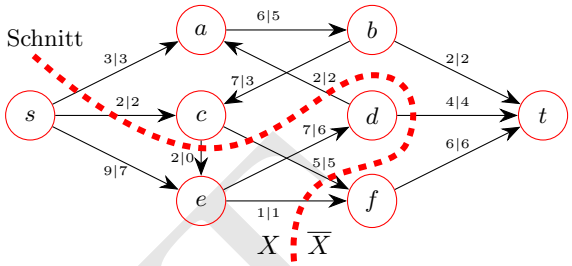
ist die **Kapazität des Schnitts X, $\overline{X}$.**



(a) Schnitt X, $\overline{X}$
$c(X, \overline{X}) = 6 + 7 + 5 + 1 = 19$
$\geq f(X, \overline{X}) = 5 + 6 + 5 + 1 = 12 \geq |f|$

$c(\overline{X}, X) = 7 + 2 = 9 \geq f(\overline{X}, X) = 3 + 2 = 5$

$|f| = f(X, \overline{X}) - f(\overline{X}, X) = 17 - 5 = 12$



(b) Schnitt X, $\overline{X}$
$c(X, \overline{X}) = 3 + 2 + 2 + 4 + 1$
$= f(X, \overline{X}) = 3 + 2 + 2 + 4 + 1 = 12 = |f|$

$c(\overline{X}, X) = 2 \geq f(\overline{X}, X) = 0$

$|f| = f(X, \overline{X}) - f(\overline{X}, X) = 12 - 0 = 12$

Abbildung 16: Schnitte durch einen Graphen G

Der Wert des Flusses kann nicht nur an der Quelle sondern auch analog an der Senke bestimmt werden:

$$|f| = \sum_{v:(v \to t) \in E} f(v \to t) - \sum_{v:(t \to v) \in E} f(t \to v)$$

Aufgrund des Konservierungsgesetzes muss dieser Wert identisch sein.

Lemma 2.1 *Sei N ein Netzwerk. Für jeden s,t-Schnitt X,$\overline{X}$ und jedem Fluss f gilt:*

$$|f| = f(X, \overline{X}) - f(\overline{X}, X) \leq c(X, \overline{X})$$

BEWEIS: $|f| = f(X, \overline{X}) - f(\overline{X}, X) \leq f(X, \overline{X}) \leq c(X, \overline{X})$. □

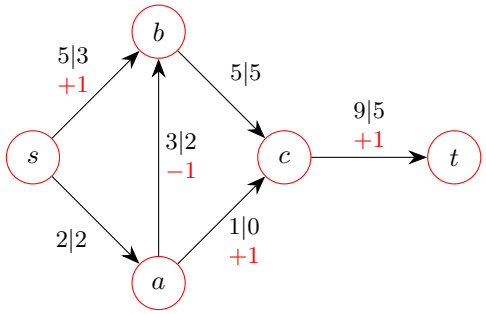
Aufgabe: Bestimme f auf N mit maximalem Wert $|f|$.

Definition 2.4 (Erweiternder Weg) *Sei N ein Netzwerk. Ein Weg von s nach t über Kanten aus $E \cup \overleftarrow{E}$ heißt* **erweiternder Weg** (augmenting path) *bezüglich Fluss f, falls*

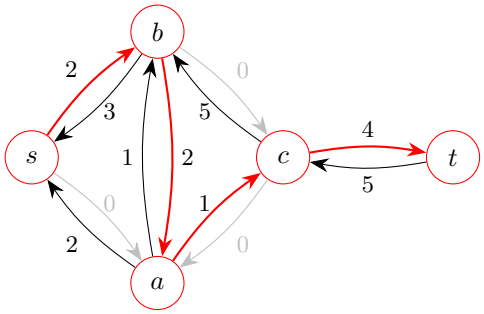
(i) für jede Kante e auf p mit $e \in E$: $f(e) \lneq c(e)$

(ii) für jede Kante $\overleftarrow{e}$ auf p mit zugehörigem $e \in E$: $f(e) \gneq 0$

Verbesserung des Flusses durch erweiternden Weg:
Ist p ein erweiternder Weg in N,f so setze
$\varepsilon_1 = \min\{c(e) - f(e) \mid e \text{ auf } p\}$,
$\varepsilon_2 = \min\{f(e) \mid \overleftarrow{e} \text{ auf } p\}$,
$\varepsilon = \min\{\varepsilon_1, \varepsilon_2\}$. Dann lässt sich f zu einem Fluss f' erweitern mit $|f'| = |f| + \varepsilon$.
Auf den „Vorwärtskanten“ e: $f(e) := f(e) + \varepsilon$.
Auf den „Rückwärtskanten“ $\overleftarrow{e}$: $f(e) := f(e) - \varepsilon$.

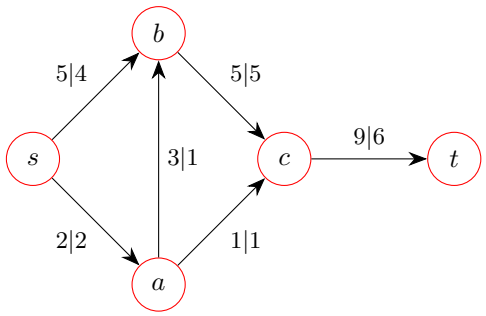


(a) Netzwerk mit einem vorhandenem Fluss und Erweiterungsmöglichkeit

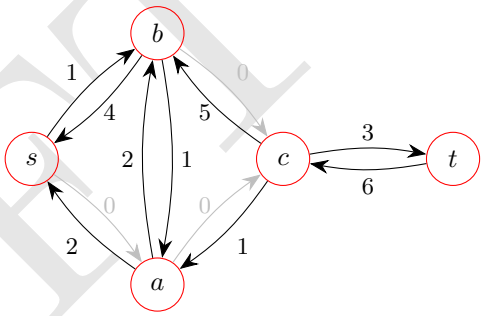


(b) Residualgraph mit erweiterndem Weg

Abbildung 17: Netzwerk mit Residualgraph



(a) Netzwerk mit maximalem Fluss



(b) Residualgraph (Es ist kein erweiternder Weg vorhanden)

Abbildung 18: erweitertes Netzwerk mit Residualgraph

Beispiel 2.1 (Unendliche Laufzeit mit reelen Kantenkapazitäten)
Sei $\lambda = (\sqrt{5}-1)/2 \approx 0.618034$. *Dann gilt*

- $\lambda^2 + \lambda = ((\sqrt{5}-1)/2)^2 + (\sqrt{5}-1)/2 = 5/4 - 2\cdot\sqrt{5}/4 + 1/4 + \sqrt{5}/2 - 1/2 = 1$,
  also $\lambda^2 + \lambda = 1 \Leftrightarrow \lambda^{i+2} = \lambda^i - \lambda^{i+1}$
- $\lambda + 2 = ((\lambda+2)(1-\lambda))/(1-\lambda) = (2-(\lambda^2+\lambda))/(1-\lambda) = 1/(1-\lambda)$,
  also $\lambda + 2 = 1/(1-\lambda)$
- $1/(1-\lambda) = \lim_{n\to\infty}(1-\lambda^{n+1})/(1-\lambda) = \lim_{n\to\infty}\sum_{i=0}^{n}\lambda^i = \sum_{i=0}^{\infty}\lambda^i$,
  also $\lambda + 2 = 1/(1-\lambda) = \sum_{i=0}^{\infty}\lambda^i$

Nun wird das Netzwerk $N = (G, s, t, c)$ *definiert. In Abbildung 19 ist der Graph, die Quelle* s *und die Senke* t *wie üblich dargestellt. Die Kantenkapazitäten* c *sind jeweils auf den Kanten vermerkt.*
Ein maximal großer Fluss kann dabei erreicht werden indem f *folgendermaßen gesetzt wird:*
$f((s\to b)) = f((b\to f)) = f((f\to t)) = 1$, $f((s\to c)) = f((c\to d)) = f((d\to t)) = \lambda$,
$f((s\to a)) = f((a\to e)) = f((e\to t)) = \lambda^2$
Es gilt dann $|f| = 1+\lambda+\lambda^2 = 2$. *Dieser Fluss ist maximal, weil der Schnitt* $X = \{s,a,b,c\}$, $\overline{X} = \{d,e,f,t\}$ *Kapazität* $1+\lambda+\lambda^2 = 2$ *aufweist, und der Fluss nicht größer sein kann, als ein* s,t*-Schnitt.*
Insbesondere gibt es bei diesem Netzwerk eine unendliche Folge von erweiternden Wegen. Der erste erweiternde Weg in dieser Folge ist der Weg $s\to b\to f\to t$, *welcher in Abbildung 20 dargestellt ist. Die maximal mögliche Erhöhung des Flusses über diesen Weg ist* 1.

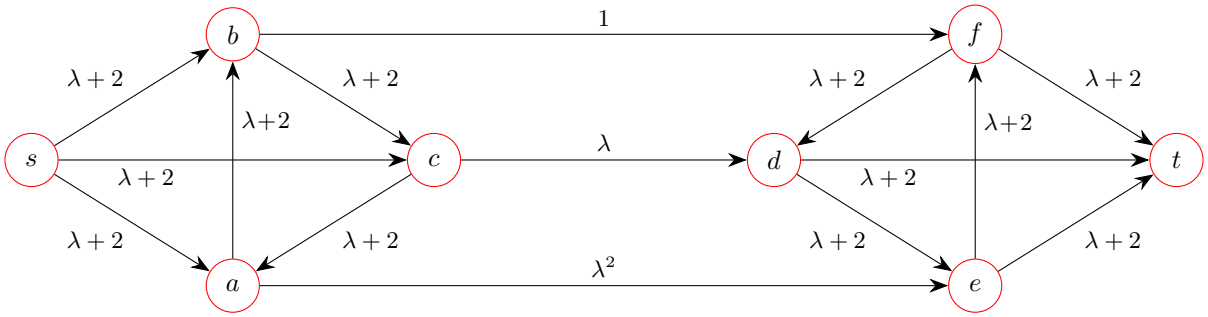


Abbildung 19: Netzwerk $N = (G, s, t, c)$

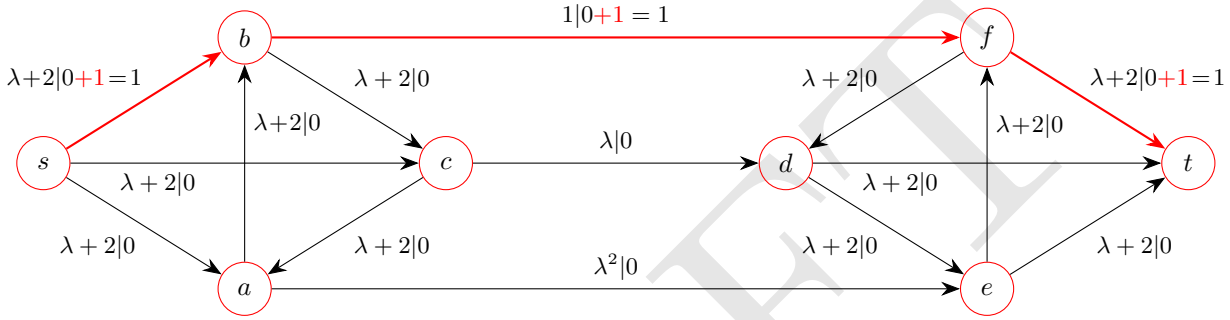


Abbildung 20: Netzwerk $N = (G, s, t, c)$ mit erstem erweiternden Weg

Im folgenden werden wir (bis auf die 3 mittleren Kanten) die Kanten nur in Fließrichtung nutzen. Die Kapazität der Kanten im linken und rechten Bereich sind größer als 2 und können daher als unendlich angenommen werden, da maximal ein Fluss mit Wert 2 erreicht werden kann. Die Situation aus Abbildung 20 kann damit vereinfacht wie in Abbildung 21 mit $i = 1$ dargestellt werden.

