

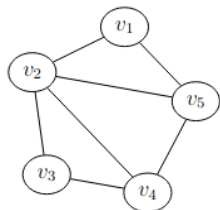
# Effiziente kombinatorische Algorithmen - Zusammenfassung

Autoren: Linda Schneider und Julian Kotzur

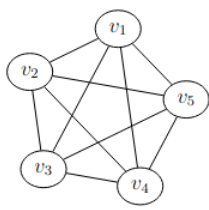
## Grundlagen

### Definition (Graphen).

Sei  $V = \{v_1, \dots, v_n\}$  eine endliche Menge und  $E \subseteq P_2(V) = \{\{u, v\} \mid u, v \in V, u \neq v\}$ . Dann heißt das geordnete Paar  $G = (V, E)$  (endlicher, schlichter, ungerichteter) **Graph**, wobei  $V$  die **Knotenmenge** und  $E$  die **Kantenmenge** von  $G$  ist. Ist  $e = \{u, v\} \in E$  so sind  $u$  und  $v$  **benachbart** (adjazent).  $e = \{u, v\}$  oder auch  $u \overset{e}{-} v$  ist dabei eine Kante.



(a) Einfacher Graph



(c) Vollständiger Graph  $K_5$

### Definition (Grad).

Der **Grad eines Knoten** in  $G = (V, E)$  ist die Anzahl an benachbarten Knoten. Der **Grad eines Graphen** entspricht dem maximalen Grad eines enthaltenen Knotens

### Definition (Regulär).

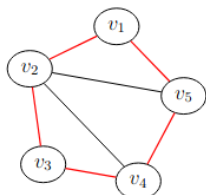
Alle Knoten eines Graphen haben den gleichen Grad

### Definition (Teilgraph).

Seien  $G$  und  $H$  Graphen.  $H$  heißt **Teilgraph** von  $G$ , falls  $V(H) \subseteq V(G)$  und  $E(H) \subseteq E(G)$  gilt.

### Definition (Wege und Kreise).

Ein **Weg** ist eine Folge von Kanten. Ein **einfacher Weg** besucht keine Knoten doppelt. Ein **Kreis** ist ein einfacher Weg, wobei Anfangs- und Endknoten äquivalent sind. Ein **einfacher Kreis** besucht keine Knoten mehrfach



(b) Einfacher Kreis im Graphen

### Definition (Zusammenhängend).

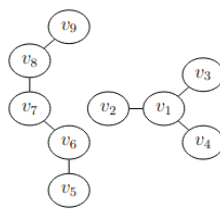
Graph  $G = (V, E)$  heißt **zusammenhängend** (zhgd), falls zwischen je zwei Knoten  $u, v \in V$  ein Weg existiert

### Definition (Wald).

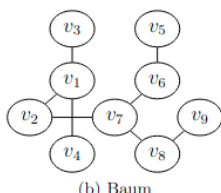
Graph  $G = (V, E)$  ist ein **Wald**, falls  $G$  keine einfachen Kreise enthält

### Definition (Baum).

Graph  $G = (V, E)$  ist ein **Baum**, falls  $G$  ein zusammenhängender Wald ist



(a) Wald



(b) Baum

### Satz (Grad und Knoten).

Bei allen Graphen  $G = (V, E)$  gilt der Zusammenhang:

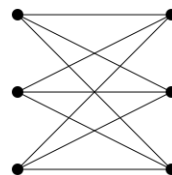
$$\sum_{u \in V} d_G(u) = 2 \cdot |E|$$

### Definition (Disjunktionen).

Zwei Graphen sind disjunkt, wenn es keine Wege von dem einen Graph in den anderen Graphen gibt. Zwei Wege sind disjunkt, wenn es keinen Knoten gibt, der in beiden Wegen enthalten ist

### Definition (Bipartiter Graph).

Ein einfacher Graph  $G = (V, E)$  heißt bipartit, falls sich seine Knoten in zwei disjunkte Teilmengen  $A$  und  $B$  aufteilen lassen, sodass zwischen den Knoten innerhalb beider Teilmengen keine Kanten verlaufen.



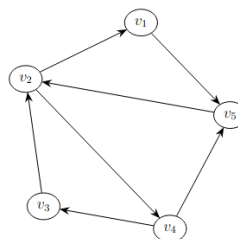
### Definition (Gerichteter Graph).

$G = (V, E)$  ist ein **gerichteter Graph** oder auch Digraph, wenn  $V = \{v_1, \dots, v_n\}, E \subseteq V \times V \setminus \{(v, v) \mid v \in V\}$ .  $d_G^+(v) = |\{e \in E \mid e = (v, w) = v \overset{e}{\rightarrow} w\}|$  heißt **Ausgangsgrad**.  $d_G^-(v) = |\{e \in E \mid e = (v, w) = w \overset{e}{\rightarrow} v\}|$  heißt **Eingangsgrad**

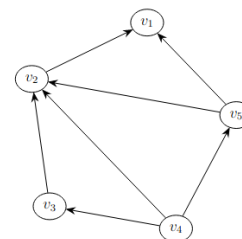
### Definition (Zusammenhängende Digraphen).

Digraph  $G = (V, E)$  heißt **stark zusammenhängend**, falls für je zwei Knoten  $u, v \in V$  gilt: Es gibt einen einfachen Weg von  $u$  nach  $v$  und von  $v$  nach  $u$ . Falls  $G$  ungerichtet zusammenhängend ist, ist  $G$  **schwach zusammenhängend**

→ ein ungerichteter Graph kann zu einem gerichteten werden, indem man  $u - v$  zu  $u \rightarrow v$  und  $v \rightarrow u$  macht



(a) Stark zusammenhängend

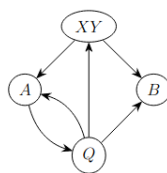


(b) Schwach zusammenhängend

### Satz (Grade und Knoten von Digraphen).

Die Summe von Ausgangsgraden ist gleich der Summe der Eingangsgrade und gleich der Anzahl an Kanten

### Graphen Repräsentation:



#### Inzidenzliste:

$A : \rightarrow Q \rightarrow$   
 $XY : \rightarrow A \rightarrow B \rightarrow$   
 $B : \rightarrow$   
 $Q : \rightarrow A \rightarrow B \rightarrow$   
 $\quad \rightarrow XY \rightarrow$

#### Adjazenzmatrix:

	A	B	Q	xy
A	0	0	1	0
B	0	0	0	0
Q	1	1	0	1
xy	1	1	0	0

# Depth-First-Search

## Algorithmus 1 Algorithmisches Schema zum Durchsuchen

- 1: Starte in beliebigen Knoten  $s \in V$
- 2:  $\mathcal{S} = \{s\}$
- 3: Markiere alle Kanten  $e \in E$  als *unbenutzt*
- 4: **while**  $\exists$  *unbenutzte* Kante  $e$  von  $u \in \mathcal{S}$  nach  $v$  **do**
- 5:     Wähle Knoten  $u$  und *unbenutzte* Kante  $e$
- 6:      $\mathcal{S} = \mathcal{S} \cup \{v\}$
- 7:     Markiere  $e$  als *benutzt*

→ Durchsuchen eines Digraphen, da algorithmisches Schema für ungerichtete Graphen äquivalent ist.

**Lemma.** Gestartet in  $s$  gilt für Knoten in  $\mathcal{S}$  nach Terminierung:

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

**Beweis:**

„ $\Rightarrow$ “ ( $\forall v \in \mathcal{S} \Rightarrow s \rightarrow v$ ) : Offensichtlich nach Konstruktion.

„ $\Leftarrow$ “ : Annahme:  $v \in V$  und  $v$  ist von  $s$  aus erreichbar.

$\Rightarrow$  Zu zeigen:  $v \in \mathcal{S}$ , also:  $\forall v \in V \wedge \exists s \rightarrow v \Rightarrow v \in \mathcal{S}$

→ Da  $v$  von  $s$  erreichbar existiert ein Weg von  $s$  nach  $v$ .

→ Durch Induktion wird gezeigt, dass alle Knoten dieses Weges in  $\mathcal{S}$  liegen.

IA:  $i = 0, v_0 = s \in \mathcal{S}$  UND [IV:]  $v_i \in \mathcal{S}$

IS:  $v_i \in \mathcal{S} \Rightarrow v_{i+1} \in \mathcal{S}$

→ Beweis durch Widerspruch, daher  $v_i \in \mathcal{S}$  und  $v_{i+1} \notin \mathcal{S}$ .

→ Nach Terminierung wäre Kante zwischen  $v_i, v_{i+1}$  *unbenutzt*, was ein Widerspruch zur Abbruchbedingung der while-Schleife wäre. □

### Detailliertere Beschreibung des Schemas:

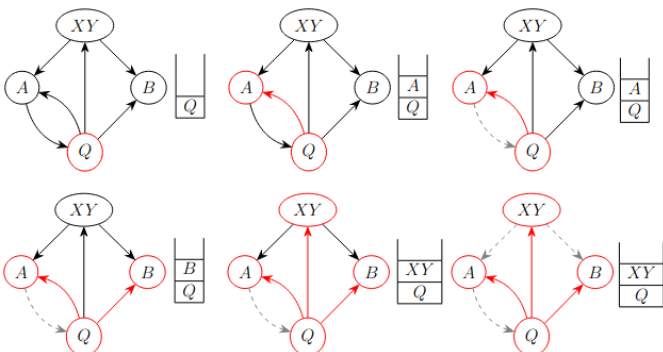
- Kodierung des Graphen: Durch Inzidenzliste mit Speicherplatz  $\mathcal{O}(|V|+|E|)$  oder Adjazenzmatrizen mit Speicherplatz  $\mathcal{O}(|V|^2)$
- Darstellung  $\mathcal{S}$ : Array mit  $\mathcal{S}[v] \in \{\text{besucht}, \text{nichtbesucht}\}$ .
- Reihenfolge von  $u \in \mathcal{S}$ : Auswahl mittels Keller und Inzidenzliste in Verbindung mit den Knotenmarkierungen.

## Algorithmus 2 Algorithmisches Schema für DFS (gerichtet)

- 1: Starte in beliebigen Knoten  $s \in V$ .
- 2: **for**  $v \in V$  **do**
- 3:     **if**  $v \neq s$  **then**  $\mathcal{S}[v] = \text{nichtbesucht}$
- 4:     **else**  $\mathcal{S}[v] = \text{besucht}$
- 5: Lege  $s$  auf Keller
- 6: **while** Keller nicht leer **do**
- 7:     Nimm obersten Knoten  $u$  vom Keller
- 8:     **if** Es gibt *unbenutzte* Kante  $u \xrightarrow{e} v$  **then**
- 9:         Markiere  $e$  als *benutzt*
- 10:         Lege  $u$  zurück auf den Keller
- 11:         **if**  $\mathcal{S}[v] = \text{nichtbesucht}$  **then**
- 12:              $\mathcal{S}[v] = \text{besucht}$
- 13:         Lege  $v$  auf Keller

$\Rightarrow$  **Laufzeit:**  $\mathcal{O}(|V| + |E|)$

$\Rightarrow$  Tiefensuche zerlegt Baum in maximale ZHK

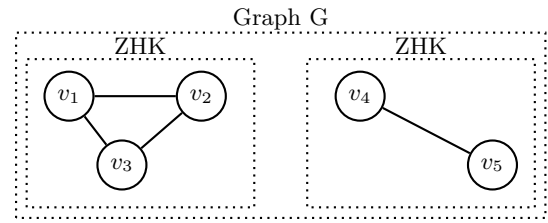


$\Rightarrow$  gestrichelte Linien sind Kanten ohne neuen Knoten

# DFS für ungerichtete Graphen und Berechnung der 2fachen ZHK

**Definition** ((Maximale) Zusammenhangskomponente).

Ein Teilgraph  $H$  von  $G$  heißt **maximale ZHK**, falls  $H$  zhgd. ist und  $H$  disjunkt zu allen Knoten in  $G \setminus \{H\}$  ist



→  $G$  sei im Folgenden immer ein zhgd. Graph.

→ Besuchreihenfolge wird in Array *dfnr* gespeichert

→ Benutzte Kanten werden ohne Richtung in  $T \subseteq E$  gespeichert. (für Tiefensuchbaum)

→ In  $B$  werden unbenutzte Rückkanten gespeichert.

## Algorithmus 3 DFS<sub>u</sub>: Tiefensuche in ungerichteten Graphen

- 1: **procedure** SUCHE( $v$ )
- 2:      $dfnr[v] = \text{zaehler}, \text{zaehler}++$  ▷Setze dfnr des Knotens
- 3:     Markiere  $v$  als *besucht*.
- 4:     **for**  $\forall$  Knoten  $w \in L[v]$  **do** ▷Knoten aus Inzidenzliste
- 5:         **if**  $w = \text{nichtbesucht}$  **then**
- 6:              $T = T \cup (v, w)$  ▷Füge Kante zu Tiefenbaum
- 7:             Suche( $w$ ) ▷Suche von w aus
- 8:
- 9:      $T = \emptyset, \text{zaehler} = 1$  ▷Starte Algorithmus
- 10: **for**  $\forall v \in V$  **do** ▷Initialisierung des Graphen
- 11:     Markiere  $v$  als *nichtbesucht*.
- 12: Wähle  $r \in V$ . ▷Wähle Wurzel
- 13: Suche( $r$ ). ▷Starte Suche von Wurzel aus
- 14: **return**  $T, r, dfnr$

$\Rightarrow$  **Laufzeit:**  $\mathcal{O}(|V| + |E|)$

→ Korrektheit des Algo kann über Rückkanten geprüft werden.

$\Rightarrow$  **Rückkanten springen innerhalb eines Astes, nicht zw. Ästen!**

$\Rightarrow$  **Rückkanteneigenschaft gibt es bei Breitensuche nicht!**

**Lemma** (Korrektheit des Algorithmus).

$DFS_u$  berechnet eine Zerlegung von  $E$  in  $T$  und  $B = E \setminus T$  mit

1. Ergebnis  $H = (V, T)$  ist Baum mit Wurzel  $r : dfnr[r] = 1$ .
2. Ist  $(v \xrightarrow{e} w) \in B$ , so ist  $v$  in  $H$  Nachfolger von  $w$ .

**Beweis:**

→ 1. Gilt, da Knoten nicht mehrfach besucht werden und somit kein Kreis entstehen kann

→ Wenn 2. falsch wäre, wäre  $v$  bis und beim Durchlauf von  $L[w]$  nicht betrachtet worden, was per Konstruktion unmöglich ist.

**Definition** (Artikulationspunkt AP).

Ein Knoten  $v \in V$  heißt Artikulationspunkt (AP) von  $G$ , falls es zwei Knoten gibt, deren Verbindung immer über den Knoten  $v$  läuft. (**Wurzel ist AP, falls sie mehr als einen Ast besitzt**)

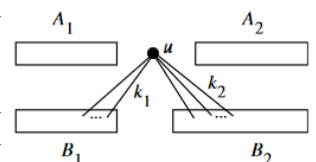
$\Rightarrow$  Entfernt man AP, so zerfällt Graph in min. 2 Graphen/ ZHK

**Definition** (Zweifach Zusammenhängend).

$G$  heißt zweifach zusammenhängend, wenn  $G$  keinen AP besitzt.

**Beispiel.** (Reguläre, bipartite Graphen)

$G$  ist zusammenhängend, regulär, bipartit von Grad  $k \Rightarrow G$  ist aufgrund der Regularität zweifach Zusammenhängend. Würde ein AP existieren, so hat man  $k|A_1|$  Kanten nach  $B_1$ , aber nur  $k|B_1| - k_1$  Kanten nach  $A_1 \Rightarrow$  Entweder nicht regulär oder  $u$  ist kein AP



**Definition** (Zweifache Zusammenhangskomponente).

Sei  $V' \subseteq V$  und  $G' = G|_{V'}$ .  $G'$  heißt zweifache Zusammenhangskomponente, falls  $G'$  zweifach zusammenhängend ist und es keine größere Teilmenge von  $V$  gibt, welche zweifach zusammenhängend ist und  $V'$  enthält.

**Grundidee** zum Finden aller 2fachen ZHK eines Graphen:

1. Enthält  $G$  keinen AP, so ist  $G$  zweifach zusammenhängend.
2. Ist  $v$  ein AP, so besteht  $G \setminus v$  aus zhgd Teilgraphen  $G_i$ .

Wende 1. auf diese  $G_i$  an.

→ **Zusatzvariable**  $tief[v]$ : Kleinste Zahl  $dfnr[u]$ , die in  $H$  über einen gerichteten Weg im Baum von  $v$  nach  $u$  erreichbar ist gefolgt von höchstens einer Rückkante.

→ Dabei gilt immer  $tief(v) \leq dfnr(v)$

**Lemma** (Hinreichende Bedingung zum Finden von AP).

Gegeben sei  $B, T, r$  von  $DFS_u$ . Falls Baumkante  $(u \rightarrow v) \in T$  mit  $dfnr[u] > 1$  und  $tief(v) \geq dfnr[u]$  existiert, so ist  $u$  ein AP.

**Beweis:**

→ Sei  $S \setminus \{u\}$  Menge der Knoten in  $H = (V, T)$  auf dem Weg von Wurzel  $r$  zu Knoten  $u$  UND  $V_v$  sei Menge der Knoten die zu dem Teilbaum gehören, der an  $v$  hängt.

→ Aus Lemma 1: Wegen Ast-Eigenschaft gibt es keine Kanten, die zwischen  $V_v$  und  $V \setminus \{S \cup \{u\} \cup V_v\}$  verlaufen.

→ Wäre  $u$  kein AP, dann gäbe es einen Weg von  $v$  über Kanten zu einem Knoten  $v' \in V_v$  und von dort aus zu einem Knoten  $w \in S$  geben.

→  $tief(v) \leq dfnr[w] < dfnr[u]$  (tief-Nr. wäre falsch) □

**Lemma** (Notwendige Bedingung, erfüllt jeder AP).

Gegeben sind  $B, T, r$  von  $DFS_u(G)$ . Sei  $u$  ein AP von  $G$  mit  $u \neq r$ . Dann gibt es Baumkante  $(u \rightarrow v) \in T$  mit  $tief(v) \geq dfnr[u]$ .

**Beweis:**

→  $G \setminus \{u\}$  besteht aus ZHK  $G_1, \dots, G_m, m \geq 2$ .

→ Jeder Weg zwischen den ZHK führt über  $u$ .

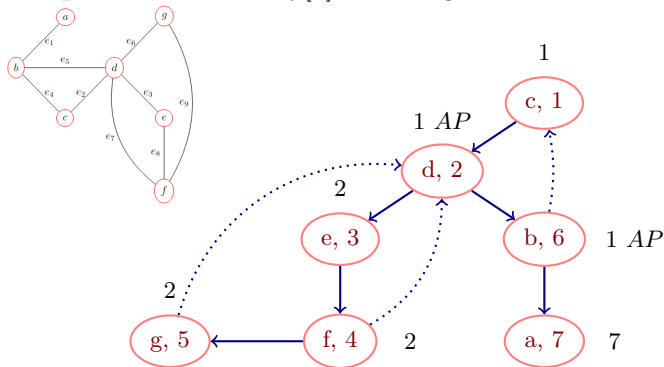
→ Sei oBdA.  $r \in V(G_1)$ .

→ Jede Suche von  $DFS_u$  gelangt über Baumkanten von  $r \rightarrow u$ .

→ Sei  $(u \rightarrow v)$  erste solche Baumkante mit  $v \in V(G_2)$ .

→ Da es keine Kanten zwischen  $V(G_2)$  und  $V \setminus \{V(G_2) \cup \{u\}\}$  gibt, gilt  $tief(v) \geq dfnr[u]$ . □

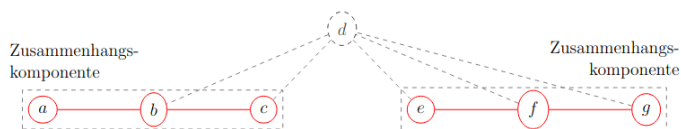
**Beispiel:**  $DFS_u$  mit  $tief[v]$  und AP gestartet in  $c$



→ Neben Knoten sind tief-Nummern

→ In Knoten sind dfnr-Nummern

→ Baum zerfällt in (maximale) Zusammenhangskomponenten, wenn man AP entfernt:



**Definition** (Superstrukturgraph).

Sei  $G$  ein zhgd. Graph. Seien  $u_i$  APs und  $C_j$  zweifache ZHK von  $G$ . Dann ist  $\bar{G}$  Superstrukturgraph von  $G$  mit den Eigenschaften:

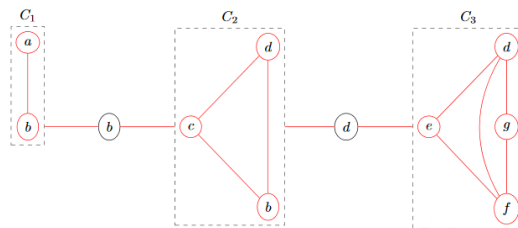
$$V(\bar{G}) = \{u_i \mid i = 1, \dots, p\} \cup \{C_j \mid j = 1, \dots, m\} \text{ und}$$

$$E(\bar{G}) = \{u_i - C_j \mid u_i \in C_j\}$$

→ Zu zusammenhängenden Graphen ist der zugehörige Superstrukturgraph ein Baum!

→  $\bar{G}$  hat min 2 Blätter, Blätter sind 2-fache ZHK von  $G$ .