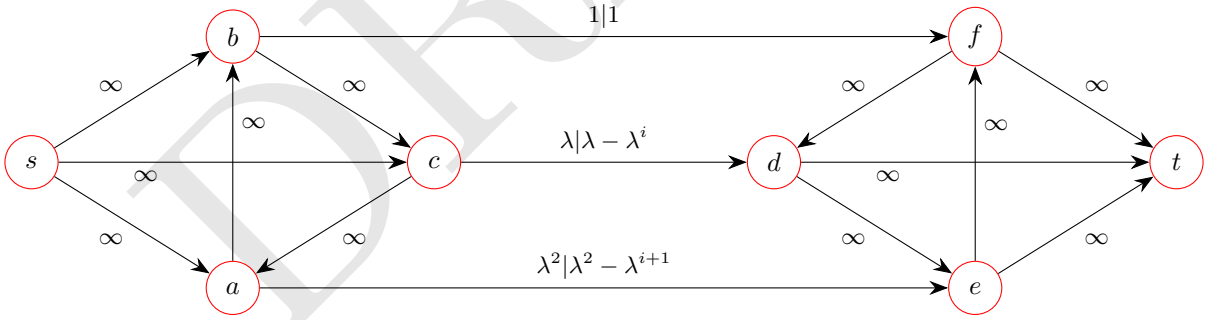


Abbildung 21: Netzwerk $N = (G, s, t, c)$, vereinfachte Darstellung des Flusses

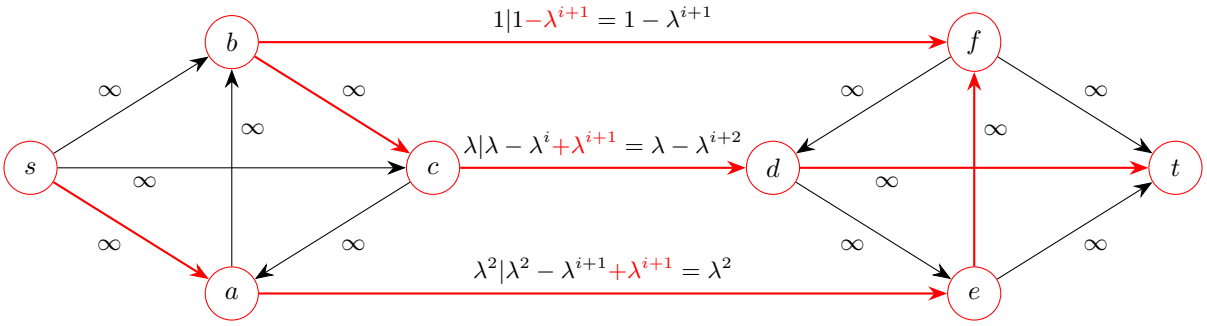


Abbildung 22: Netzwerk $N = (G, s, t, c)$, mit erweiterndem Weg (1)

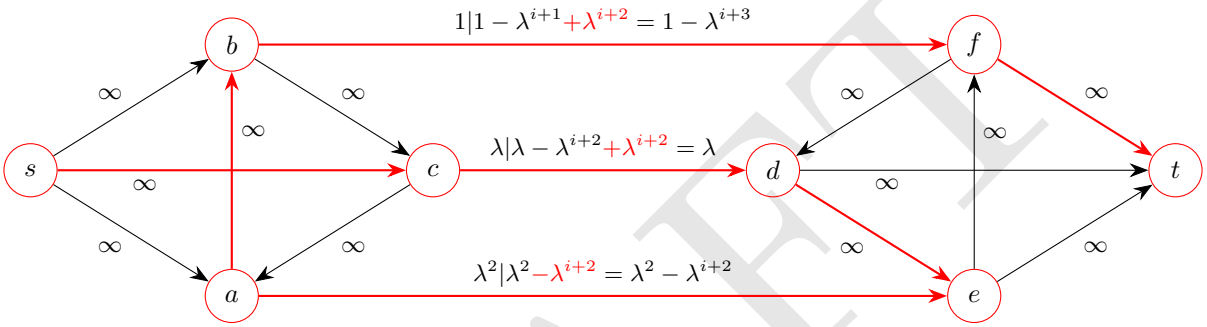


Abbildung 23: Netzwerk $N = (G, s, t, c)$, mit erweiterndem Weg (2)

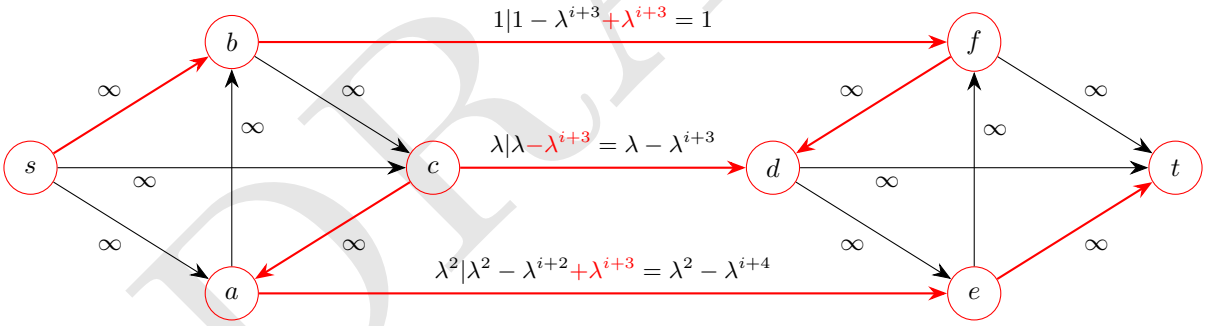


Abbildung 24: Netzwerk $N = (G, s, t, c)$, mit erweiterndem Weg (3)

Mit der in den Abbildungen 22, 23 und 24 dargestellten Folge von erweiternden Wegen landen wir wieder in der Situation, die in Abbildung 21 visualisiert ist, jedoch mit einem um 3 erhöhtem Wert i. Entsprechend kann man diese Folge von erweiternden Wegen unendlich oft wiederholen.
Nach der Anwendung des ersten erweiternden Weges hat der Fluss den Wert 1. Nach der zweiten Anwendung erhöht sich der Wert des Flusses um λ^2. Bei jeder weiteren Anwendung eines erweiternden Weges nimmt die Erhöhung des Flusswertes um den Faktor λ ab. Die Erhöhung bleibt jedoch immer echt positiv. Da nach jeder Iteration der Wert des Flusses echt größer wird, kann der Fluss nie den Wert 2 annehmen, da er ab der Iteration, in der der Fluss den Wert 2 annimmt, nicht mehr zunehmen könnte. Wählt man also eine Kapazitätsfunktion $c : E \to \mathbb{R}$ so kann es passieren, dass das iterative Anwenden von erweiternden Wegen in endlicher Zeit nicht zu einem maximalen Fluss führt.

Beispiel 2.2 (Exponentielle Laufzeit mit ganzzahligen Kantenkapazitäten)
Sei Das Netzwerk $N = (G, s, t, c)$ durch Abbildung 25 definiert, wobei die Kantenkapazitäten an den Kanten vermerkt sind. Hierbei ist anzunehmen, dass k eine große natürliche Zahl ist.

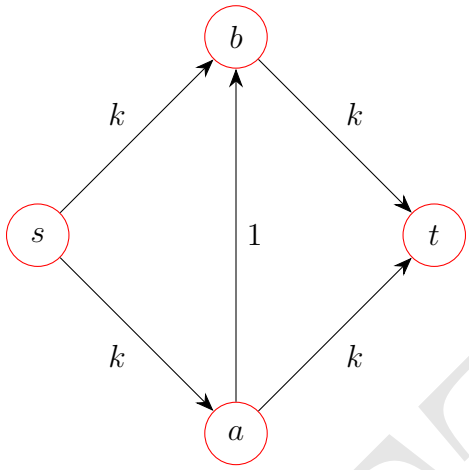
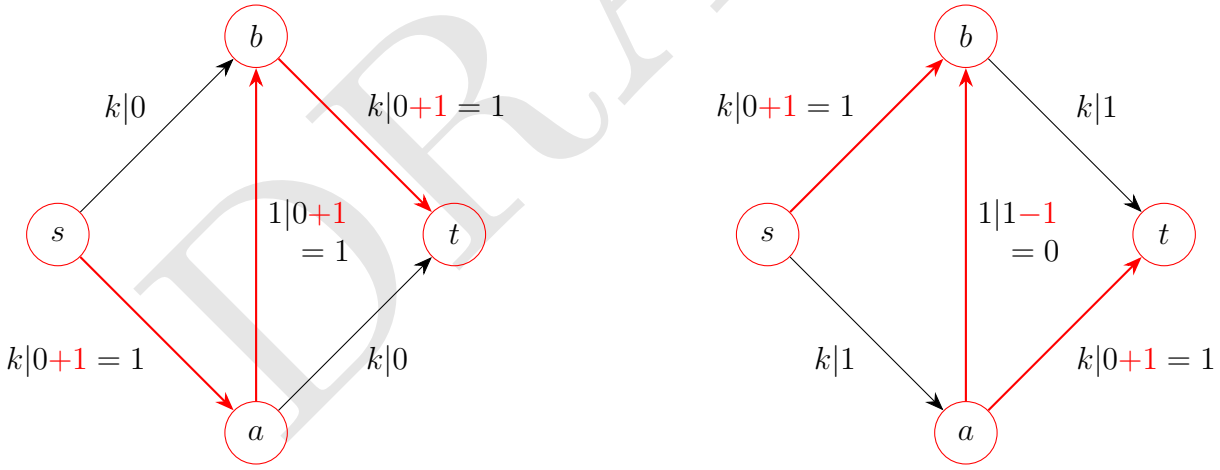


Abbildung 25: Netzwerk $N = (G, s, t, c)$

Der maximale Fluss in diesem Netzwerk hat Wert $(2k)$ für $f((s \to a)) = f((s \to b)) = f((a \to t)) = f((b \to t)) = k$ und $f((a \to b)) = 0$. Ein minimaler s,t-Schnitt mit Kapazität $(2k)$, welcher die Maximalität des Flusses zeigt, ist beispielsweise $X = \{s\}$, $\overline{X} = \{a, b, t\}$. Wenn man mit einem Fluss mit Wert 0 beginnt können jedoch, wie in Abbildung 26, ungünstige erweiternde Wege gewählt werden.



(a) erweiternder Weg (1)

(b) erweiternder Weg (2)

Abbildung 26: Erweiternde Wege im Netzwerk $N = (G, s, t, c)$

Wenn abwechselnd die erweiternden Wege $s \to a \to b \to t$ und $s \to b \to a \to t$ (wie in Abbildung 26) gewählt werden, so kann der Fluss jeweils nur um 1 erhöht werden. Insgesamt müssen also $(2k)$ viele erweiternde Wege angewendet werden. Da die Kapazitäten mit logarithmisch vielen Bits dargestellt werden können, so benötigt das iterative anwenden von erweiternden Wegen in diesem Netzwerk (bei entsprechender Wahl der erweiternden Wege) exponentielle Laufzeit.

Satz 2.1 (Ford/Fulkerson 1956/1957) *Sei N ein Netzwerk.*

(1) (Satz über erweiternde Wege)
f ist ein maximaler Fluss auf N $\Leftrightarrow$ In N gibt es keinen erweiternden Weg bezüglich f.

(2) (Max Flow Min Cut Theorem)
*Der maximale Wert $|f|$ eines Flusses auf N ist **gleich** der minimalen Kapazität eines s,t-Schnittes.*

BEWEIS:

(1) „$\Rightarrow$": Falls N bezüglich f einen erweiternden Weg enthält, kann f gemäß der Diskussion (Verbesserung des Flusses durch erweiternden Weg) echt verbessert werden, also ist f nicht maximal.$\surd$
„$\Leftarrow$": Angenommen, es gibt keinen erweiternden Weg bezüglich f. Dann endet jeder Versuch, einen erweiternden Weg von s nach t zu finden, jeweils an Knoten v mit:

- $f(e) = c(e)$ falls $q(e) = v$ $(e = (v \to .))$: Alle Kanten, die in v starten sind voll ausgenutzt.
- $f(e) = 0$ falls $z(e) = v$ $(e = (. \to .))$: Alle Kanten, die in v enden sind ungenutzt.

Sei $X = \{s\} \cup \{v \in V \mid$es gibt erweiternden Weg von s nach v in N bezüglich $f\}$.
Bemerkung: $t \notin X$.

$$\Rightarrow |f| = f(X, \overline{X}) = \sum_{v \in X} \sum_{\substack{u:(v \to u) \in E, \\ u \in \overline{X}}} f(v \to u) = \sum_{v \in X} \sum_{\substack{u:(v \to u) \in E, \\ u \in \overline{X}}} c(v \to u) = c(X, \overline{X})$$

$\Rightarrow$ wegen Lemma 2.1: f ist maximaler Fluss.

(2) Sei f ein maximaler Fluss von s nach t.
Nach (1) gibt es einen s,t-Schnitt X, $\overline{X}$ mit $c(X, \overline{X}) = f(X, \overline{X}) = |f|$. Für jeden anderen s,t-Schnitt Y, $\overline{Y}$ gilt mit Lemma 2.1: $|f| \leq c(Y, \overline{Y}) \Rightarrow X, \overline{X}$ ist ein s,t-Schnitt mit minimaler Kapazität.

□

Korollar 2.1 (Integrality Theorem) *Sei N ein Netzwerk mit $c(e) \in \mathbb{N}_0$ für alle $e \in E$. Dann* gibt es *einen maximalen Fluss f auf N mit $f(e) \in \mathbb{N}_0$ für alle $e \in E$.*

BEWEIS: Starte mit dem leeren Fluss: $f(e) = 0$ für alle $e \in E$. Jede Erweiterung von f mittels erweiternder Wege erhält die Ganzzahligkeit auf den Kanten. Durch jeden erweiternden Weg kann der Fluss um mindestens 1 verbessert werden. Nach spätestens f_{max} Iterationen ist der Fluss maximal, wobei f_{max} der Wert eines maximalen Flusses ist. □

Algorithmus von Dinic: (Laufzeit $O(|V|^2|E|)$)

Definition 2.5 *Sei $N = (G = (V, E), s, t, c)$ ein Netzwerk und f ein Fluss auf N. Eine Kante $e \in E \cup \overleftarrow{E}$ hat **Restkapazität** $\tilde{\mathbf{c}}(\mathbf{e})$ in N bezüglich f, wobei $\tilde{\mathbf{c}}(\mathbf{e}) = \begin{cases} c(e) - f(e) & \text{falls } e \in E \\ f(e) & \text{falls } e \in \overleftarrow{E} \end{cases}$ gilt.*

Die Kanten $e \in E \cup \overleftarrow{E}$ mit $\tilde{c}(e) > 0$ heißen **nützliche Kanten** *von N bezüglich f. Erweiternde Wege enthalten ausschließlich nützliche Kanten.*
f heißt **Sperrfluss** (blocking flow)*, falls kein erweiternder Weg von N bezüglich f existiert, der nur Kanten aus E enthält (also keine Kanten aus $\overleftarrow{E}$).*

Sei $\tilde{\delta}(v)$ die Länge (=Anzahl der Kanten) eines kürzesten Weges von s zu v über nützliche Kanten von N bezüglich f, für alle $v \in V$. Falls kein solcher Weg zu v existiert, ist $\tilde{\delta}(v) = \infty$.
Das Netzwerk $\tilde{N}(f) = ((\tilde{V}, \tilde{E}), s, t, \tilde{c})$, wobei $(\tilde{V}, \tilde{E})$ ein Digraph ist mit

- $\tilde{V} = \{v \in V | v$ ist auf einem kürzestem erweiterndem Weg von s nach t, und $\tilde{\delta}(v) \leq \tilde{\delta}(t)\}$
- $\tilde{E} = \{e \in E \cup \overleftarrow{E} \mid e = (u \to v)$ ist nützliche Kante mit $u, v \in \tilde{V}$ und $\tilde{\delta}(v) = \tilde{\delta}(u) + 1\}$.

ist ein ***geschichtetes Netzwerk*** *(Layered Network).*
Bemerkung:

(a) $(\tilde{V}, \tilde{E})$ ist ein DAG (directed acyclic graph).

(b) f ist maximaler Fluss $\Leftrightarrow \tilde{V} = \emptyset$

Lemma 2.2 *$\tilde{\delta}$ und $\tilde{N}(f)$ können mittels Breitensuche in Zeit $O(|E|)$ berechnet werden.* □

Lemma 2.3 *Sei $\tilde{N}(f)$ das zu N, f gehörende geschichtete Netzwerk, und sei $\tilde{f}$ ein Sperrfluss auf $\tilde{N}(f)$. Dann ist f' mit*

$$f'(e) = \begin{cases} f(e) + \tilde{f}(e) & \text{für } e \in E \\ f(e) - \tilde{f}(e) & \text{für } e \text{ Rückkante} \\ f(e) & \text{für restliche Kanten} \end{cases}$$

ein Fluss auf N mit $|f'| = |f| + |\tilde{f}|$.

Algorithm 8: Dinic - Algorithmisches Schema zur Berechnung eines maximalen Flusses auf N

```
   Input  : Netzwerk N = (G, s, t, c)
   Output: maximaler Fluss f auf N
1  Setze den Startfluss f(e) = 0 für alle e ∈ E;
2  repeat
3  |  Berechne geschichtetes Netzwerk Ñ(f);
4  |  if Ṽ ≠ ∅ then
5  |  |  Berechne Sperrfluss f̃ auf Ñ(f);
6  |  |  Berechne aus f und f̃ verbesserten Fluss f auf N;
7  |  end
8  until Ṽ = ∅;
9  return f;
```

Satz 2.2 *Die repeat-Schleife wird höchstens $|V| - 1$ mal durchlaufen.*

BEWEIS: Zeige: die Abstandsfunktion $\tilde{\delta}(t)$ in $\tilde{N}(f)$ ist von repeat-Iteration zu repeat-Iteration streng monoton steigend.
Sei f aktueller Fluss, $\tilde{N}(f)$, $\tilde{\delta}$ in $\tilde{N}(f)$, und seien f', $\tilde{N}(f')$, $\tilde{\delta}'$, die Größen in der darauffolgenden Iteration.
Sei $p = (s = u_0 \to u_1 \to \ldots \to u_k = t)$ Weg in $\tilde{N}(f')$ mit $\tilde{\delta}'(u_i) = i$.
Zeige durch Induktion: $\forall i : \tilde{\delta}'(u_i) \geq \tilde{\delta}(u_i)$
$i = 0 : \tilde{\delta}(s) = 0 = \tilde{\delta}'(s) \surd$
$i \to i + 1 :$ (Es gilt Induktionsvoraussetzung - IV: $\tilde{\delta}'(u_i) \geq \tilde{\delta}(u_i)$)
Annahme: $\tilde{\delta}'(u_{i+1}) < \tilde{\delta}(u_{i+1})$
$\Rightarrow$ es existiert nützliche Kante $u_i \to u_{i+1}$ von $\tilde{N}(f')$ bezüglich dem leeren Fluss.

Fall 1: $u_i \to u_{i+1}$ ist nützliche Kante von $\tilde{N}(f)$ bezüglich dem leeren Fluss
$\Rightarrow \tilde{\delta}'(u_{i+1}) = 1 + \tilde{\delta}'(u_i) \overset{IV}{\geq} 1 + \tilde{\delta}(u_i) \geq \tilde{\delta}(u_{i+1}) \lightning$

Fall 2: $u_i \to u_{i+1}$ ist keine nützliche Kante von $\tilde{N}(f)$ bezüglich dem leeren Fluss
$\Rightarrow$ die Kante $u_{i+1} \to u_i$ muss im vorherigen Schritt aktualisiert worden sein
$\Rightarrow$ die Kante $u_{i+1} \to u_i$ lag in $\tilde{N}(f)$ auf kürzestem Weg von s nach t
$\Rightarrow \tilde{\delta}(u_{i+1}) + 1 = \tilde{\delta}(u_i)$
Damit gilt insgesamt $\tilde{\delta}'(u_{i+1}) = 1 + \tilde{\delta}'(u_i) \overset{IV}{\geq} 1 + \tilde{\delta}(u_i) = 2 + \tilde{\delta}(u_{i+1}) > \tilde{\delta}(u_{i+1})$↯

Damit haben wir $\tilde{\delta}'(t) \geq \tilde{\delta}(t)$.
Noch zu zeigen: $\tilde{\delta}'(t) > \tilde{\delta}(t)$.
Annahme: $\tilde{\delta}'(t) = \tilde{\delta}(t)$.
$\Rightarrow \tilde{\delta}'(u_i) = \tilde{\delta}(u_i)$ für alle i.
$\Rightarrow p$ ist Weg in $\tilde{N}(f)$. Dann kann der berechnete Fluss kein Sperrfluss gewesen sein. □

Dinic-Realisierung von „berechne Sperrfluss $\tilde{f}$ auf $\tilde{N}(f)$“:
Sei $\tilde{N}(f)$ das zu N, f gehörende geschichtete Netzwerk mit Schichten $V_0, V_1, \ldots, V_k$, $V_0 = \{s\}$, $V_k = \{t\}$, $V_i = \{v \in \tilde{V} \mid \tilde{\delta}(v) = i\}, 1 \leq i < k$.

Algorithm 9: SPERRFLUSS

Input : geschichtetes Netzwerk $\tilde{N}(f)$
Output: Fluss auf $\tilde{N}(f)$: $\tilde{f}$

```
1  for all e ∈ Ẽ do
2  |  f̃(e) := 0;
3  end
4  repeat
5  |  v := t;
6  |  a := ∞;
   |  /* Berechne erweiternden Weg                                          */
7  |  for i := k down to 1 do
8  |  |  Wähle Kante e_i = u → v;
9  |  |  a := min{c̃(e_i), a};
10 |  |  v := u;
11 |  end
   |  /* Erweitere f̃ um die auf diesem Weg gefundene Möglichkeit a          */
   |  /* Aktualisiere Kapazitäten                                           */
12 |  for i := 1 to k do
13 |  |  f̃(e_i) := f̃(e_i) + a;
14 |  |  c̃(e_i) := c̃(e_i) − a;
15 |  |  if c̃(e_i) = 0 then
16 |  |  |  Entferne e_i aus Ẽ;
17 |  |  end
18 |  end
   |  /* Entferne unnütze Knoten und zugehörige Kanten                      */
19 |  for i := 1 to k do
20 |  |  for all v ∈ V_i do
21 |  |  |  if d⁻_(Ṽ,Ẽ)(v) = 0 then
22 |  |  |  |  Entferne alle seine inzidenten Kanten e ∈ Ẽ mit q(e) = v;
23 |  |  |  end
24 |  |  end
25 |  end
26 until t ∉ V_k;
27 return f̃;
```

Satz 2.3 SPERRFLUSS *berechnet auf* $\tilde{N}(f)$ *in* $O(|V| \cdot |E|)$ *Schritten einen Sperrfluss.*

BEWEIS: Korrektheit:

- in erster for-Schleife wird rückwärts von t aus einen Weg von s zu t berechnet. a enthält den maximal möglichen Wert, um den Fluss über diesen erweiternden Weg verbessert werden kann.
- In zweiter for-Schleife wird $\tilde{f}$ um a erhöht und die Kapazität der Kanten auf diesem Weg neu berechnet. Dadurch erhält mindestens eine Kante die Restkapazität 0 und wird entfernt.
- In letzter doppelter for-Schleife werden alle Knoten außer s aus $\tilde{V}$ entfernt, bei denen die Suche nach einem erweiterndem Weg von t aus vorzeitig stoppen würde. Kanten, die aus diesem Knoten herausgehen, werden ebenfalls eliminiert. Damit wird die repeat-Schleife mit den gleichen Bedingungen wie zu Beginn wiederholt.

Zum Schluss gibt es keine erweiternden Wege über Vorwärtskanten des Netzwerks $\tilde{N}(f)$ bezüglich $\tilde{f}$, weil nur vollständig ausgeschöpfte Kanten oder Kanten, welche in erweiternden Wegen über $\tilde{E}$ nicht mehr vorkommen können, entfernt werden. Also ist $\tilde{f}$ ein Sperrfluss.
Laufzeit: Die repeat-Schleife wird $O(|E|)$ mal durchlaufen. Eine repeat-Iteration dauert $O(|V|)$ Schritte. □

Korollar 2.2 *Das algorithmische Schema Algorithmus 8 mit Algorithmus 9* SPERRFLUSS *zur Berechnung des Sperrflusses im geschichteten Netzwerk berechnet in Zeit* $O(|V|^2 \cdot |E|)$ *einen maximalen Fluss auf* N.