→ Ist  $C$  2-fache ZHK, die Blatt in  $\bar{G}$  ist, so ist  $C$  mit Restgraph durch genau einen AP verbunden.



**Wichtig für DFS<sub>u</sub>:** Am Ende zusätzlicher Fall: Wurzel als AP!  
Idee: tief und dfnr werden während rekursivem DFS erzeugt und damit AP erkannt. Mit Kellern und AP werden dann 2ZHK ausgegeben. Ergebnis: Alle 2ZHK und Superstrukturgraph

**Algorithmus 4 DFS-2ZK**

```

1:  >Besichtigte-Knoten und Besichtigte-Kanten sind Keller
2:  procedure SUCHE( $v$ )
3:   $tief(v) = dfnr[v] = zaehler, zaehler++$ 
4:  Markiere  $v$  als besucht
5:  for  $\forall w \in L[v]$  do                                >Knoten in Inzidenzliste
6:  if  $w = \text{nichtbesucht}$  then
7:   $T = T \cup (v \rightarrow w)$ 
8:   $vater[w] = v$                                        >Hierarchie merken
9:  Lege  $w$  auf Besichtigte-Knoten
10:  $(v \rightarrow w)$  auf Besichtigte-Kanten
11: Suche( $w$ )                                           >Suche in Knoten weiter (rekursiv)
12: if  $tief(w) \geq dfnr[v]$  und  $v \neq r$  then
13:   Gib 2ZHK aus: Alles über  $w$  und  $w$ . Lösche dies
   danach. Zudem  $v$  und lösche nur  $v \rightarrow w$  (nicht  $v$ )
14:  $tief(v) = \min\{tief(v), tief(w)\}$ 
15: else if  $w \neq vater[v]$  und  $dfnr[v] > dfnr[w]$  then
16:   >Wir haben eine Rückkante von  $v$  nach  $w$  gefunden
17:   Lege  $(v \rightarrow w)$  auf Besichtigte-Kanten
18:    $tief(v) = \min\{tief(v), dfnr[w]\}$ 
19: Start von DFS-2ZK:
20:  $T = \emptyset, zaehler = 1$ 
21: for  $\forall v \in V$  do
22:   Markiere  $v$  als nichtbesucht
23:   Wähle Wurzel  $r \in V$  und starte Suche( $r$ )
24:   >Sonderfall:  $r$  ist Wurzel abschließend behandeln
25:   while Besichtigte-Kanten nicht leer do
26:     Gib alle Kanten auf Besichtigte-Kanten bis  $(r \rightarrow v) \in T$ 
27:     Gib alle Knoten von Besichtigte-Knoten bis  $v$  und  $r$  aus
28:     Lösche Ausgabe, bis auf  $r$ , vom Keller

```

→ **Laufzeit:**  $\mathcal{O}(|V| + |E|)$ .

→ Alles außer DFS hat konstanten Aufwand und fällt weg

→ Man muss Kanten und nicht Knoten betrachten! Je Kante im Baum gibt es konstanten Aufwand an beiden Enden.

→ An einer Seite wird verzögert ein min für tief berechnet

→ Auf der anderen Seite: besucht oder  $dfnr = dfnr+1$

→ Keller  $pop()$  hat auch Konstante Laufzeit

**Definition** (Brücke).

Eine Kante  $e \in E$  heißt Brücke, wenn  $G' = (V, E \setminus \{e\})$  mehr maximale Zusammenhangskomponenten hat als  $G$ .

**Lemma** (Brücken und 2fache ZHK).

Die Brücken eines Graphen  $G$  sind genau die Kanten derjenigen 2fachen ZHK, welche nur aus genau zwei Knoten bestehen.

→ Erkennbar im Algorithmus:  $(u, v) \in T$  und  $dfnr[v] = tief(v)$

# DFS für Digraphen - Schnelle Berechnung der starken Zusammenhangskomponente

**Wichtig:** Es ist folgende Schleife in DFS einzufügen (wg. Wald):  
 $while \exists \text{ nicht-besuchte Knoten } r \in V \text{ do: } \text{suche}(r)$   
 $\Rightarrow$  DFS berechnet zu G den **Tiefensuchwald** H:

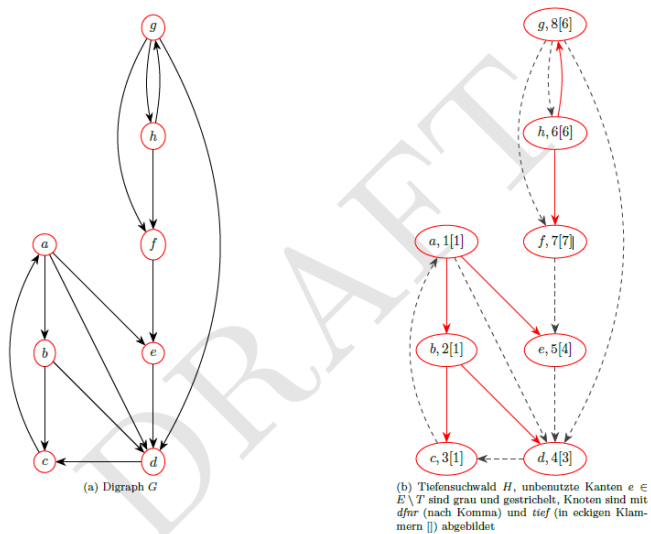


Abb. 1: Suche wird bei a und h gestartet

DFS produziert 4 Kantentypen:

1. Kanten aus T sind **Baumkanten** und führen zu neu gesuchten Knoten (rote Kanten)
2. **Vorwärtskanten:** Kanten  $(a \rightarrow d) \in E \setminus T$  mit Weg in H.
3. **Rückkanten:** Kanten  $(g \rightarrow h) \in E \setminus T$  mit  $g \rightarrow \dots \rightarrow h$  ist Weg in H. ( $\Rightarrow$  Bsp:  $c \rightarrow a, g \rightarrow h$ )
4. **Crosskanten:** Kanten  $(u \rightarrow v) \in E \setminus T$ , wobei es in H weder einen Weg von u nach v noch von v nach u gibt.  
 $\Rightarrow$  Ist  $u \rightarrow v$  eine Crosskante, so gilt:  $dfnr[u] > dfnr[v]$ .  
 $\Rightarrow$  Bsp:  $d \rightarrow c, e \rightarrow d, g \rightarrow f, f \rightarrow e, g \rightarrow d, g \rightarrow f$

**Definition** (Starke Zusammenhangskomponente).

$u \sim v \Leftrightarrow$  In G gibt es einen Weg von u nach v und andersherum.  
 Sei  $V_1, \dots, V_k$  eine Zerlegung von V unter  $\sim$  in die Äquivalenzklassen. Dann heißen die durch die  $V_i$  induzierten Teilgraphen  $G_i$  starke Zusammenhangskomponente von G.

**Einschub Äquivalenzrelation:**

$\rightarrow$  Reflexiv: Es gilt axiomatisch  $v \sim v$ , Symmetrie:  $v \sim u \Rightarrow u \sim v$ , Transitiv:  $u \sim v \wedge v \sim w \Rightarrow u \sim w$

**Definition** (Superstrukturgraph).

Seien  $C_1, \dots, C_k$  starke Zusammenhangskomponenten von G. Dann heißt der Digraph  $\tilde{G}$  mit  $V(\tilde{G}) = \{C_1, \dots, C_k\}$  und  $E(\tilde{G}) = \{C_i \rightarrow C_j \mid i \neq j, \text{ falls } \exists \text{ Kanten zwischen } i \text{ und } j\}$  der Superstrukturgraph von G.

$\Rightarrow$  **Fakt:**  $\tilde{G}$  ist ein kreisfreier Graph (**D**irected **A**cylic **G**raph)

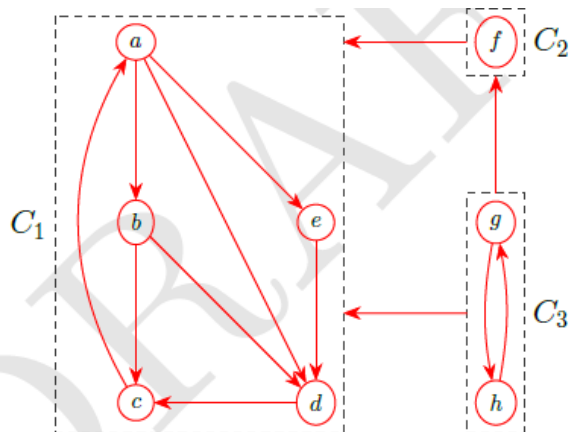


Abb. 2: Superstrukturgraph zu Abbildung 1

Übung:

**Beispiel** (Ist die Sprache endlich?).

Suche in Automat S-ZHK. Für Komponenten mit  $\geq 2$  Zustände muss jeweils Endzustand enthalten sein, da man sonst endlose Zyklen haben kann und die Sprache unendlich wäre.

**Beispiel** (Brückenlos und starke ZHK).

Aufgabe: Mache aus brückenlosen, ungerichteten, zusammenhängendem Graphen eine starke ZHK

$\rightarrow$  Führe Tiefensuche durch und merke Baum- und Rückkanten

$\rightarrow$  Da G brückenlos:  $\exists u \in V \setminus \{r\} : dfnr[u] = tief(u)$

$\rightarrow$  Auch Wurzel muss also über Rückkante erreichbar sein!

$\Rightarrow$  Erstelle gerichteten Kreis für Kriterium der starke ZHK

**Meta-Vorgehen von DFS im Superstrukturgraphen:**

$\rightarrow$  Start in Knoten r in ZHK  $C_1$ : Wird  $C_1$  verlassen, ehe

$C_1$  vollständig durchsucht ist  $\Leftrightarrow d_G^+(C_1) \geq 1$ .

$\rightarrow$  In diesem Fall gelangt DFS zur nächsten starken ZHK, bis eine ZHK  $C_k$  mit  $d_G^+(C_k) = 0$  erreicht ist.

$\Rightarrow$  Diese muss es geben, da  $\tilde{G}$  ein DAG ist.

$\rightarrow C_k$  wird vollständig durchsucht, bevor es rückwärts wieder verlassen wird.

**Identifizieren von ZHK mittels neuer tief-Zahl:**

$\rightarrow$   $tief(v)$  ist die kleinste Zahl  $dfnr[u]$  eines Knotens u, der in der selben starken ZHK wie v liegt und erreichbar ist von v aus auf einem Weg über Baumkanten, gefolgt von **höchstens** einer Rück- oder Crosskante!