3 Parametrisierte Komplexität und Vertex Cover

Definition 3.1 *Sei $G = (V, E)$ ein ungerichteter Graph. Eine* **Knotenüberdeckung** (Vertex Cover) *ist eine Knotenmenge C, $C \subseteq V$, sodass für jede Kante $\{u, v\} \in E : \{u, v\} \cap C \neq \emptyset$.*
Das **Entscheidungsproblem $\mathbf{VC_{ent}}$** *ist*

$$VC_{ent} = \{\langle G, k\rangle \mid G \text{ hat Vertex Cover } C \text{ mit } |C| = k\}.$$

Das **Optimierungsproblem $\mathbf{VC_{OPT}}$** *bestimmt zur Eingabe G eine Knotenüberdeckung kleinster Kardinalität.*

Bemerkungen:

(a) Wenn VC_{OPT} in Zeit $p(n)$ gelöst werden kann, kann auch VC_{ent} in Zeit $O(p(n))$ entschieden werden.

(b) Wenn man VC_{ent} in Zeit $p(n)$ entscheiden kann mit $n = |V| + |E| + \log(k)$, dann kann VC_{OPT} in Zeit $O(|V| + |E| + |V| \cdot p(n))$ gelöst werden.

Fakt: VC_{ent} ist NP-vollständig.

O^*-Notation: Gegeben zwei Funktionen f und g. Wir schreiben $f(n) = O^*(g(n))$, wenn es ein festes Polynom $poly(n)$ gibt, sodass $f(n) = O(g(n) \cdot poly(n))$.
Beispiel: $n^2 \cdot 2^n = O^*(2^n)$

3.1 Ein exakter Algorithmus für Vertex Cover

Für jeden Knoten v gilt: Entweder ist v im Vertex Cover oder **alle** seine Nachbarn.

Algorithm 10: MVC

Input : ungerichteter Graph $G = (V, E)$
Output: minimales Vertex Cover VC_{OPT}

```
1  Setze VC_OPT := V;
2  Procedure MVC(VC: subset of V, G: Graph)
3      if |E(G)| = 0 then
4          if |VC| < |VC_OPT| then
5              VC_OPT:=VC;
6          end
7      else
8          while es gibt Kante (u − v) ∈ E(G) mit deg_G(u) = 1 do
9              VC:=VC ∪ {v};
10             G:=G \ {v};
11         end
           /* Es gibt keine Knoten mit Grad 1 mehr                          */
12         if für alle Knoten v ∈ V(G) gilt: deg_G(v) ∈ {0, 2} then
               /* Auf Kreisen kann leicht ein Vertex Cover berechnet werden. */
13             Bestimme optimales Cover für G und füge es zu VC hinzu;
14             E(G) := ∅;
15             MVC(VC, G);
16         else
               /* Es gibt mindestens einen Knoten v mit deg_G(v) ≥ 3         */
17             Wähle v ∈ V(G) mit deg_G(v) = k ≥ 3;
18             Bestimme die Nachbarn u_1, ..., u_k von v;
19             MVC(VC ∪ {v}, G \ {v});
20             MVC(VC ∪ {u_1, ..., u_k}, G \ {u_1, ..., u_k});
21         end
22     end
23 return VC_OPT;
```

Satz 3.1 *MVC löst VC_{OPT} in Zeit $O^*(1.3803^n)$.*

BEWEIS: MVC löst VC_{OPT} wegen „Correct by Construction“.
Sei $t(n) \leq c$ für $n \leq 4$.
$t(n) \leq t(n-1) + t(n-4) + poly(n)$.
Löse nun die zugehörige lineare Rekurrenz (beziehungsweise lineare Differenzengleichung):
$\lambda^4 = \lambda^3 + 1$ hat größte Nullstelle $\lambda \approx 1.3803$
$\Rightarrow t(n) = O^*(1.3803^n)$. □

3.2 Parametrisierte Komplexität

Bislang: Worst-Case-Laufzeiten „in der Eingabelänge“.
Jetzt: Wir wollen aus der Eingabe Parameter extrahieren, die nicht von der Eingabelänge abhängen. Die Laufzeit wird nun in Abhängigkeit von den Parametern angegeben.

Definition 3.2 *Es sei L ein Entscheidungsproblem (L ist Menge und wir fragen „$x \in L$?“). Eine beliebige in Polynomzeit berechenbare Funktion $Par : L \to \mathbb{N}$ nennt man* **Parametrisierung** *von L. Ein Algorithmus A ist ein* **Par-parametrisierter Polynomzeit-Algorithmus** *für L, falls gilt:*

(i) A löst das Entscheidungsproblem L.

(ii) Es gibt eine Funktion $g : \mathbb{N} \to \mathbb{N}$ und ein Polynom $p : \mathbb{N} \to \mathbb{N}$, sodass die Laufzeit von A für jede Eingabe I in $O(g(Par(I)) \cdot p(|I|))$ ist.

Wenn es für L einen solchen Par-parametrisierten Polynomzeit-Algorithmus gibt, nennt man L **fixed-parameter tractable in Bezug auf Par**.

Beispiel:

- $Par(I) = |I|$. $g(n) = 2^n$. Bezüglich dieser Parametrisierung ist Vertex Cover fixed-parameter-tractable.
- Rucksackproblem: Es gibt Algorithmus mit Laufzeit $O(n^2 \cdot p_{max})$, wobei n die Anzahl der Waren und p_{max} der größte vorkommende Preis ist.
  $Par(I)$ =Anzahl Bits des größten vorkommenden Preises, $g(n) = 2^n$.

Problemkern-Methode

- Reduziere die Eingabe I auf eine „gleichwertige“ Instanz I' (Problemkern), wobei die Größe von I' nur von Parameter k abhängt (Kernelization).
- Löse I' und übertrage die Antwort auf I.

Lemma 3.1 *Wenn $G = (V, E)$ eine Knotenüberdeckung C mit $|C| = k$ hat, muss C alle Knoten aus V enthalten, die Grad größer als k haben.*

BEWEIS: Wäre v mit $deg_G(v) > k$ nicht in C, so wären alle $> k$ vielen Nachbarn in C ↯ □

Lemma 3.2 *In einem Graphen $G = (V, E)$ (ohne isolierte Knoten) gilt:*
Wenn alle Knoten in G Grad $\leq k$ haben und dieser Graph eine Knotenüberdeckung der Größe m besitzt, dann hat G höchstens $m \cdot k$ Kanten.

BEWEIS: Eine Knotenüberdeckung C berührt jede Kante, das heißt $|C| \cdot k$ ist obere Schranke für die Anzahl der Kanten. □

Algorithm 11: DivideAndConquerVC

Input : ungerichteter Graph $G = (V, E)$, k
Output: Antwort auf Frage: $|VC_{OPT}| \leq k$?

```
1 if |E| = 0 then
2 |   return True;
3 else if k = 0 then
4 |   return False;
5 end
  /* E ≠ ∅ und k > 0:                                          */
6 Wähle eine beliebige Kante {u, v} ∈ E;
7 G_1 := G|_{V\{u}};
8 G_2 := G|_{V\{v}};
9 return DivideAndConquerVC(G_1, k − 1) or DivideAndConquerVC(G_2, k − 1);
```

Der Algorithmus 11 (DivideAndConquerVC) beantwortet die Frage „$|VC_{OPT}| \leq k$?“ in Zeit $O(|V|^2 \cdot 2^k)$ beziehungsweise in Zeit $O^*(2^k)$.

Algorithm 12: Buss & Goldsmith

Input : ungerichteter Graph $G = (V, E)$, k
Output: Antwort auf Frage: $|VC_{OPT}| \leq k$?

```
   /* 1. Kernelization                                                    */
1  H:={v | deg_G(v) > k};
2  if |H| > k then
   |  /* Kein Vertex Cover der Größe k möglich                           */
3  |  return FALSE;
4  end
5  G':=G \ H;
6  Entferne alle isolierten Knoten aus G';
7  m:=k - |H|;
   /* 2. Weitere Kernelization                                           */
8  if G' enthält mehr als m · k Kanten then
   |  /* Kein Vertex Cover der Größe k möglich                           */
9  |  return FALSE;
10 end
   /* G' hat Grad ≤ k, |E(G')| ≤ k · m ≤ k^2, |V(G')| ≤ 2 · k^2          */
   /* 3. Löse das Problem auf dem viel kleineren Graphen                 */
11 Löse VC_ent auf G' durch Ausprobieren mit Frage ⟨G', m⟩ ∈ VC;
   /* oder benutze Algorithmus 11 DivideAndConquerVC                     */
12 return Ergebnis des Ausprobierens;
```

Satz 3.2 *Der Algorithmus von Buss & Goldsmith entscheidet VC_{ent} auf Graphen mit n Knoten in Zeit $O(k \cdot n + 2^k \cdot k^{2k+2})$ beziehungsweise (mit Algorithmus 11) in Zeit $O(k \cdot n + 2^k \cdot k^4)$.*

BEWEIS: Algorithmus ist Correct by Construction $\surd$
Laufzeit:
Kernelisation (1.+2.) benötigt Zeit $O(k \cdot n)$
Ausprobieren: Die Anzahl der m-elementigen Teilmengen von $V(G')$ ist $\leq \binom{2km}{m} \leq (2km)^m \leq (2k^{2k})$.
3. Schritt benötigt Zeit $O((2km)^m \cdot km) \leq O((2^k \cdot k^{2k} \cdot k^2) = O(2^k \cdot k^{2k+2})$
Mit Algorithmus 11 benötigt der 3. Schritt Zeit $O(2^k \cdot k^4)$. □

Lemma 3.3 (Nach Schöning und Torán)
(aus dem Anhang des Buches „Das Erfüllbarkeitsproblem SAT - Algorithmen und Analysen“)
Für $\delta \in]0, \frac{1}{2}]$ und $\delta \cdot n \in \mathbb{N}$ gelten die folgenden Ungleichungen:

$$\frac{1}{n+1}\left(\left(\frac{1}{\delta}\right)^{\delta} \cdot \left(\frac{1}{1-\delta}\right)^{1-\delta}\right)^n \leq \sum_{i=0}^{\delta n} \binom{n}{i} \leq \left(\left(\frac{1}{\delta}\right)^{\delta} \cdot \left(\frac{1}{1-\delta}\right)^{1-\delta}\right)^n$$

BEWEIS: Bestimmung einer oberen Schranke für $\sum_{i=0}^{\delta n} \binom{n}{i}$:

$$
\begin{aligned}
1 = (\delta + (1-\delta))^n & & = \sum_{i=0}^{n} \binom{n}{i} \cdot \delta^i \cdot (1-\delta)^{n-i} \\
= (1-\delta)^n \sum_{i=0}^{n} \binom{n}{i} \cdot \left(\frac{\delta}{1-\delta}\right)^i & & \geq (1-\delta)^n \cdot \sum_{i=0}^{\delta n} \binom{n}{i} \cdot \underbrace{\left(\frac{\delta}{1-\delta}\right)^i}_{\leq 1} \\
\geq (1-\delta)^n \cdot \sum_{i=0}^{\delta n} \binom{n}{i} \cdot \left(\frac{\delta}{1-\delta}\right)^{\delta n} & & = \delta^{\delta n} \cdot (1-\delta)^{(1-\delta)n} \sum_{i=0}^{\delta n} \binom{n}{i}
\end{aligned}
$$

Also insgesamt

$$1 \geq \delta^{\delta n} \cdot (1-\delta)^{(1-\delta)n} \sum_{i=0}^{\delta n} \binom{n}{i}$$

und somit

$$\sum_{i=0}^{\delta n} \binom{n}{i} \leq \left(\left(\frac{1}{\delta}\right)^{\delta} \cdot \left(\frac{1}{1-\delta}\right)^{1-\delta}\right)^n$$

Bestimmung einer unteren Schranke für $\sum_{i=0}^{\delta n} \binom{n}{i}$:
Sei $p_i := \binom{n}{i}\delta^i(1-\delta)^{n-i}$.
Gesucht ist $\max_i p_i$.