$\Rightarrow$  Induktives Berechnungsschema für tief im Tiefensuchwald:

1.  $v \in V : d_G^+(v) = 0$ :

$$tief(v) = \min(\{dfnr[v]\} \cup \{dfnr[u] \mid v \rightarrow u$$

Rück- oder Crosskante, u in gleicher ZHK wie v})

2.  $v \in V : d_G^+(v) \geq 1$ :

$$tief(v) = \min(\{dfnr[v]\} \cup \{dfnr[u] \mid v \rightarrow u$$

Rück- oder Crosskante, u in gleicher ZHK wie v})  $\cup$

$$\{tief(u) \mid (v \rightarrow u) \in T\}$$

**Algorithmus 5** DFS-SZK

- 1: **procedure** S(v)
- 2:  $tief(v) = dfnr[v] = \text{zaehler}$ ,  $\text{zaehler}++$
- 3: Markiere v als besucht.
- 4: Lege v auf Keller K,  $\text{auf\_Keller}[v] = \text{true}$ .
- 5: **for**  $\forall w \in L[v]$  **do** ▷Knoten in Inzidenzliste
- 6: **if**  $w = \text{nicht\_besucht}$  **then**
- 7:  $S(w)$ ,  $tief(v) = \min\{tief(v), tief(w)\}$
- 8: **else if**  $w = \text{besucht}$  **und**  $\text{auf\_Keller}[w]$  **then**
- 9:  $tief(v) = \min\{tief(v), dfnr[w]\}$
- 10: **if**  $tief(v) = dfnr[v]$  **then** ▷Portalknoten erkennen
- 11: **while**  $\text{auf\_Keller}[v]$  **do**
- 12: Nimm obersten Knoten x vom Keller K.
- 13: Gebe x aus.
- 14:  $\text{auf\_Keller}[x] = \text{false}$
- 15: „Ende der SZK“.
- 16:
- 17:  $\text{zaehler} = 1$ ,  $K = \emptyset$ .
- 18: **for**  $v \in V$  **do**
- 19: Markiere v als *nicht-besucht*.
- 20: Setze  $\text{auf\_Keller}[v] = \text{false}$ .
- 21: **while**  $\exists$  nicht besuchter Knoten  $r \in V$  **do** S(r)

**Lemma** (Finden SZK).

Sei t der erste Knoten, der an der rot markierten Stelle die if-Bedingung erfüllt. Dann bilden genau die Knoten oberhalb von t (inkl. t) auf dem Keller K eine SZK.

**Beweis:**

1. Von  $t$  aus sind alle Knoten auf  $K$  oberhalb von  $t$  über Baumkanten erreichbar, da Suche von  $t$  aus zu diesem Zeitpunkt beendet ist.
2. zz. Ist  $v$  von  $t$  aus über Baumkanten erreichbar, und  $v$  auf  $K$ , so gibt es einen Weg von  $v$  zu  $t$ . Konstruktion:  
 → Ist die Suche von  $t$  aus beendet, so ist auch die Suche von  $v$  aus beendet.  
 ⇒  $dfnr[v] > tief(v) \geq tief(t) = dfnr[t]$ .  
 → Sei Knoten  $u$  mit  $dfnr[u] = tief(v)$ .  
 → Ist  $u = t$ , so sind wir fertig.  
 → Ansonsten ist  $u$  auf  $K$  oberhalb von  $t$  und es gibt gerichteten Weg von  $v$  nach  $u$  und wir argumentieren für  $u$  genau wie für  $v$ .  
 ⇒ Es ergibt sich ein Weg in  $G$  von  $v$  nach  $t$ .  
 ⇒ 1. + 2. ⇒ Alle Knoten oberhalb von  $t$  liegen auf SZK.

3. zz.  $C$  enthält keine weiteren Knoten.  
**Annahme:**  $C$  enthält  $y$ , der nicht oberhalb von  $t$  auf  $K$  liegt.  
 ⇒  $dfnr[y] < dfnr[t]$   
 → Alle Knoten  $y \in V(C)$ , die nicht über  $t$  auf  $K$  liegen, haben diese Eigenschaft.  
 → Mindestens ein solcher Knoten  $y'$  ist über Rück- oder Crosskanten erreichbar.  
 ⇒  $x \rightarrow y'$  von  $x$  oberhalb von  $t$  auf  $K$ .  
 → Da  $x, y', t \in V(C)$  sind, gilt  
 $tief(t) \leq tief(x) \leq dfnr[y'] < dfnr[t]$  ❗

**Satz (Laufzeit).**

Der Algorithmus berechnet in  $\mathcal{O}(|V| + |E|)$  die starken Zusammenhangskomponenten.

- Nach vorherigem Lemma wird immer korrekt eine starke ZK berechnet!
- Die starken ZK werden in umgekehrt topologischer Reihenfolge der Knoten im Superstrukturgraph ausgegeben.
- Der Digraph  $G$  ist genau dann ein DAG, wenn alle seine starken ZK aus genau einem Knoten bestehen.

**Flüsse in Netzwerken**

**Definition (Kombinatorisches Optimierungsproblem).**

Kombinatorische Optimierungsprobleme  $\Pi$  sind durch 4 Komponenten charakterisiert:

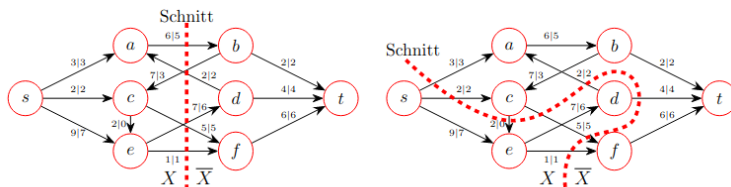
1. Domain  $D$ : Menge der Instanzen/Eingaben
2.  $S(I)$  für  $I \in D$ : Menge der zu  $I$  zulässigen Lösungen
3. Bewertungsfunktion  $f : S(I) \rightarrow \mathbb{N}$
4.  $ziel \in \{min, max\}$

Gesucht ist zu  $I \in D$  eine zulässige Lösung  $\sigma_{opt} \in S(I)$ , so dass  $OPT(I) = f(\sigma_{opt})$  der Wert einer optimalen Lösung ist

**Definition (Flussproblem).**

Ein Netzwerk  $N$  ist ein Digraph  $G$  mit einer Kapazitätsfunktion  $c : E \rightarrow \mathbb{R}^+$  und zwei besonderen Knoten  $s$  (Quelle) und  $t$  (Senke).  $c(e)$  mit  $e \in E$  ist Kapazität. Wir schreiben  $N = (G, s, t, c)$ .

- **Fluss:** Funktion  $f$  die Kanten (reellen) Wert zuweist. Es gilt  $0 \leq f(e) \leq c(e)$  und für Wert der Rückkanten  $f(\overleftarrow{e}) = -f(e)$
- **Konservierungsgesetz:** Gilt für Flüsse und Kanten  $\setminus s, t$ :  
 In = Out:  $\forall v \in V \setminus s, t : \sum_{u:u \rightarrow v} f(u \rightarrow v) = \sum_{u:v \rightarrow u} f(v \rightarrow u)$
- **Wert des Flusses:** Alles was aus der Quelle rausfließt, minus das was wd. reinfließt. Definition auch bei Senke, möglich, wg. Konservierungsgesetz.
- **s,t-Schnitt:** Menge an Kanten zw. 2 Partitionen  $S$  und  $V \setminus S$   
 ⇒ Eine Seite hat immer  $s$  und die andere immer  $t$   
 ⇒ Die Kapazität des Schnittes ist gleich der Summe der Kapazität der Kanten, die von  $S$  nach  $V \setminus S$  führen  
 ⇒ Wichtige Eigenschaft für den Wert eines Schnittes gilt:  
 $|f| = f(S, V \setminus S) - f(V \setminus S, S) \leq f(S, V \setminus S) \leq c(S, V \setminus S)$



(a) Schnitt  $X, \overline{X}$   
 $c(X, \overline{X}) = 6 + 7 + 5 + 1 = 19$   
 $\geq f(X, \overline{X}) = 5 + 6 + 5 + 1 = 12 \geq |f|$   
 $c(\overline{X}, X) = 7 + 2 = 9 \geq f(\overline{X}, X) = 3 + 2 = 5$   
 $|f| = f(X, \overline{X}) - f(\overline{X}, X) = 17 - 5 = 12$

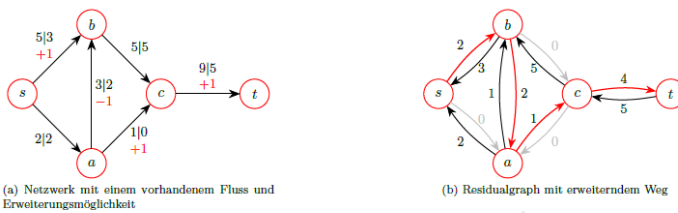
(b) Schnitt  $X, \overline{X}$   
 $c(X, \overline{X}) = 3 + 2 + 2 + 4 + 1 = f(X, \overline{X}) = 3 + 2 + 2 + 4 + 1 = 12 = |f|$   
 $c(\overline{X}, X) = 2 \geq f(\overline{X}, X) = 0$   
 $|f| = f(X, \overline{X}) - f(\overline{X}, X) = 12 - 0 = 12$

**Definition (Erweiternder Weg).**

Ein Weg von  $s$  nach  $t$  über Kanten aus  $E \cup \overleftarrow{E}$  heißt erweiternder Weg  $p$  bezüglich  $f$ , falls

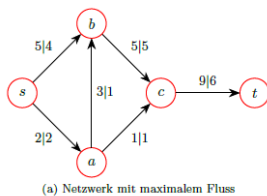
1. für jede Kante  $e$  auf  $p$  mit  $e \in E : f(e) < c(e)$ .
2. für jede Rückkante  $\overleftarrow{e}$  auf  $p$  mit  $e \in E : f(e) > 0$

→ Verbesserung durch erweiternden Weg: Finde Weg  $p$ , suche schmalste Stelle und verbessere Weg um diesen Wert!



(a) Netzwerk mit einem vorhandenem Fluss und Erweiterungsmöglichkeit

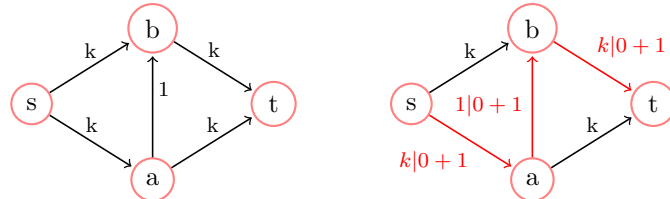
(b) Residualgraph mit erweiterndem Weg



(a) Netzwerk mit maximalem Fluss

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

- **Bemerk:** Bei reellen Kantenkapazitäten kann es passieren, dass es unendliche viele erweiternde Wege gibt und man somit nie auf den maximalen Fluss kommt (weil man die Zahlen beliebig gut approximieren kann)
- **Bemerk:** Folgendes Netzwerk hat eine exponentielle Laufzeit (worst case), da  $2^k$  erweiternde Wege gefunden werden können:



→ **Bemerk:** Rückkanten nach  $s$  oder aus  $t$  können nicht zur Gewinnung des maximalen Flusses beitragen.

**Satz (Ford/Fulkerson).**

1. Satz über erweiternde Wege  
 $f$  ist max Fluss auf  $N \Leftrightarrow$  In  $N$  gibt es keine erweiternden Wege bzgl.  $f$ .
2. Max-Flow-Min-Cut-Theorem  
 Der maximale Wert eines Flusses ist gleich der minimalen Kapazität eines  $s, t$ -Schnittes.

**Beweis:**

- ⇒ “: Enthält  $N$  erweiternden Weg bzgl.  $f$ , so ist  $f$  gemäß unseres Vorgehens noch besserbar und war  $f$  nie maximal
- ⇐ “: Angenommen  $\not\exists$  erweiternden Weg bezüglich  $f$   
 → Jede Berechnung eines erweiternden Wegs endet in  $v$  mit:  
 ⇒ Im Residualgraphen gibt es einen Weg von  $s$  nach  $v$  und  $v \neq t$   
 ⇒ Alle Kanten die von  $v$  starten sind max  
 ⇒ Oder alle Kanten die in  $v$  enden sind 0

→ Formal: Aufteilen der Knotenmenge in zwei Partitionen:  
 $X = \{s\} \cup \{v \in V \mid \exists \text{erweiternder Weg von } s \text{ nach } v\}$ ,  $\bar{X} = V \setminus X$   
 $\Rightarrow |f| = f(X, \bar{X}) = \text{Summe der Flüsse von } X \text{ nach } \bar{X} = c(X, \bar{X})$   
 → Dies beweist auch (2), da nach der Eigenschaft für s,t-Schnitte der Wert aller anderen Schnitte größer gleich  $c(X, \bar{X})$  ist

→ Integrality Theorem: Sind alle Kapazitäten ganzzahlig, so gibt es auch immer einen maximalen ganzzahligen Fluss.  
 → Gestartet mit dem leeren Fluss ist immer ein ganzzahliger erweiternder Weg möglich!

## Algorithmus von Dinic

→ **Insgesamte Laufzeit:**  $\mathcal{O}(|V|^2|E|)$   
 → Restkapazität  $\tilde{c}(e) = \begin{cases} c(e) - f(e) & \text{falls } e \in E \\ f(e) & \text{falls } e \in \bar{E} \end{cases}$   
 → Kanten mit  $\tilde{c}(e) > 0$  heißen **nützliche Kanten**.  
 → Erweiternde Wege bzgl. f enthalten nur nützliche Kanten!  
 → f heißt **Sperrfluss**, falls es keinen erweiternden Weg gibt der nur Kanten aus E (also keine aus  $\bar{E}$ ) enthält  
 →  $\tilde{\delta}(v) = \text{Anzahl Kanten eines kürzesten Weges von } s \text{ zu } v \text{ über nützliche Kanten.}$   
 → Existiert kein solcher Weg, so ist  $\tilde{\delta}(v) = \infty$

### Definition (Geschichtetes Netzwerk).

Geschichtete Netzwerk  $\tilde{N}(f) = ((\tilde{V}, \tilde{E}), s, t, \tilde{c})$  zu  $N, f$  mit

- $\tilde{V} = \{v \in V \mid v \text{ ist auf kürzestem erweiterndem Weg von } s \text{ nach } t, \tilde{\delta}(v) \leq \tilde{\delta}(t)\}$
- $\tilde{E} = \{e = (u, v) \in E \cup \bar{E} \mid e \text{ ist nützliche Kante mit Knoten in } \tilde{V}, \tilde{\delta}(v) = \tilde{\delta}(u) + 1\}$
- $\tilde{c}(e)$  ist Restkapazität.

Am Ende verbleiben die Knoten und Kanten, die zum kürzesten s-t-Weg gehören. Gibt es mehrere s-t-Wege gleicher kürzester Länge, so verbleiben sie alle im geschichteten Netzwerk.

### Bemerkung:

→  $(\tilde{V}, \tilde{E})$  ist ein DAG.  
 → f ist maximaler Fluss, genau dann wenn  $\tilde{V} = \emptyset$ .  
 →  $\tilde{\delta}$  und  $\tilde{N}(f)$  können mittels Breitensuche in  $\mathcal{O}(|E|)$  berechnet werden.

### Lemma (Berechnung f').

Sei  $\tilde{N}(f)$  das zu  $N, f$  gehörige geschichtete Netzwerk, und sei  $\tilde{f}$  ein Sperrfluss auf  $\tilde{N}(f)$ . Dann ist

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

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

→ Gilt wegen Erfülltheit des Konservierungsgesetzes.

### Algorithmus 6 Dinic - Algorithmisches Schema

**Input:** Netzwerk  $N = (G, s, t, c)$

**Output:** Maximaler Fluss f auf N

- 1: Setze Startfluss  $f(e) = 0 \quad \forall e \in E$ .
- 2: **repeat**
- 3: Berechne geschichtetes Netzwerk  $\tilde{N}(f)$ .
- 4: **if**  $\tilde{V} \neq \emptyset$  **then**
- 5: Berechne Sperrfluss  $\tilde{f}$  auf  $\tilde{N}(f)$ .
- 6: Berechne aus f und  $\tilde{f}$  verbesserten Fluss f auf N.
- 7: **until**  $\tilde{V} = \emptyset$
- 8: **return** f

### Satz (Laufzeit).

Die while-Schleife wird höchstens  $|V| - 1$  mal durchlaufen.

### Beweis:

**Zu zeigen:** Die Abstandsfunktion für die Senke  $\tilde{\delta}(t)$  ist streng monoton steigend in  $\tilde{N}(f)$ .

→ Sei f das aktuelle und f' die nächste Iteration.

→ Sei  $p = (s \rightarrow u_1 \rightarrow \dots \rightarrow u_k = t)$  Weg mit  $\delta'(u_i) = i$  in  $\tilde{N}(f')$

**Zeige mit Induktion:**  $\delta'(u_i) \geq \tilde{\delta}(u_i) \quad \triangleright \text{Monotones Wachstum}$

→ IA:  $i = 0 : \tilde{\delta}(s) = 0 = \tilde{\delta}'(u_i)$

→ IS:  $u_i \rightarrow u_{i+1} \Rightarrow \text{Annahme: } \tilde{\delta}'(u_{i+1}) < \tilde{\delta}(u_{i+1})$

⇒  $\exists$  nützliche Kante  $u_i \rightarrow u_{i+1}$  bezüglich leerem Fluss in  $\tilde{N}(f')$

**Fall 1:**  $u_i \rightarrow u_{i+1}$  ist nützliche Kante in  $\tilde{N}(f)$

▷ Kante hätte vorher aktualisiert werden müssen

⇒  $\delta'(u_{i+1}) = 1 + \delta'(u_i) \geq 1 + \tilde{\delta}(u_i) \geq \tilde{\delta}(u_{i+1}) \quad \nabla$

**Fall 2:**  $u_i \rightarrow u_{i+1}$  ist keine nützliche Kante in  $\tilde{N}(f)$

▷ Berechnung des geschichteten Netzwerks wäre falsch

→ Kante  $u_{i+1} \rightarrow u_i$  muss im vorherigen Schritt aktualisiert worden sein.

→ Kante  $u_{i+1} \rightarrow u_i$  lag in  $\tilde{N}(f)$  auf kürzestem Weg von s zu t

⇒  $\tilde{\delta}(u_{i+1}) + 1 = \tilde{\delta}(u_i)$

⇒  $\delta'(u_{i+1}) = 1 + \delta'(u_i) \geq 1 + \tilde{\delta}(u_i) \geq 2 + \tilde{\delta}(u_{i+1}) > \tilde{\delta}(u_{i+1}) \quad \nabla$

**Noch zu zeigen:**  $\delta'(t) > \tilde{\delta}(t) \quad \triangleright \text{Streng monotonen Wachstum}$

→ Unter Annahme von Gleichheit wären alle Knoten in der nächsten Iteration gleich ⇒ f wäre kein Sperrfluss gewesen!  $\nabla$

### Algorithmus 7 Sperrfluss

- 1: **In:**  $\tilde{N}(f)$  mit Graph mit Durchmesser k
- 2: Setze Fluss  $\tilde{f}$  von  $\tilde{N}(f)$  auf 0 ▷ Starte mit leerem Fluss
- 3: **while**  $t \in V_k$  **do** ▷ Solange Ziel noch nicht in Graph
- 4:  $v = t, a = \infty$
- 5: **for**  $i = k, \dots, 1$  **do** ▷ Berechne erweiternden Weg
- 6: Wähle Kante  $e_i = (u, v)$
- 7:  $a = \min\{a, \tilde{c}(e)\}, v = u$  ▷ a speichert Bottleneck
- 8: Aktualisiere in  $\tilde{N}(f)$  den Fluss mit konstruierten Weg um a und reduziere Kapazitäten um a (a wird entfernt)
- 9: Entferne unnütze Knoten und Kanten.
- 10: **return** f

→ **Laufzeit:**  $\mathcal{O}(|V||E|)$  ( $|E|$  Anzahl Repeat Durchläufe)

→ Zuerst wird Weg von s nach t berechnet. Dann wird  $\tilde{f}$  auf  $\tilde{N}(f)$  aktualisiert und min. eine Kante (mit a) wird 0 und entfernt ⇒ Rest wird gelöscht: Repeat Schleife wird mit gleichen Ausgangsbedingungen neu gestartet

⇒ Jede Iteration wird mindestens eine Kante entfernt und höchstens  $|V|$  Knoten betrachtet.

## Parametrisierte Komplexität und VC

### Definition (Vertex Cover).

Sei G ein ungerichteter Graph. Ein Knotenüberdeckung ist eine Knotenmenge  $C \subseteq V$ , sodass jede Kante „überdeckt“ wird. Entscheidungsproblem:

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

Das Optimierungsproblem  $VC_{opt}$  bestimmt ein minimales VC.

→  $VC_{ent}$  ist NP-vollständig

→  $VC_{opt}$  in Polyzeit lösbar  $\Leftrightarrow VC_{ent}$  in Polyzeit entscheidbar

→  $\mathcal{O}^*$ -Notation: Nur exponentieller Anteil bleibt erhalten!