$$\text{Sei } \alpha_i = p_{i+1}/p_i = \frac{\frac{n!}{(n-i-1)!(i+1)!}\delta^{i+1}(1-\delta)^{n-i-1}}{\frac{n!}{(n-i)!(i)!}\delta^i(1-\delta)^{n-i}} = \frac{(n-i)\delta}{(i+1)(1-\delta)}$$

p_i ist maximal, für das kleinste i sodass $\alpha_i \leq 1$.

$$
\begin{aligned}
& & \alpha_i &\leq 1 \\
\Leftrightarrow & & \frac{(n-i)}{(i+1)}\frac{\delta}{(1-\delta)} &\leq 1 \\
\Leftrightarrow & & n\delta - i\delta &\leq 1 - \delta + i - i\delta \\
\Leftrightarrow & & n\delta - 1 + \delta &\leq i
\end{aligned}
$$

Mit $n\delta \in \mathbb{N}$ ist damit p_i maximal für $i = n\delta$

$$
\begin{aligned}
1 = (\delta + (1-\delta))^n & & = \sum_{i=0}^{n} \binom{n}{i} \cdot \delta^i \cdot (1-\delta)^{n-i} \\
\leq (1+n) \max_{0 \leq i \leq n} \binom{n}{i} \cdot \delta^i (1-\delta)^{n-i} & & = (1+n)\binom{n}{n\delta} \cdot \delta^{n\delta}(1-\delta)^{n(1-\delta)}
\end{aligned}
$$

Also insgesamt

$$1 \leq (1+n)\binom{n}{n\delta} \cdot \delta^{n\delta}(1-\delta)^{n(1-\delta)}$$

und somit

$$\sum_{i=0}^{\delta n} \binom{n}{i} \geq \binom{n}{n\delta} \geq \frac{1}{n+1}\left(\left(\frac{1}{\delta}\right)^{\delta} \cdot \left(\frac{1}{1-\delta}\right)^{1-\delta}\right)^n$$

□

3.3 Umwandlung parametrisierter Algorithmen in exakte Algorithmen

Sei Π ein Minimierungsproblem, sei A ein k-parametrisierter Algorithmus für Π, der in Zeit $O^*(c^k)$ die (NP-vollständige) Frage „Gibt es eine Lösung mit Wert $\leq k$?“ beantworten kann.
Sei zu Eingabe I die Menge U eine Obermenge, aus der alle zulässigen Lösungen stammen (Lösungsuniversum, bei VC ist $U = V$, die Menge aller Knoten).
Sei $|U| = n$. Da es 2^n Teilmengen gibt, hat der Brute-Force-Ansatz Laufzeit $O^*(\sum_{i=0}^{n} \binom{n}{i}) = O^*(2^n)$.

Algorithm 13: EXACT (berechnet **Wert** einer optimalen Lösung)

Input : Minimierungsproblem Π, k-parametrischer Algorithmus A für Π mit Laufzeit $O^*(c^k)$, Eingabe I für Π, Lösungsuniversum U zu I
Output: $\Pi_{OPT}(I)$

```
1 Berechne das größte λ mit c^⌊λn⌋ ≤ Σ_{i=⌊λn⌋+1}^{n} (n choose i);
2 for k:=1 to ⌊λn⌋ do
  |  /* Algorithmus A hat Laufzeit O*(c^k)                          */
3 |  if A(I,k) =TRUE then
4 |  |  return k;
5 for k:=⌊λn⌋+1 to n do
  |  /* Nutzung des Brute-Force-Ansatzes über alle k-elementigen Teilmengen von U  */
  |  /* BRUTEFORCE hat Laufzeit O*((n choose k))                    */
6 |  if BRUTEFORCE(Π,I,U,k) =TRUE then
7 |  |  return k;
```

Satz 3.3 *Sei Π ein Minimierungsproblem, I eine Eingabe für Π, U wie oben das zugehörige Lösungsuniversum, $|U| = n$, λ wie im Algorithmus* EXACT *festgelegt und $\lambda \geq 1/2$, A ein k-parametrisierter Algorithmus für Π mit Laufzeit $O^*(c^k)$.* EXACT *löst Π in Zeit $O^*(c^{\lambda n})$.*

BEWEIS: Der Algorithmus 13 (EXACT) ist Correct by Construction.
Laufzeit: $O^*\left(\sum_{i=1}^{\lfloor \lambda n \rfloor} c^i + \sum_{i=\lfloor \lambda n \rfloor+1}^{n} \binom{n}{i}\right) \overset{\lambda \geq \frac{1}{2}}{=} O^*(c^{\lambda n})$. □

Beispiel 3.1 *Falls $c < 4$, so kann für große n ein $\lambda > 1/2$ gewählt werden sodass $c^\lambda < 2$. Damit ergibt sich dann ein Algorithmus (Laufzeit $O^*(c^{\lambda n})$), der asymptotisch schneller ist als der Brute-Force-Ansatz (Laufzeit $O^*(2^n)$).*

Beispiel 3.2 *Algorithmus 11* DIVIDEANDCONQUERVC *ist ein k-parametrisierter Algorithmus für* VC *mit Laufzeit $O^*(2^k)$. Also ist die Konstante $c = 2$. Asymptotisch ist das Abrunden und das „+1“ in der Potenz und im Summenindex der Ungleichung*

$$c^{\lfloor \lambda n \rfloor} \leq \sum_{i=\lfloor \lambda n \rfloor+1}^{n} \binom{n}{i}$$

irrelevant. Unter Verwendung der Ungleichung des Lemmas 3.3 kann eine obere Schranke für λ bestimmt werden, indem folgende Gleichung gelöst wird:

$$2^{\lambda n} = \left(\left(\frac{1}{\lambda}\right)^{\lambda} \cdot \left(\frac{1}{1-\lambda}\right)^{1-\lambda}\right)^{n}.$$

*Das entsprechende λ ist dann $\lambda = 0.77290780478\ldots$ und damit ist $c^\lambda \leq 1.70872$. Die Laufzeit von Algorithmus 13 (*EXACT*) in Verbindung mit Algorithmus 11 (*DIVIDEANDCONQUERVC*) ist damit in $O^*(1.70872^n)$.*

4 Das Erfüllbarkeitsproblem - SAT

Definition 4.1 *Ein* **Literal** *ist eine boolsche Variable, oder deren Negation. Eine* **Klausel** *ist eine Disjunktion (Oder-Verknüpfung) von Literalen. Eine* **konjunktive Normalform - KNF** (conjuntive normal form - CNF) *ist eine Konjunktion (Und-Verknüpfung) von Klauseln. Das* **Erfüllbarkeitsproblem - SAT** (Satisfiability problem) *stellt die Frage, ob für eine boolsche Formel Φ in konjunktiver Normalform über n Variablen $x_1, \ldots, x_n$ und m Klauseln eine Belegung der Variablen existiert, sodass Φ erfüllt ist. Sei $k \in \mathbb{N}$. Eine* **k-KNF** *ist eine KNF, deren Klauseln maximal aus k Literalen besteht.* **k-SAT** *beschränkt die Anzahl der Literale in den Klauseln der konjunktiven Normalform Φ durch k.*

Beispiel 4.1

$$\Phi = (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_1 \vee x_2) \wedge (x_4 \vee \overline{x_3}) \wedge (x_4 \vee \overline{x_1})$$

ist eine 2-SAT Instanz mit 4 Variablen und 5 Klauseln. Φ kann erfüllt werden durch die Variablenbelegung $x_1 =$ True, $x_2 =$ True, $x_3 =$ False, $x_4 =$ True.

Bekannte Komplexitätsklassen von SAT

- SAT ist NP-vollständig.
- 3-SAT ist NP-vollständig.
- 2-SAT ist in P.

4.1 Ein polynomieller Algorithmus für 2-SAT

Sei Φ eine boolsche Formel in 2-KNF, in der alle Klauseln der Länge 1 (iterativ) eliminiert wurden durch die entsprechenden „Zwangsbelegungen“.
Seien l_1 und l_2 Literale, dann gilt: $l_1 \vee l_2 \Leftrightarrow (\overline{l_1} \Rightarrow l_2) \Leftrightarrow (\overline{l_2} \Rightarrow l_1)$.

Algorithm 14: SOLVE2SAT Laufzeit: $O(n + \#Klauseln)$

Input : SAT-Instanz Φ, in der jede Klausel Länge 2 hat
Output: erfüllende Belegung der Variablen falls Φ erfüllbar, sonst NULL

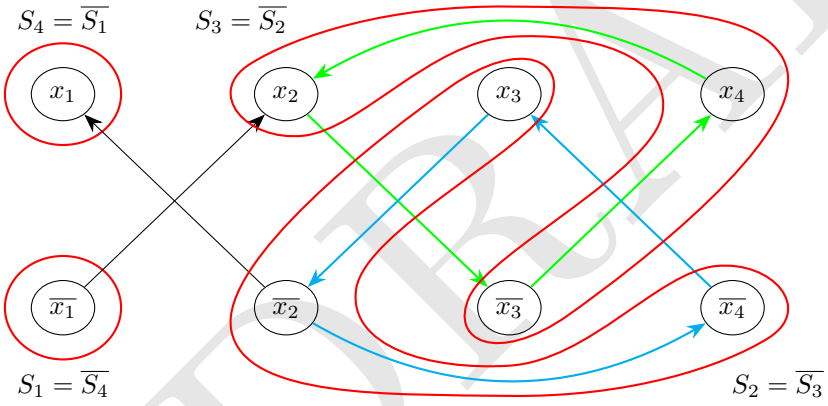
/* Konstruiere den Digraphen $G_\Phi = (V, E)$ */
n:=Anzahl der Variablen in Φ;
V:=$\{x_1, \ldots, x_n\} \cup \{\overline{x_1}, \ldots, \overline{x_n}\}$; /* V seien alle möglichen Literale */
E:=$\emptyset$;
for *Klausel* $(l_1 \vee l_2) \in \Phi$ **do** /* E enthalte zu allen Klauseln die beiden Kanten, */
  E:=$E \cup \{(\overline{l_1} \to l_2), (\overline{l_2} \to l_1)\}$; /* die aus den Implikationen entstehen */
Berechne mit DFS-SZK die starken Zusammenhangskomponenten in G_Φ;
/* Ist in einer erfüllenden Belegung ein Literal l_i TRUE, dann müssen auch alle Literale, die in G_Φ über einen gerichteten Pfad erreichbar sind, TRUE sein. */
/* Ist in einem Superknoten des Superstrukturgraphen bezüglich der starken Zusammenhangskomponenten eine Variable x und ihre Negierung $\overline{x}$ enthalten, dann ist Φ nicht erfüllbar, denn $x \Rightarrow l_1 \Rightarrow l_2 \Rightarrow \ldots \Rightarrow \overline{x} \Rightarrow l'_1 \Rightarrow l'_2 \Rightarrow \ldots \Rightarrow x$ heißt $x \Leftrightarrow \overline{x}$. */
if $\exists$*Variable* $x : x$ *und* $\overline{x}$ *sind in einer starken Zusammenhangskomponente* **then**
  return NULL;
/* Durchlaufe den Superstrukturgraphen in topologischer Reihenfolge. */
while $\exists$ *Komponente* S_i *im Superstrukturgraph mit* ***Eingangs****grad* 0 **do**
  /* S_i enthält ausschließlich Literale, deren Variablen noch nicht belegt wurden */
  Setze alle Variablen aus S_i, sodass alle Literale in S_i den Wert FALSE erhalten;
  /* Diese Variablen können gefahrlos so gesetzt werden, da es keine verbleibenden Kanten in diese Komponente gibt von Literalen, die auf TRUE gesetzt sind beziehungsweise noch auf TRUE gesetzt werden könnten. */
  /* Da die Literale in S_i den Wert FALSE erhalten, so können Kanten aus dieser Komponente heraus entfernt werden, da sie keinen Effekt haben. */
  /* (FALSE→FALSE und FALSE→TRUE sind erlaubt) */
  Entferne den Knoten S_i und seine (ausgehenden) Kanten aus dem Superstrukturgraphen;
  /* Aufgrund der Symmetrie im Graphen G_Φ gibt es immer eine starke Zusammenhangskomponente $\overline{S_i}$, welche die Komplemente der Literale aus S_i enthält und Ausgangsgrad 0 hat. */
  /* Die Variablen sind ja dann bereits so gesetzt worden, dass alle Literale in $\overline{S_i}$ den Wert TRUE aufweisen. */
  /* Dies ist ebenfalls gefahrlos, da es keine verbleibenden Kanten aus dieser Komponente gibt zu Literalen, die auf FALSE gesetzt sind beziehungsweise noch auf FALSE gesetzt werden könnten. */
  /* Da die Literale in $\overline{S_i}$ den Wert TRUE erhalten, so können Kanten die in diese Komponente hinein gehen entfernt werden, da sie keinen Effekt haben. */
  /* (TRUE→TRUE und FALSE→TRUE sind erlaubt) */
  Entferne den Knoten $\overline{S_i}$ und seine (eingehenden) Kanten aus dem Superstrukturgraphen;
/* Der Superstrukturgraph ist nun leer, denn er ist ein DAG und enthält keine Zyklen. Also sind alle Variablen nun gesetzt! */
return berechnete Belegung der Variablen;