→ **Relevante Abschätzung** mit  $\delta \in ]0, \frac{1}{2}[$ :

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

⇒ Prof. Wanka findet den Beweis cool, weil dieser darauf beruht die 1 komplex umzuschreiben (Rest wsl. unwichtig):

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

# Ein exakter Algorithmus für VC

## Algorithmus 8 Minimum Vertex Cover MVC

```

1: In:  $G = (V, E)$  Out: minimales  $VC_{opt}$ 
2: procedure MVC( $VC, G$ )  $\triangleright$ Setze vorher  $VC_{opt} = VC$ 
3:   if  $|E(G)| = 0$  then  $\triangleright$ Es gibt keine Kanten mehr
4:     if  $|VC| < |VC_{opt}|$  then  $VC_{opt} = VC$ 
5:   return  $VC_{opt}$ 
6:   while  $\exists(u - v) \in E(G) : deg_G(u) = 1$  do
7:      $\triangleright$ Nachbarn von Knoten mit Grad 1 sind immer in  $VC_{opt}$ 
8:      $VC = VC \cup \{v\}, G = G \setminus \{v\}$ 
9:   if  $\forall v \in V(G) : deg_G(v) \in \{0, 2\}$  then
10:     $\triangleright$ Nur noch Knoten mit Grad 0 / 2
11:     $\triangleright \exists$  nur noch Kreise:  $VC_{opt}$  effizient berechenbar
12:    Bestimme optimales Cover und füge es VC hinzu
13:     $E(G) = \emptyset, MVC(VC, G)$ 
14:  else  $\triangleright$ Es gibt noch Knoten mit Grad  $\geq 3$ 
15:    Wähle  $v \in V(G)$  mit  $deg_G(v) = k \geq 3$ 
16:    Bestimme Nachbarn von  $u_1, \dots, u_k$  von  $v$ 
17:     $MVC(VC \cup \{v\}, G \setminus \{v\})$ 
18:     $MVC(VC \cup \{u_1, \dots, u_k\}, G \setminus \{u_1, \dots, u_k\})$ 

```

**Laufzeit:**  $VC_{opt}$  wird in Zeit  $\mathcal{O}^*(1.38^n)$  berechnet

### Beweis:

→ Für jeden Knoten  $v$  gilt:  $v$  ist im VC oder alle seine Nachbarn!  
 ⇒ Algorithmus ist „Correct by construction“.  
 → Laufzeit sei  $t(n)$  bei  $n$  Knoten.  
 ⇒  $t(n) \leq t(n-1) + t(n-4) + poly(n)$ .  
 → Löse zugehörige lineare Rekurrenz (Differenzgleichung)  
 ⇒  $\lambda^4 = \lambda^3 + 1$  mit NS  $\lambda \approx 1.3803 \Rightarrow$  Laufzeit  $\mathcal{O}^*(1.3803^n)$  □

## Parametrisierte Komplexität

→ Bislang: Worst-Case Laufzeit in der Eingabelänge  
 → Jetzt: Extrahieren von Parametern aus der Eingabe, die nicht von der Eingabelänge abhängen. Laufzeit soll Abhängigkeit von nur diesen Parametern angeben werden!

### Definition (Par-parametrisierter Polynomzeit-Algo).

Sei  $L$  ein Entscheidungsproblem. Eine beliebige in Polyzeit berechenbare Funktion  $Par : L \rightarrow \mathbb{N}$  nennt man **Parametrisierung** von  $L$ . Ein Algorithmus  $A$  ist ein **Par-parametrisierter Polynomzeit-Algorithmus** für  $L$ , falls gilt:

- $A$  löst das Entscheidungsproblem  $L$ .
- Es gibt Funktion  $g : \mathbb{N} \rightarrow \mathbb{N}$  und Polynom  $p : \mathbb{N} \rightarrow \mathbb{N}$ , sodass Laufzeit von  $A$  für jede Eingabe  $I$  in  $\mathcal{O}(g(Par(I)) \cdot p(|I|))$  existiert dieser, so nennt man  $L$  **fixed-parameter tractable in Bezug auf Par.** (für VC:  $Par(I) = |I|, g(n) = 2^n$ )

### Problemkern-Methode:

- Reduziere Eingabe  $I$  auf gleichwertige Instanz  $I'$  (Problemkern), wobei die Größen von  $I'$  nur von Parameter  $k$  abhängen (Kernelization)
- Löse  $I'$  und übertrage Antwort auf  $I$ .

## Algorithmus 9 DivideAndConquerVC( $G, k$ ) DAC

**In:**  $G = (V, E), k$  **Out:** Antwort:  $|VC_{opt}| \leq k$ ?

```

1: if  $E = \emptyset$  then return True
2: else if  $k = 0$  then return False
3: Wähle Kante  $\{u, v\} \in E$ .
4:  $G_1 = G|_{V \setminus \{u\}}, G_2 = G|_{V \setminus \{v\}}$ 
5: return  $DAC(G_1, k-1)$  or  $DAC(G_2, k-1)$ 

```

→ **Laufzeit:**  $\mathcal{O}(2^k \cdot (|V| \cdot |E|))$

## Problemkerngrundlagen für Buss & Goldsmith:

- Hat  $G$  ein VC der Größe  $k \Rightarrow$  alle Knoten mit Grad größer  $k$  müssen in VC  $\triangleright$ Sonst müssten mehr als  $k$  Nachbarn ins VC
- Für einen Graphen ohne isolierte Knoten gilt: Wenn alle Knoten Grad  $\leq k$  haben und  $G$  ein VC der Größe  $m$  besitzt, so hat  $G$  höchstens  $m \cdot k$  Kanten  $\triangleright$ Oberste mögliche Schranke

## Algorithmus 10 Buss & Goldsmith

**In:**  $G = (V, E), k$  **Out:** Antwort:  $|VC_{opt}| \leq k$ ?

```

1:  $H = \{v \mid deg_G(v) > k\}$   $\triangleright$ Kernelization
2: if  $|H| > k$  then return False  $\triangleright$ Kein VC möglich
3:  $G' = G \setminus H$ 
4: Entferne alle isolierten Knoten aus  $G'$ .
5:  $m = k - |H|$ 
6: if  $G'$  mehr als  $m \cdot k$  Kanten enthält then  $\triangleright$ Kernelization
7:   return False  $\triangleright$ Kein VC möglich
8: /*Löse Problem auf kleineren Graphen  $G'$  mit  $m$  statt  $k$ */
9: Löse  $VC_{ent}$  mit Holzhammer DAC oder MVC
10: return Ergebnis

```

→ **Laufzeit** mit Holzhammer:  $\mathcal{O}(kn + 2^k k^{2k+2})$

→ **Laufzeit** mit DAC:  $\mathcal{O}(kn + 2^k k^4)$ , mit MVC:  $\mathcal{O}(kn + 1.38^{2k^2})$

⇒ Kernelization benötigt  $\mathcal{O}(kn)$

⇒ Eigenschaften  $G'$ :  $|E(G')| \leq km \leq k^2, |V(G')| \leq 2km \leq 2k^2$

⇒ Der Additiven Zusammenhang ist stärker als benötigt, ein rein Multiplikativer Zusammenhang wäre per Definition auch erlaubt

## Umwandlung parametrisierter Algorithmen in exakte Algorithmen

→ Sei  $\Pi$  Minimierungsproblem und  $A$   $k$ -parametrisierter Algorithmus, der in Zeit  $\mathcal{O}^*(c^k)$  die Entscheidungsfrage löst.  
 → Sei zu Eingabe  $I$  die Menge  $U$  eine Obermenge, aus der alle zulässigen Lösungen stammen.  
 → Mit  $|U| = n$  gibt es  $2^n$  Teilmengen.  
 ⇒ Holzhammer hat Laufzeit  $\mathcal{O}^*(\sum_{i=0}^n \binom{n}{i}) = \mathcal{O}^*(2^n)$ .

## Algorithmus 11 Exakt

```

1: In:  $\Pi$ , Algo.  $A$ , Eingabe  $I$  und Lösungen  $U$  Out:  $\Pi_{opt}(I)$ 
2:  $\triangleright$ Zuerst: Berechne maximalen Schnittpunkt der Laufzeiten
3: Berechne größtes  $\lambda$  mit  $c^{\lfloor \lambda n \rfloor} \leq \sum_{i=\lfloor \lambda n \rfloor + 1}^n \binom{n}{i}$ 
4: for  $k = 1, \dots, \lfloor \lambda n \rfloor$  do  $\triangleright$ Algo  $A$ 
5:   if  $A(I, k) = True$  then return  $k$ 
6: for  $k = \lfloor \lambda n \rfloor + 1, \dots, n$  do  $\triangleright$ Holzhammer
7:   if Holzhammer( $\Pi, I, U, k$ ) = True then return  $k$ 

```

→ **Laufzeit:**  $\mathcal{O}^*(\sum_{i=1}^{\lfloor \lambda n \rfloor} c^i + \sum_{i=\lfloor \lambda n \rfloor + 1}^n \binom{n}{i}) \stackrel{\lambda > 1/2}{\equiv} \mathcal{O}^*(c^{\lambda n})$

→ Falls  $c < 4$  kann für große  $n$  ein  $\lambda > 1/2$  gewählt werden, so dass der Algo. asymptotisch schneller ist als Brute-Force.

→ **Exakt** mit DAC (für  $A$ ) hat Laufzeit  $\mathcal{O}^*(1.70872^n)$

→ **Exact** ist Correct by Construction

→ **Idee:** Kombination zweier Algos  $\Rightarrow$  Gemeinsamer Worst Case ist kleiner als Worst Case beider zugrundeliegenden Algos

## Das Erfüllbarkeitsproblem - SAT

### Definition (SAT).

Gegeben ist eine boolesche Formel mit Klauseln in konjunktiver Normalform (CNF, verundet). Die Klauseln bestehen aus veroderten Literalen (boolesche Variable oder Negation). In SAT wird eine gültige Belegung gesucht.  $k$ -SAT beschränkt die Anzahl der Literale in den Klauseln der KNF durch  $k$ .

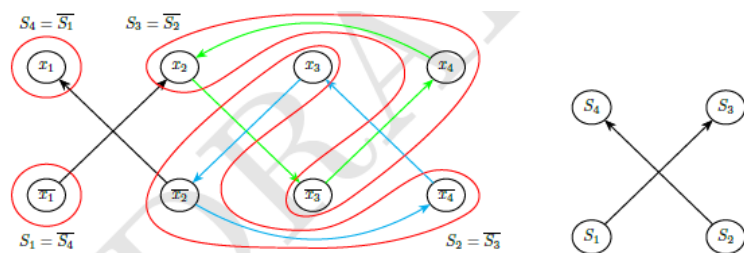
**Beispiel:**  $\Phi = (x_1 \vee \bar{x}_2) \wedge (x_2 \vee x_3) \Rightarrow (x_1 \vee \bar{x}_2)$  ist Klausel,  $x_2, \bar{x}_2$  ein Literal,  $x_1 = \top \wedge x_3 = \top$  eine erfüllende Variablenbelegung

# Polynomieller Algorithmus für 2-SAT

## Algorithmus 12 Solve2SAT

- 1:  $n = \text{Anzahl der Variablen in } \Phi$
- 2:  $V = \{x_1, \dots, x_n\} \cup \{\bar{x}_1, \dots, \bar{x}_n\}, E = \emptyset$
- 3: Füge zu E zu allen Klauseln die möglichen Kanten hinzu.
- 4: Berechne mit DFS-SZK die starken ZK in  $G_\Phi$
- 5: Sind  $x$  und  $\bar{x}$  in einer ZK, **return** Null
- 6: **while**  $\exists S_i$  im Superstrukturgraph mit Eingangsgrad 0 **do**
- 7:     Setze Variablen aus  $S_i$  so, dass Literale in  $S_i$  False sind.
- 8:     /\* Keine Kante hier könnte auf True gesetzt werden \*/
- 9:     Entferne  $S_i$  und ausg. Kanten aus Superstrukturgraph
- 10:    Entferne  $\bar{S}_i$  und ausg. Kanten aus Superstrukturgraph
- 11: **return** Berechnete Belegung der Variablen

**Beispiel.**  $(\Phi = (x_1 \vee x_2) \wedge (x_2 \vee \bar{x}_4) \wedge (\bar{x}_2 \vee \bar{x}_3) \wedge (x_3 \vee x_4))$   
 → **Kantenerstellung:**  $x_i \vee x_j \Rightarrow (\bar{x}_i \Rightarrow x_j, \bar{x}_j \Rightarrow x_i)$   
 → Idee des Setzens von  $S_i = \perp$ : Aus  $\perp$  kann man alles folgern!