Algorithm 15: SOLVE2SAT (Ohne Kommentare)

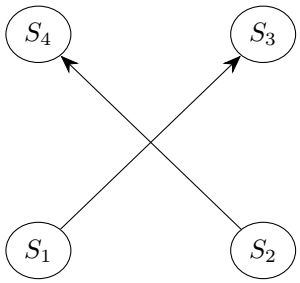
Input : SAT-Instanz Φ, in der jede Klausel Länge 2 hat
Output: erfüllende Belegung der Variablen falls Φ erfüllbar, sonst NULL
`/* Konstruiere den Digraphen` $G_\Phi = (V, E)$ `*/`

```
1  n:=Anzahl der Variablen in Φ;
2  V:={x_1, ..., x_n} ∪ {¬x_1, ..., ¬x_n};
3  E:=∅;
4  for Klausel (l_1 ∨ l_2) ∈ Φ do
5  |  E:=E ∪ {(¬l_1 → l_2), (¬l_2 → l_1)};
6  end
7  Berechne mit DFS-SZK die starken Zusammenhangskomponenten in G_Φ;
8  if ∃Variable x : x und ¬x sind in einer starken Zusammenhangskomponente then
9  |  return NULL;
10 end
11 while ∃ Komponente S_i im Superstrukturgraph mit Eingangsgrad 0 do
12 |  Setze alle Variablen aus S_i, sodass alle Literale in S_i den Wert FALSE erhalten;
13 |  Entferne den Knoten S_i und seine (ausgehenden) Kanten aus dem Superstrukturgraphen;
14 |  Entferne den Knoten ¬S_i und seine (eingehenden) Kanten aus dem Superstrukturgraphen;
15 end
16 return berechnete Belegung der Variablen;
```

Beispiel 4.2 *Sei* $\Phi = (x_1 \vee x_2) \wedge (x_2 \vee \overline{x_4}) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee x_4)$.



(a) Graph G_Φ zu Φ mit Zyklen $x_3 \to \overline{x_2} \to \overline{x_4} \to x_3$ und $\overline{x_3} \to x_4 \to x_2 \to \overline{x_3}$ und seinen starken Zusammenhangskomponenten $S_1, \ldots, S_4$.



(b) Superstrukturgraph zu G_Φ.

Abbildung 27: Graph G_Φ und der zugehörige Superstrukturgraph.

Im Algorithmus SOLVE2SAT *könnte mit* S_1 *oder mit* S_2 *begonnen werden. Hier wird zum Beispiel* S_1 *als erstes ausgewählt. Setze erst alle Literale in* S_1 *auf* FALSE *und damit die Variable* x_1 *auf* TRUE. *Entferne danach die Zusammenhangskomponenten* S_1 *und* $\overline{S_1} = S_4$. *Im nächsten Schritt könnte sowohl* S_2 *als auch* S_3 *ausgewählt werden. Beide Varianten führen zu einer erfüllenden Belegung. Wähle* S_3 *aus. Setze alle Literale in* S_3 *auf* FALSE *und damit die Variable* x_2 *auf* FALSE, x_3 *auf* TRUE *und* x_4 *auf* FALSE. *Entferne danach die Zusammenhangskomponenten* S_3 *und* $\overline{S_3} = S_2$. *Nun ist kein Knoten mehr im Superstrukturgraph übrig und wir erhalten die erfüllende Belegung* $x_1 =$ TRUE, $x_2 =$ FALSE, $x_3 =$ TRUE, $x_4 =$ FALSE.

4.2 Der Algorithmus von Monien/Speckenmeyer für k-SAT

Algorithm 16: MonienSpeckenmeyer (Algorithmus von Monien/Speckenmeyer)

Input : SAT-Instanz Φ
Output: True falls Φ erfüllbar, sonst False

```
1  if Φ =False then
   |  /* Φ enthält leere Klausel, die nicht mehr erfüllt werden kann   */
2  |  return False;
3  else if Φ =True then
   |  /* Φ enthält keine zu erfüllenden Klauseln mehr                  */
4  |  return True;
5  end
6  Wähle eine Klausel C = l_1 ∨ ... ∨ l_k von Φ;
7  if C = l_1 then
8  |  return MonienSpeckenmeyer(Φ_[l_1:=True]);
9  else if C = l_1 ∨ l_2 then
10 |  return MonienSpeckenmeyer(Φ_[l_1:=True]) ∨ MonienSpeckenmeyer(Φ_[l_1:=False,l_2:=True]);
11 else if C = l_1 ∨ l_2 ∨ l_3 then
12 |  return MonienSpeckenmeyer(Φ_[l_1:=True]) ∨ MonienSpeckenmeyer(Φ_[l_1:=False,l_2:=True])
   |         ∨ MonienSpeckenmeyer(Φ_[l_1:=False,l_2:=False,l_3:=True]);
14 else if C = l_1 ∨ ... ∨ l_k then
15 |  for i:=1 to k do
16 |  |  if MonienSpeckenmeyer(Φ_[l_i:=True,∀j<i:l_j:=False]) = True then
17 |  |  |  return True;
18 |  |  end
19 |  end
20 end
21 return False;
```

Sei $t(n)$ die Laufzeit von MonienSpeckenmeyer bei beliebiger Eingabe einer 3-KNF über n Variablen.
Es gilt:
$t(n) = const$ für $n \leq 3$
$t(n) \leq t(n-1) + t(n-2) + t(n-3) + \underbrace{poly(n)}_{\text{Verwaltung}}$

Laufzeit:
$\lambda^3 = \lambda^2 + \lambda + 1$ hat größte Nullstelle $\lambda = 1.8392\ldots$
Somit ist die Laufzeit in $O^*(1.84^n)$.

Verallgemeinerung auf k-SAT:
Rekursion:
$t(n) = const$ für $n \leq k$
$t(n) \leq t(n-1) + \ldots + t(n-k) + poly(n)$

Laufzeit:
$\lambda^k = \lambda^{k-1} + \ldots + \lambda + 1 = \frac{1-\lambda^k}{1-\lambda}$
$\Leftrightarrow \lambda^{k+1} + 1 = 2\lambda^k$.
Lösungen für die größte Nullstelle - Basis der exponentiellen Laufzeit:

k	2	3	4	5	$\to \infty$
größte Nullstelle	$1.618\ldots$	$1.839\ldots$	$1.928\ldots$	$1.966\ldots$	$\to 2$

4.3 Random Walks und Schönings Algorithmus für 3-SAT

4.3.1 Ein randomisierter Polynomzeit Algorithmus für 2-SAT

Algorithm 17: SAT-RandomWalk

Input : SAT-Instanz Φ, maximale Iterationszahl MaxIter
Output: True falls Lösung gefunden wurde, sonst False

```
1  Berechne eine zufällige Belegung der Variablen;
2  if Belegung erfüllt Φ then
3  |  return True;
4  end
5  for i:=1 to MaxIter do
6  |  Wähle eine beliebige Klausel C, die nicht erfüllt ist;
7  |  Wähle zufällig uniform ein Literal der Klausel C und tausche den Wert der zugehörigen Variable;
8  |  if Belegung erfüllt Φ then
9  |  |  return True;
10 |  end
11 end
12 return False;
```

Satz 4.1 *Falls Φ eine erfüllbare 2-SAT Instanz über n Variablen ist und $c \in \mathbb{N}$, dann gibt der Algorithmus 17* SAT-RandomWalk *mit Eingabe Φ und* MaxIter $= 2cn^2$ *mit mindestens Wahrscheinlichkeit $1 - 1/2^c$* True *aus.*

$$Pr\left(\text{SAT-RandomWalk}(\Phi, 2cn^2) = \text{True}\right) \geq 1 - \frac{1}{2^c}.$$

BEWEIS: Sei S eine beliebige aber feste Belegung der Variablen, sodass Φ erfüllt wird.
(Es können mehrere erfüllende Belegungen existieren. Entsprechend kann der Algorithmus auch terminieren, wenn diese erreicht werden. Solche Fälle bleiben jedoch unberücksichtigt wodurch wir eine obere Schranke für die Laufzeit bekommen)
A_i sei die Belegung der Variablen nach i Iterationen.
X_i sei die Anzahl der Variablen, deren Belegung in A_i und S identisch sind.
Der Algorithmus terminiert spätestens mit True, wenn $X_i = n$. Entsprechen bleibt man in diesem Zustand:

$$\forall i : \Pr[X_{i+1} = n \mid X_i = n] = 1$$

Falls $X_i = 0$ so führt jede Änderung eines Variablenwertes zu $X_{i+1} = 1$:

$$\forall i : \Pr[X_{i+1} = 1 \mid X_i = 0] = 1$$

Falls eine Klausel ausgewählt wird, so sind alle Literale darin False. In S muss diese Klausel erfüllt sein und darum muss in S mindestens eines der Literale True sein. Darum gilt:

$$\forall i : \forall 0 < j < n : \Pr[X_{i+1} = j + 1 \mid X_i = j] \geq \frac{1}{2}$$
$$\forall i : \forall 0 < j < n : \Pr[X_{i+1} = j - 1 \mid X_i = j] \leq \frac{1}{2}$$

Wir verschlechtern die Abschätzung nun indem wir annehmen, dass jeweils Gleichheit gilt:

$$\forall i : \forall 0 < j < n : \Pr[X_{i+1} = j + 1 \mid X_i = j] = \frac{1}{2}$$
$$\forall i : \forall 0 < j < n : \Pr[X_{i+1} = j - 1 \mid X_i = j] = \frac{1}{2}$$

Erst ab jetzt ist der Prozess ein Markov-Prozess und lässt sich demnach als Markov-Kette darstellen.

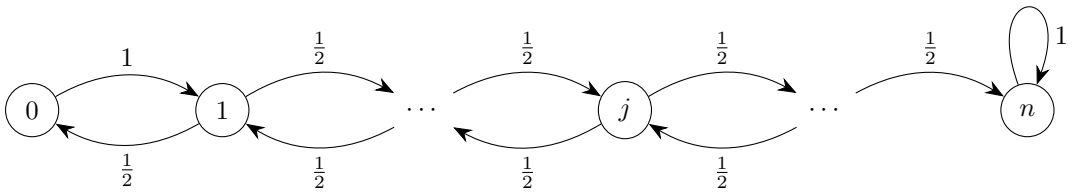


Abbildung 28: Graphische Repräsentation des Prozesses

Sei nun h_j die erwartete Anzahl an Schritten, um n zu erreichen, wenn man in j startet.
Es ergeben sich dadurch folgende Gleichungen:

$$\begin{aligned} h_n &= 0 \\ h_j &= \frac{1}{2}(h_{j-1} + h_{j+1}) + 1 && \forall j \in \{1, \dots, n-1\} \\ h_0 &= h_1 + 1 \end{aligned}$$

Dieses Gleichungssystem hat die eindeutige Lösung $h_j = n^2 - j^2$.
Falls Φ erfüllbar ist, so ist die erwartete Anzahl an Schritten, bis eine erfüllende Belegung der Variablen gefunden wird, durch n^2 begrenzt (im schlimmsten Fall gilt $X_0 = 0$).
Sei T die erwartete Zeit, bis eine erfüllende Belegung gefunden wird. Mit Markov-Ungleichung gilt damit

$$\Pr\left[T \geq 2n^2\right] \leq \frac{\mathrm{E}[T]}{2n^2} \leq \frac{n^2}{2n^2} = \frac{1}{2} \ .$$

Der Algorithmus soll $2cn^2$ viele Iterationen durchführen. Diese können in c Pakete mit je $2n^2$ Iterationen aufgeteilt werden. Je Abschnitt kann nun die Markov-Ungleichung benutzt werden. Der Algorithmus scheitert nur, wenn in allen Abschnitten keine Lösung gefunden wird. Es gilt damit

$$\Pr\left[\text{SAT-RandomWalk}(\Phi, 2cn^2) = \textsc{False}\right] \leq \frac{1}{2^c}$$

und somit auch

$$\Pr\left[\text{SAT-RandomWalk}(\Phi, 2cn^2) = \textsc{True}\right] \geq 1 - \frac{1}{2^c} \ .$$

□

Jeder Iterationsschritt kann in polynomieller Zeit umgesetzt werden.
Somit ist SAT-RandomWalk$(\Phi, 2cn^2)$ ein randomisierten Algorithmus mit polynomieller Laufzeit, welcher mit festlegbarer Mindestwahrscheinlichkeit korrekt arbeitet.