Beginnt man mit  $S_1$ , so setzt man erst alle Literale aus  $S_1$  auf False und entfernt ZK  $S_1$  und  $\bar{S}_1 = S_4$  aus  $G_\Phi$ . Anschließend kann  $S_3$  gewählt werden ...  
 ⇒ Erhalte Belegung:  $x_1 = x_3 = \text{True}, x_2 = x_4 = \text{False}$

**Laufzeit:**  $\mathcal{O}(\#\text{Variablen} + \#\text{Klauseln})$   
 ⇒ 1x Superstrukturgraph und 1x topologisch Durchlaufen  
 ⇒  $\mathcal{O}(n + m) + \mathcal{O}(n + m) = \mathcal{O}(n + m)$

## Algorithmen für k-SAT

### Algorithmus 13 MonienSpeckenmeyer MS

- 1: **if**  $\Phi = \text{False}$  **then return** False
- 2: **if**  $\Phi = \text{True}$  **then return** True
- 3:
- 4: Wähle Klausel  $C = l_1 \vee \dots \vee l_k$ .
- 5: **if**  $C = l_1$  **then return** MS( $\Phi_{[l_1=\text{True}]}$ )
- 6: **else if**  $C = l_1 \vee l_2$  **then**
- 7:     **return** MS( $\Phi_{[l_1=\text{True}]}$ )  $\vee$  MS( $\Phi_{[l_1=\text{False}, l_2=\text{True}]}$ )
- 8: **else if**  $C = l_1 \vee l_2 \vee l_3$  **then**
- 9:     **return** MS( $\Phi_{[l_1=\text{True}]}$ )  $\vee$  MS( $\Phi_{[l_1=\text{False}, l_2=\text{True}]}$ )  $\vee$   
       MS( $\Phi_{[l_1=\text{False}, l_2=\text{False}, l_3=\text{True}]}$ )
- 10:
- 11: **else if**  $C = l_1 \vee \dots \vee l_k$  **then**
- 12:     **for**  $i = 1, \dots, k$  **do**
- 13:         **if** MS( $\Phi_{[l_i=\text{True}, \forall j < i: l_j=\text{False}]}$ ) **then return** True
- 14: **return** False

**Laufzeit:**  
 → Sei  $t(n)$  Laufzeit mit beliebiger 3-KNF über  $n$  Variablen  
 ⇒  $t(n) = \text{const}$  für  $n \leq 3 \Rightarrow \text{poly}(n)$   
 ⇒  $t(n) \leq t(n-1) + t(n-2) + t(n-3) + \text{poly}(n)$   
 ⇒ Laufzeit entspricht größte Nullstelle von  $\lambda^3 = \lambda^2 + \lambda + 1$   
 ⇒ Laufzeit in  $\mathcal{O}^*(1.84^n)$   
 → Verallgemeinert auf k-SAT:  
 ⇒  $t(n) \leq t(n-1) + \dots + t(n-k) + \text{poly}(n)$   
 ⇒  $\lambda^k = \lambda^{k-1} + \dots + \lambda + 1 = \frac{1-\lambda^k}{1-\lambda} \Leftrightarrow \lambda^{k+1} + 1 = 2\lambda^k$

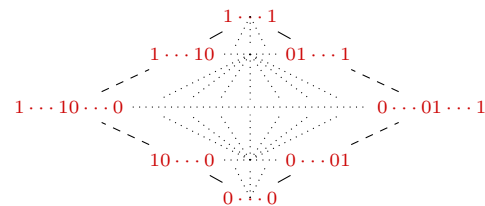
k	2	3	4	5	→ ∞
größte Nullstelle	1.62	1.84	1.93	1.97	→ 2

# Algorithmen für SAT mit Hamming-Kugeln

**Definition** (Hamming-Abstand und Hamming-Kugel).

Der **Hamming-Abstand**  $h(a, b) = \sum_{i=1}^n |a_i - b_i|$  zweier 0-1-Folgen  $a, b$  gibt an, um wie vielen Stellen sich die Folgen unterscheiden. Die **Hamming-Kugel**  $\mathcal{H}(a, d) = \{b \mid h(a, b) \leq d\}$  mit einem Radius  $d$  um  $a$  enthält alle Folgen  $b$ , welche sich an höchstens  $d$  Stellen von  $a$  unterscheiden.

Hamming Kugel ist ein Hypercube mit Dimensionen der Länge der Binärzahlen. Schaubild rechts:



### Algorithmus 14 LocalSearch( $\Phi, a, d$ )

- 1: **Input:** Instanz  $\Phi$ , Belegung  $a = (a_1, \dots, a_n)$  für  $\Phi$ , Distanz  $d$
- 2: **if**  $a$  erfüllt  $\Phi$  **then return** True
- 3: **if**  $d = 0$  **then return** False
- 4: Finde Klausel  $C = l_1 \vee \dots \vee l_k$ , die von  $a$  nicht erfüllt wird
- 5: Für  $i \in \{1, \dots, k\}$  sei  $\bar{a}_i$  Belegung, wo in  $a$  nur Variable  $l_i$  invertiert ist
- 6: **return** LocalSearch( $\bar{a}_1, d-1$ )  $\vee \dots \vee$  LocalSearch( $\bar{a}_k, d-1$ )

→ Laufzeit:  $\mathcal{O}^*(k^d)$  (falls  $\Phi$  eine k-KNF ist)

**Lemma** (LocalSearch durchmustert Hamming-Ball).

Für jede beliebige Belegung  $a$  durchmustert LocalSearch( $a, d$ ) den Hamming-Ball  $\mathcal{H}(a, d)$  und gibt genau dann True aus, falls es eine gültige Belegung im Hamming-Ball gibt.

**Beweis:**

- Falls erfüllende Belegung  $b$  in  $\mathcal{H}(a, d)$  existiert, gilt  $h(a, b) \leq d$ .
- Wenn mindestens eine Klausel nicht erfüllt ist, so muss eines der Literale geflipt werden, um diese zu erfüllen.
- Flipt man „richtiges“ Literal, so ist man bzgl. des Hamming-Abstandes um 1 näher an  $b$ .
- LocalSearch probiert alle möglichen Flips und schaut, wo es auf eine Lösung kommt. □

### Algorithmus 15 Hablierender deterministischer Ansatz

- 1: **return** LocalSearch( $\Phi, 0 \dots 0, \frac{n}{2}$ )  $\vee$  LocalSearch( $\Phi, 1 \dots 1, \frac{n}{2}$ )

⇒ Laufzeit hier:  $\mathcal{O}^*(k^{n/2}) \Rightarrow$  3-SAT:  $\mathcal{O}^*(1.73^n) \Rightarrow$  4-SAT:  $\mathcal{O}^*(2^n)$

### Algorithmus 16 RandomLocalSearch RLS

- 1: **Input:** Instanz  $\Phi$ , Rekursionsverhältnis  $\delta \in ]0, \frac{1}{2}]$
- 2: ▷ Wahl von  $a$  ist preprocessing/kann beliebig intelligent sein
- 3: Wähle zufällige Belegung  $a \in \{0, 1\}^n$  für  $\Phi$
- 4: **return** LocalSearch( $\Phi, a, \delta \cdot n$ )

**Lemma** (Erfolgswahrscheinlichkeit von RLS mit  $\delta \in ]0, \frac{1}{2}]$ ).

$$Pr_{succ} \geq \frac{1}{2^n} \cdot \sum_{i=0}^{\delta n} \binom{n}{i} \geq \dots \geq \frac{1}{n+1} \left( \frac{1}{2} \cdot \left( \frac{1}{\delta} \right)^\delta \cdot \left( \frac{1}{1-\delta} \right)^{1-\delta} \right)^n$$

**Beweis:**

- Falls es erf. Bel.  $b$  gibt, ist der Hamming Abstand einer zufälligen Belegung  $a$  zu  $b$  binomial verteilt:  $|\mathcal{H}(a, \delta n)| = \sum_{i=0}^{\delta n} \binom{n}{i}$
- Gesamtanzahl Belegungen ist  $2^n \Rightarrow$  Erste Abschätzung
- $\geq$ , weil es mehr als eine erf. Bel. geben könnte, sonst =
- Rest des Beweises basiert auf Ausrechnen durch Anwendung der Abschätzung aus der parametrisierten Komplexität □
- Erwartete Wiederholungen von RLS:  $E[\#\text{RLS}] = 1/Pr_{succ}$
- Erwartete Laufzeit:  $E[\#\text{RLS}] \cdot \text{Laufzeit}(\text{LocalSearch})$

k	2	3	4	5	→ ∞
bestes (min) $\delta$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$	→ 0
Laufzeit $\mathcal{O}^*$	$(4/3)^n$	$(3/2)^n$	$(8/5)^n$	$(5/3)^n$	→ $2^n$



# Dynamische Programmierung

## Das TSP

**Gegeben:** Vollständiger Graph auf  $n$  Knoten, Abstandskosten  $c$

**Gesucht:** Hamilton-Kreis  $K^*$  mit möglichst niedrigen Kosten.

**Holzhammer:** Probiere  $(n-1)!$  viele Permutationen.

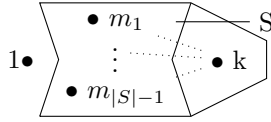
→ Laufzeit:  $\mathcal{O}^*((n-1)!) = \mathcal{O}^*\left(\left(\frac{n}{e}\right)^n\right)$

**Besser:** Nutze dynamische Programmierung:

$c(S, k)$  = Minimale Kosten eines Hamilton-Pfads von Knoten 1 zu Knoten  $k$  auf dem durch die Knoten  $\{1\} \cup S$  induzierten Teilgraph,  $S \subseteq \{2, \dots, n\}, k \in S$

**Idee:** Betrachte alle Wege und merke die Ergebnisse:

$c(S, k) =$   
 $c(S \setminus \{k\}, m_1) + c(m_1, k) \vee$   
 $c(S \setminus \{k\}, m_2) + c(m_2, k) \vee \dots$



Berechne dies mittels folgender rekursiver Beziehung:

$$c(S, k) = \begin{cases} c(1, k) & \text{für } S = \{k\} \\ \min\{c(S \setminus \{k\}, m) + c(m, k) \mid m \in S \setminus \{k\}\} & \text{für } |S| > 1 \end{cases}$$

Bellmansche Optimalitätsbeziehung

- $c(K^*) = \min\{c(\{2, \dots, n\}, m) + c(m, 1) \mid k \in \{2, \dots, n\}\}$
- Berechnung von  $c(S, k)$  ist, wenn alle  $C(S \setminus \{k\}, m)$  bekannt sind, mit  $|S| - 1$  Additionen und  $|S| - 2$  Vergleichen möglich.
- Insgesamt:  $2|S| - 3$  Schritte.
- Zudem gibt es  $\binom{n-1}{l}$   $l$ -elementige Mengen  $S$  mit jeweils 1 Möglichkeiten für  $k$ .
- Dazu  $2n - 3$  Schritte zur Berechnung von  $c(K^*)$ .
- **Laufzeit:**

$$\sum_{l=2}^{n-1} \left[ (2l-3) \cdot l \cdot \binom{n-1}{l} \right] + 2n - 3 \leq 2n^2 2^n + 2n - 3 = \mathcal{O}^*(2^n)$$

→ **Bellmansche Optimalitätsbeziehung:** Optimale Lösung lässt sich aus optimalen Teillösungen zusammensetzen.