4.3.2 Ein randomisierter $O^*(2^n)$-Zeit Algorithmus für 3-SAT

Wendet man den selben Algorithmus auf 3-SAT an so kann man auch Laufzeitschranken zeigen.

Satz 4.2 *Falls Φ eine erfüllbare 3-SAT Instanz über n Variablen ist, dann ist die erwartete Iterationsanzahl, die der Algorithmus 17* SAT-RandomWalk *benötigt, um eine erfüllende Belegung für Φ zu finden durch $O(2^n)$ begrenzt.*

BEWEIS: Die Vorgehensweise für den Beweis ist analog zur 2-SAT-Variante.
Sei S eine beliebige aber feste Belegung der Variablen, sodass Φ erfüllt wird.
(Es können mehrere erfüllende Belegungen existieren. Entsprechend kann der Algorithmus auch terminieren, wenn diese erreicht werden. Solche Fälle bleiben jedoch unberücksichtigt wodurch wir eine obere Schranke für die Laufzeit bekommen)
A_i sei die Belegung der Variablen nach i Iterationen.
X_i sei die Anzahl der Variablen, deren Belegung in A_i und S identisch sind.
Der Algorithmus terminiert spätestens mit True, wenn $X_i = n$.
Entsprechend bleibt man in diesem Zustand:

$$\forall i : \Pr[X_{i+1} = n \mid X_i = n] = 1$$

Falls $X_i = 0$ so führt jede Änderung eines Variablenwertes zu $X_{i+1} = 1$:

$$\forall i : \Pr[X_{i+1} = 1 \mid X_i = 0] = 1$$

Falls eine Klausel ausgewählt wird, so sind alle Literale darin False. In S muss diese Klausel erfüllt sein und darum muss in S mindestens eines der Literale True sein. Darum gilt:

$$\forall i : \forall 0 < j < n : \Pr[X_{i+1} = j + 1 \mid X_i = j] \geq \frac{1}{3}$$
$$\forall i : \forall 0 < j < n : \Pr[X_{i+1} = j - 1 \mid X_i = j] \leq \frac{2}{3}$$

Wir verschlechtern die Abschätzung nun indem wir annehmen, dass jeweils Gleichheit gilt:

$$\forall i : \forall 0 < j < n : \Pr[X_{i+1} = j + 1 \mid X_i = j] = \frac{1}{3}$$
$$\forall i : \forall 0 < j < n : \Pr[X_{i+1} = j - 1 \mid X_i = j] = \frac{2}{3}$$

Erst ab jetzt ist der Prozess ein Markov-Prozess und lässt sich demnach als Markov-Kette darstellen.

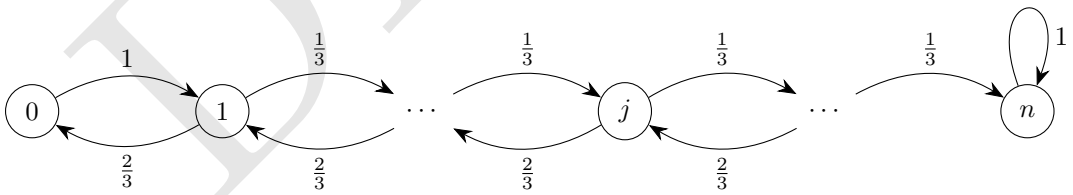


Abbildung 29: Graphische Repräsentation des Prozesses

Sei nun h_j die erwartete Anzahl an Schritten, um n zu erreichen, wenn man in j startet.
Es ergeben sich dadurch folgende Gleichungen:

$$h_n = 0$$
$$h_j = \frac{2}{3}h_{j-1} + \frac{1}{3}h_{j+1} + 1 \qquad \forall j \in \{1, \ldots, n-1\}$$
$$h_0 = h_1 + 1$$

Dieses Gleichungssystem hat die eindeutige Lösung $h_j = 2^{n+2} - 2^{j+2} - 3(n-j)$.
Falls Φ erfüllbar ist, so ist die erwartete Anzahl an Schritten, bis eine erfüllende Belegung der Variablen gefunden wird, durch $2^{n+2} = O(2^n)$ begrenzt (im schlimmsten Fall gilt $X_0 = 0$). □

Somit ergibt sich durch polynomiellen Overhead je Schritt eine Laufzeit von $O^*(2^n)$.

4.3.3 Ein randomisierter $O^*((4/3)^n)$-Zeit Algorithmus für 3-SAT - Schönings Algorithmus

Beobachtungen:

- Wenn A_0 zufällig uniform gewählt wird, so ist X_0 binomial verteilt:

$$\Pr[X_0 = j] = \binom{n}{j} \left(\frac{1}{2}\right)^n \quad .$$

  Es gilt also $\mathrm{E}[X_0] = n/2$. Weiterhin ist die Wahrscheinlichkeit für das Ereignis $\{X_0 > n/2\}$ nicht zu vernachlässigen.

- Im Verlauf des Algorithmus ist es wahrscheinlicher, dass sich X_i in Richtung 0 bewegt anstatt sich in Richtung n zu bewegen. Je länger der Algorithmus läuft, umso wahrscheinlicher ist es, dass sich X_i in Richtung 0 orientiert hat.

Algorithm 18: SchoeningSAT (Schönings 3-SAT Algorithmus)

Input : SAT-Instanz Φ, maximale Anzahl an Restarts MaxRestart
Output: True falls Lösung gefunden wurde, sonst False

```
1 n:=Anzahl Variablen in Φ;
2 for i:=1 to MaxRestart do
3 |   if SAT-RandomWalk(Φ, n) then
4 |   |   return True;
5 |   end
6 end
7 return False;
```

Satz 4.3 *Falls Φ eine erfüllbare 3-SAT Instanz über n Variablen ist, dann ist die erwartete Anzahl an Restarts, die der Algorithmus 18* SchoeningSAT *benötigt, um eine erfüllende Belegung für Φ zu finden durch $O(n \cdot (4/3)^n) = O^*((4/3)^n)$ begrenzt.*

BEWEIS: Sei S eine beliebige aber feste Belegung der Variablen, sodass Φ erfüllt wird.
(Es können mehrere erfüllende Belegungen existieren. Entsprechend kann der Algorithmus auch terminieren, wenn diese erreicht werden. Solche Fälle bleiben jedoch unberücksichtigt wodurch wir eine obere Schranke für die Laufzeit bekommen)
A_i sei die Belegung der Variablen nach i Iterationen.
X_i sei die Anzahl der Variablen, deren Belegung in A_i und S identisch sind.
Der Algorithmus terminiert spätestens mit True, wenn $X_i = n$.
Wir erweitern das Markov-Modell um die Laufzeit nach oben abzuschätzen um unendlich viele Zustände. Es gibt nun für jede ganze Zahl $j \in \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ einen Zustand. Der Startzustand ist weiterhin binomial auf den Zuständen 0 bis n verteilt. Betrachtet man nun eine feste Iterationszahl (hier n) so findet der Algorithmus die Referenzbelegung mindestens, wenn wir uns in einem Zustand j mit $j \geq n$ befinden. Damit bestimmen wir dann eine Mindestwahrscheinlichkeit mit der der Algorithmus nach n Schritten eine erfüllende Belegung findet. Durch diese Abschätzung müssen wir keine Randfälle mehr beachten, was die nachfolgenden Schritte ermöglicht. Die Wahrscheinlichkeiten für die Veränderung der X_i werden ähnlich wie bisher abgeschätzt um die Mindestwahrscheinlichkeit für den Erfolg zu berechnen:

$$\forall i : \forall j : \Pr[X_{i+1} = j + 1 \mid X_i = j] = \frac{1}{3}$$
$$\forall i : \forall j : \Pr[X_{i+1} = j - 1 \mid X_i = j] = \frac{2}{3}$$

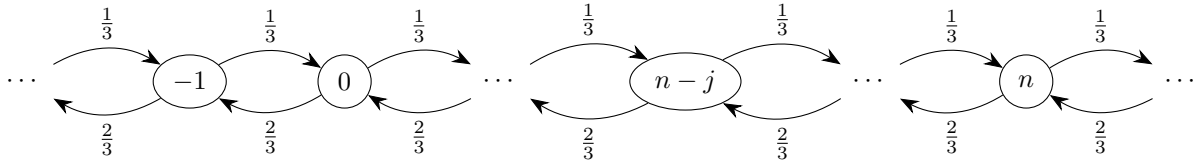


Abbildung 30: Graphische Repräsentation des Prozesses - Abschätzung mit zusätzlichen Zuständen

In einer Sequenz von $j + 2k$ Iterationen genau k Schritte nach unten ($X_{i+1} = X_i - 1$) und $k + j$ Schritte nach oben ($X_{i+1} = X_i + 1$) zu machen ist demnach

$$\binom{j+2k}{k} \cdot \left(\frac{2}{3}\right)^k \cdot \left(\frac{1}{3}\right)^{j+k} .$$

q_i sei nun (eine untere Schranke für) die Wahrscheinlichkeit, dass Schönings Algorithmus n erreicht, wenn der Aufruf von SAT-RandomWalk mit Belegung der Variablen beginnt sodass $X_0 = n - j$. Es gilt

$$q_j \geq \max_{k \in \{0,\ldots,j\}} \binom{j+2k}{k} \cdot \left(\frac{2}{3}\right)^k \cdot \left(\frac{1}{3}\right)^{j+k} \geq \binom{3j}{j} \cdot \left(\frac{2}{3}\right)^j \cdot \left(\frac{1}{3}\right)^{2j}$$

Es gilt

$$\begin{aligned}
1 = \left(\frac{1}{3} + \frac{2}{3}\right)^n &= \sum_{i=0}^{n} \left(\binom{n}{i} \cdot \left(\frac{1}{3}\right)^i \cdot \left(\frac{2}{3}\right)^{n-i} \right) \\
&\leq (n+1) \cdot \max_{i \in \{0,\ldots,n\}} \left(\binom{n}{i} \cdot \left(\frac{1}{3}\right)^i \cdot \left(\frac{2}{3}\right)^{n-i} \right) \\
&= (n+1) \cdot \binom{n}{n/3} \cdot \left(\frac{1}{3}\right)^{n/3} \cdot \left(\frac{2}{3}\right)^{2n/3} \\
\Rightarrow \binom{n}{n/3} &\geq \frac{1}{n+1} \cdot \left(3^{1/3} \cdot \left(\frac{3}{2}\right)^{2/3}\right)^n \qquad (1)
\end{aligned}$$

q sei nun die Wahrscheinlichkeit, dass SAT-RandomWalk(Φ, n) eine erfüllende Belegung findet. Es gilt

$$\begin{aligned}
q &\geq \sum_{j=0}^{n} \Pr[X_0 = n - j] \cdot q_j \\
&\geq \Pr[X_0 = n - n/3] \cdot q_{n/3} \\
&\geq \binom{n}{n/3} \cdot \left(\frac{1}{2}\right)^n \cdot \binom{n}{n/3} \cdot \left(\frac{2}{3}\right)^{n/3} \cdot \left(\frac{1}{3}\right)^{2n/3} \\
&\overset{(1)}{\geq} \left(\frac{1}{n+1} \cdot \left(3^{1/3} \cdot \left(\frac{3}{2}\right)^{2/3}\right)^n \right)^2 \cdot \left(\frac{1}{2} \cdot \left(\frac{2}{3}\right)^{1/3} \cdot \left(\frac{1}{3}\right)^{2/3} \right)^n \\
&= \frac{1}{(n+1)^2} \cdot \left(\frac{3^{((1/3)+(2/3)) \cdot 2} \cdot 2^{1/3}}{2^{(2 \cdot 2/3)+1} \cdot 3^{1/3+2/3}} \right)^n = \frac{1}{(n+1)^2} \cdot \left(\frac{3}{4}\right)^n
\end{aligned}$$

Falls Φ erfüllbar ist, so ist die erwartete Anzahl an Restarts im Algorithmus SchoeningSAT, bis eine erfüllende Belegung der Variablen gefunden wird mittels Abschätzung durch geometrischer Verteilung, durch

$$\frac{1}{q} \leq (n+1)^2 \cdot \left(\frac{4}{3}\right)^n = O\left(n^2 \cdot \left(\frac{4}{3}\right)^n\right) = O^*\left(\left(\frac{4}{3}\right)^n\right) = o(1.333333333334^n)$$

begrenzt. □

5 Permutationsprobleme und Dynamische Programmierung

5.1 Das TSP (Traveling Salesperson Problem)

Gegeben: Vollständiger Graph auf n Knoten
$c : \{1, \ldots, n\}^2 \to I\!N$
$c(i, j)$: „Abstand/Distanz von Knoten i zu Knoten j“
Ein Kreis $K = (u_{i_1}, u_{i_2}, \ldots, u_{i_n}, u_{i_1})$ hat Kosten $c(K) = c(u_{i_n}, u_{i_1}) + \sum_{j=1}^{n-1} c(u_{i_j}, u_{i_{j+1}})$
Gesucht: Hamilton-Kreis K^* mit möglichst niedrigen Kosten

Ein **Hamilton**-Kreis/Pfad ist ein Kreis/Pfad in dem **jeder** Knoten **genau einmal** besucht wird.

„Holzhammer“ probiert $(n-1)!$ viele Permutationen aus.
Laufzeit:

$$O^*((n-1)!) = O^*\left(\sqrt{n-1} \cdot \left(\frac{n-1}{e}\right)^{n-1}\right) = O^*\left(\left(\frac{n}{e}\right)^n\right) \ .$$

Benutze folgende Hilfskonstruktion:
$c(S, k) \quad \hat{=} \quad$ Minimale Kosten eines Hamilton-*Pfads* von Knoten 1 zu Knoten k
auf dem durch die Knoten $\{1\} \cup S$ induzierten Teilgraph, $S \subseteq \{2, \ldots, n\}$, $k \in S$.

Berechnung von $c(S, k)$ mittels folgender rekursiver Beziehung:

$$\begin{array}{ll} \text{Für } S = \{k\} & c(S,k) = c(1,k) \ , \\ \text{für } |S| > 1 & c(S,k) = \underbrace{\min\{c(S \setminus \{k\}, m) + c(m,k) \mid m \in S \setminus \{k\}\}}_{\text{Bellmansche Optimalitätsbeziehung}} \ . \end{array}$$

Abschließend gilt

$$c(K^*) = \underbrace{\min\{c(\{2, \ldots, n\}, k) + c(k, 1) \mid k \in \{2, \ldots, n\}\}}_{\text{Bellmansche Optimalitätsbeziehung}} \ .$$

Berechnung von $c(S, k)$ ist, wenn alle $c(S \setminus \{k\}, m)$ bekannt sind, mit $|S|-1$ Additionen und $|S|-2$ Vergleichen, also in $2 \cdot |S| - 3$ Schritten möglich.
Es gibt $\binom{n-1}{l}$ l-elementigen Mengen S, für die es jeweils l Möglichkeiten für den Knoten k gibt.
Zum Schluss kommen $2n-3$ Schritte zur Berechnung von $c(K^*)$ hinzu ($n-1$ Additionen und $n-2$ Vergleiche).

Laufzeit zur Berechnung von $c(K^*)$:

$$\sum_{l=2}^{n-1} \left[(2l-3) \cdot l \cdot \binom{n-1}{l} \right] + 2n - 3 \underbrace{\leq}_{\substack{\text{da } l<n \text{ und} \\ \sum_{i=0}^{n} \binom{n}{i} = 2^n}} 2 \cdot n^2 \cdot 2^n + 2n - 3 = O(n^2 \cdot 2^n) = O^*(2^n) \ .$$

Die **Bellmansche Optimalitätsbeziehung** besagt, dass sich eine optimale Lösung aus optimalen Teillösungen zusammensetzen lässt. Bei dem TSP lassen sich optimale Hamiltonpfade im Subgraphen (und schließlich der Hamiltonkreis im gesamten Graph) durch Teillösungen ohne den aktuellen Endknoten und den Kosten des verbliebenen direkten Weges berechnen.

Die Rekonstruktion einer optimalen Reihenfolge kann dadurch bestimmt werden, dass man sich bei jedem $c(S, k)$ merkt durch welche Sublösung der minimale Wert erreicht wurde.

5.2 Zeitplanung

Betrachte folgendes *Zeitplanungsproblem* (Scheduling Problem):
Gegeben: n Jobs: $J_1, \ldots, J_n$
Ausführungszeiten $\tau_1, \ldots, \tau_n$
Kostenfunktionen $c_1(t), \ldots, c_n(t)$
Jeder Job J_i benötigt τ_i Schritte zur Abarbeitung. Wenn der Job J_i zum Zeitpunkt t beendet wird, sind seine Kosten $c_i(t)$. Es ist nur eine Maschine verfügbar, welche immer arbeiten muss. Ein Job kann nicht unterbrochen werden. (non-preemptive scheduling)

Die Reihenfolge, in der die Jobs ausgeführt werden, heißt Fahrplan (Schedule) und kann durch eine Permutation π beschrieben werden. $J_{\pi(i)}$ ist der Job, der als i-tes ausgeführt wird. Mit π gegeben ist die Abschlusszeit $t_{\pi(i)}$ von $J_{\pi(i)}$

$$t_{\pi(i)} = \sum_{j=1}^{i} \tau_{\pi(j)}$$

und die Kosten von π sind

$$\sum_{i=1}^{n} c_{\pi(i)}(t_{\pi(i)}) \ .$$

Gesucht: Finde Fahrplan π, der die Kosten minimiert.

Für eine Teilmenge $S \subseteq \{J_1, \ldots, J_n\}$ sei $t_S = \sum_{J_i \in S} \tau_i$.
$opt(S)$ bezeichne die Kosten eines minimalen Fahrplans, der in der Zeit $[0, t_S]$ ausgeführt wird.
Berechnung von $opt(S)$ mittels folgender rekursiver Beziehung:

Für $S = \{J_i\}$	$opt(S) = c_i(\tau_i)$,
für $\lvert S\rvert > 1$	$opt(S) = \underbrace{\min\{opt(S \setminus \{J_i\}) + c_i(t_S) \mid J_i \in S\}}_{\text{Bellmansche Optimalitätsbeziehung}}$.

Laufzeit zur Berechnung von $opt(\{J_1, \ldots, J_n\})$:

$$\sum_{i=1}^{n-1} i \cdot \binom{n}{i} = O(n \cdot 2^n) = O^*(2^n) \ .$$

Auch bei diesem Zeitplanungsproblem kann die Bellmansche Optimalitätsbeziehung angewendet werden, denn die optimale Lösung lässt sich aus Teillösungen mit einem Job weniger und den Kosten des verbliebenen Jobs berechnen.

Die Rekonstruktion einer optimalen Reihenfolge kann dadurch bestimmt werden, dass man sich bei jedem $opt(S)$ merkt durch welche Sublösung der minimale Wert erreicht wurde.

5.3 Matrixmultiplikation

Betrachte folgendes Problem bezüglich Matrixmultiplikation:
Gegeben: n Matrizen: $M_1, \ldots, M_n$
Matrix M_i habe r_{i-1} Zeilen und r_i Spalten.
Die Berechnung der $r_{i-1} \times r_{i+1}$-Matrix $M_i \cdot M_{i+1}$ benötigt mit der Schulmethode $r_{i-1} \cdot r_i \cdot r_{i+1}$ Zahlenmultiplikationen.
Gesucht: Klammerung sodass Zahlenmultiplikationen minimiert werden

Für eine Teilfolge $M_i, \ldots, M_j$ sei $m_{i,j}$ die minimale Anzahl an Zahlenmultiplikationen zur Berechnung von $M_i \cdot \ldots \cdot M_j$.
Berechnung von $m_{i,j}$ mittels folgender rekursiver Beziehung:

$$\begin{aligned} &\text{Für } i = j && m_{i,j} = 0 \ , \\ &\text{für } i < j && m_{i,j} = \underbrace{\min\{m_{i,k} + m_{k+1,j} + r_{i-1} \cdot r_k \cdot r_j \mid i \le k < j\}}_{\text{Bellmansche Optimalitätsbeziehung}} \ . \end{aligned}$$

Laufzeit zur Berechnung von $m_{1,n}$:

$$\sum_{i=1}^{n} \sum_{j=i}^{n} (j - i + 1) = O(n^3) \ .$$

Auch bei diesem Problem kann die Bellmansche Optimalitätsbeziehung angewendet werden, denn die optimale Lösung lässt sich aus Teillösungen bis/ab einem Teilungspunkt und den Kosten der verbliebenen Multiplikation berechnen.

Die Rekonstruktion einer optimalen Reihenfolge kann dadurch bestimmt werden, dass man sich bei jedem $m_{i,j}$ merkt durch welche Sublösung der minimale Wert erreicht wurde.

Beispiel 5.1 *Matrixmultiplikation mit $n = 4$ Matrizen: $r_0 = 10$, $r_1 = 1$, $r_2 = 50$, $r_3 = 20$, $r_4 = 20$.*

Die Reihenfolge

$$(M_1 \underbrace{\cdot}_{10\cdot 1\cdot 50 = 500} M_2) \underbrace{\cdot}_{10\cdot 50\cdot 20 = 10\,000} (M_3 \underbrace{\cdot}_{50\cdot 20\cdot 20 = 20\,000} M_4)$$

benötigt 30 500 *Zahlenmultiplikationen. Die Reihenfolge*

$$((M_1 \underbrace{\cdot}_{10\cdot 1\cdot 20 = 200} M_2) \underbrace{\cdot}_{10\cdot 50\cdot 20 = 10\,000} M_3) \underbrace{\cdot}_{10\cdot 20\cdot 20 = 4\,000} M_4$$

benötigt 14 200 *Zahlenmultiplikationen.*

Zur Bestimmung der optimalen Klammerung werden die $m_{i,j}$ bestimmt. Diese können diagonal von links unten nach rechts oben, oder einfach rekursiv mittels Memoization berechnet werden.

$m_{i,j}$:

$i \backslash j$	1	2	3	4
1	0	500	1 200	1 600
2		0	1 000	1 400
3			0	20 000
4				0

Optimale Teilungspunkte bei der Berechnung von $m_{i,j}$:

$i \backslash j$	1	2	3	4
1	–	1	1	1
2		–	2	3
3			–	3
4				–

$m_{1,4}$ wird folglich nach Matrix 1 *getrennt: $M_1 \cdot (M_2 \cdot M_3 \cdot M_4)$*
$m_{2,4}$ wird nach Matrix 3 *getrennt: $M_1 \cdot ((M_2 \cdot M_3) \cdot M_4)$*
Das ergibt folgende Anzahl an Zahlenmultiplikationen:

$$M_1 \underbrace{\cdot}_{10\cdot 1\cdot 20 = 200} ((M_2 \underbrace{\cdot}_{1\cdot 50\cdot 20 = 1\,000} M_3) \underbrace{\cdot}_{1\cdot 20\cdot 20 = 400} M_4)$$

also in Summe, wie bei $m_{1,4}$ berechnet 1 600.