→ Beim TSP lassen sich optimale Hamiltonpfade durch Teillösungen ohne den aktuellen Endknoten und den Knoten des verbliebenen direkten Weges berechnen.

→ **Dynamische Programmierung:** Durch speichern berechneter Teilergebnisse, können gleichartige Teilprobleme darauf zurückgreifen und müssen erneut berechnen. Man tauscht Laufzeit durch Speicherplatz

## Zeitplanung

**Gegeben:**  $n$  Jobs  $J_i$ , Zeiten  $a_1, \dots, a_n$ , Kosten  $c_1(t), \dots, c_n(t)$

**Gesucht:** Reihenfolge (Schedule)  $\pi$  mit minimalen Kosten

→ Maschine muss  $J_i$  vollständig erledigen (steht nie still)

→ Reihenfolge kann durch Permutation  $\pi$  beschrieben werden

→ Abschlusszeit  $t_{\pi(i)}$  von  $J_{\pi(i)}$ :  $t_{\pi(i)} = \sum_{j=1}^i a_{\pi(j)}$

→ Kosten:  $\sum_{i=1}^n c_{\pi(i)}(t_{\pi(i)})$

→ Für Teilmenge  $S \subseteq \{J_1, \dots, J_n\}$  sei  $t_S = \sum_{J_i \in S} a_i$ .

→  $\text{opt}(S)$  bezeichne Kosten eines minimalen Fahrplans in  $[0, t_S]$ .

Berechnung von  $\text{opt}(S)$  mittels rekursiver Beziehung:

$$\text{opt}(S) = \begin{cases} c_i(a_i) & \text{für } S = \{J_i\} \\ \min\{\text{opt}(S \setminus \{J_i\}) + c_i(t_S) \mid J_i \in S\} & \text{für } |S| > 1 \end{cases}$$

Bellmansche Optimalitätsbeziehung

→ **Laufzeit** von  $\text{opt}(\{J_1, \dots, J_n\})$ :  $\sum_{i=1}^{n-1} i \cdot \binom{n}{i} = \mathcal{O}^*(2^n)$

→ Auch hier wieder die Bellmansche Optimalitätsbeziehung

→ Optimale Lösung lässt sich aus einzelnen Jobs bilden.

→ Rekonstruktion: Optimale Reihenfolge kann dadurch bestimmt werden, dass man sich bei jedem  $\text{opt}(S)$  merkt, durch welche Sublösung der optimale Wert erreicht wurde.

## Matrixmultiplikation

**Gegeben:**  $n$  Matrizen  $M_i$  mit  $r_{i-1}$  Zeilen und  $r_i$  Spalten.

**Gesucht:** Klammerung damit Multiplikationen minimiert wird  
 Berechnen 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.

Für eine Teilfolge  $M_i, \dots, M_j$  sei  $m_{i,j}$  die minimale Anzahl an Zahlenmultiplikationen.

Berechnung von  $m_{i,j}$  mittels folgender rekursiver Beziehung:

$$m_{i,j} = \begin{cases} 0 & , i = j \\ \min\{m_{i,k} + m_{k+1,j} + r_{i-1} \cdot r_k \cdot r_j \mid i \leq k < j\} & , i < j \end{cases}$$

Bellmansche Optimalitätsbeziehung

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

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

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

→ **Beispiel:** 4 Matrizen:  $r_0 = 10, r_1 = 1, r_2 = 50, r_3 = 20, r_4 = 20$   
 $\Rightarrow (M_1 \cdot M_2) \cdot (M_3 \cdot M_4)$  benötigt 30500 Mults, aber  $((M_1 \cdot M_2) \cdot M_3) \cdot M_4$  nur 14200 und  $M_1 \cdot ((M_2 \cdot M_3) \cdot M_4)$  nur 1600.

## Subsetsum in der dynamischen Programmierung:

Problem: Finde Summe der  $a_1, \dots, a_n$ , welche  $S$  ergeben.

Basisfall:  $f(1, S) = \begin{cases} 1 & , S = a_1 \text{ oder } S = 0 \\ 0 & , \text{sonst} \end{cases}$

Rekursiver Aufruf:  $f(i, S) = \max\{f(i-1, S), f(i-1, S - a_i)\}$

Laufzeit:  $\mathcal{O}(n \cdot S)$ , Zusammenhang Parametrisierte Komplexität:  $S$  ist der wichtigste Parameter, die Bitkomplexität impliziert Exponentialität

# Klausurfragen:

## Kapitel 1: Tiefensuche

1. Erklären Sie DFS für ungerichtete Graphen. Warum funktioniert dies?
2. Welche Eigenschaften haben Rückkanten im Tiefensuchbaum? Wieso?
3. Was kann mit tiefer Suche berechnet werden? Was ist eine nicht triviale Eigenschaft, die auf ungerichteten Graphen mit DFS berechnet wird?
4. Wann ist ein Graph (Teilgraph) maximal zusammenhängend? Was ist eine max ZHK?
5. Definieren Sie Artikulationspunkt und erklären Sie, wie man einen solchen erkennt. Warum funktioniert dies?
6. Definieren Sie zweifache Zusammenhangskomponente und erläutern Sie einen Ansatz zum Erkennen.
7. Erklären Sie die tief-Funktion und ihre Zusammenhänge zur  $dfnr$ .
8. Was ist ein Superstrukturgraph? Wieso ist dieser wichtig? Welche Eigenschaften hat dieser?
9. Wie findet man über Tiefensuche 2fache ZHK? Wo kann ein AP im Keller liegen?
10. Warum funktioniert DFS-2ZHK? Laufzeit und warum?
11. Wie müsste man DFS-2ZHK verändern, wäre der Ausgangsgraph nicht zusammenhängend
12. Warum braucht die Wurzel eine Sonderbehandlung? (Argumentation für AP benötigt hierarchisch höhere Knoten  $\rightarrow$  Argumentation gilt nicht für Wurzel. Zudem: Hat Wurzel nur 1 Kind ist Bedingung erfüllt, Wurzel aber nicht AP  $\zeta$ )
13. Was kann man bei gerichtetem Graphen berechnen? (starke ZHK)
14. Welche Kantentypen gibt es im Tiefensuchwald für Digraphen?
15. Was gilt für die Crosskanten? ( $(u \rightarrow v) \Rightarrow dfnr(u) > dfnr(v) \rightarrow$  können ignoriert werden)
16. Definieren Sie starke Zusammenhangskomponente und stellen Sie eine Verbindung zum Superstrukturgraphen her. Erklären sie kurz Äquivalenzrelation und Äquivalenzklassen
17. Beschreiben Sie die Tiefensuche zum finden starker ZHK. Warum funktioniert dies? Wie ist tief definiert? Welche Eigenschaft aus dem Superstrukturgraph ist wichtig? (DAG) Warum Tiefensuche und nicht Breitesuche? (Letzte SZHK voll durchsuchen und verlassen, geht mit BFS ned)
18. Wieso ist die Laufzeit linear in der Eingabelänge?
19. Welche Anwendung hatten wir dafür in der Übung?
20. Algorithmen DFS-2ZHK und DFS-SZHK an Beispiel durchführen.

## Kapitel 2: Flüsse in Netzwerken

1. Definieren Sie ein kombinatorisches Optimierungsproblem.
2. Definiere Fluss und Flussproblem. Nennen sie Eigenschaften von Flüssen
3. Definiere Wert eines Flusses über Senke / Kante. Erläutern sie weitere Eigenschaften von Flüssen
4. Was ist Kapazität eines Schnittes?(Nur was man rüberbringt). Welchen Zusammenhang wurde für s-t-Schnitte definiert?
5. Definieren Sie maximalen Fluss.
6. Erklären Sie das Konzept der erweiternden Wege und des Residualgraphen. Wieso kann es zu einer exponentiellen Laufzeit kommen? Wie kann es zu unendlicher Laufzeit kommen?
7. Wieso ist das Ausnutzen von Vor- und Rückkanten im Residualgraphen wichtig?
8. Beweisen Sie den Satz über erweiternde Wege.
9. Was besagt das Integrality-Theorem?
10. Erklären Sie den Algorithmus von Dinic. Erläutern Sie die Laufzeit.
11. Erklären Sie den Zusammenhang von geschichtetem Netzwerk und Sperrfluss.

## Kapitel 3: Parametrisierte Komplexität

1. Definieren Sie VC und geben Sie einen exakten Algorithmus an.
2. Definieren Sie parametrisierte Komplexität.
3. Geben sie für ein Problem  $x$  den Parameter an (Sat z.b. Länge der Klausel, VC Anzahl Knoten eines VC)
4. Wann ist ein Algo parametrisiert?
5. Erklären Sie den Algorithmus von Buss & Goldsmith. Welche Laufzeit hat dieser (Laufzeit in Parameter / Eingabegröße)? Wäre statt dem Additiven Zusammenhang auch ein multiplikativer möglich? (Ja)
6. Wann kann es kein VC der Größe  $k$  geben? (Angabe und Erklärung der 2 Ideen für B & G)
7. Was bedeutet Kernalisierung?
8. Wie kann ein parametrisierter Algorithmus in einen exakten (Optimierungs-)Algorithmus umwandeln?
9. Mit welchem Ansatz kann man Laufzeiten in  $\mathcal{O}^*$  bestimmen?
10. Wie kann Kombination von Algos besser sein, als die einzelnen? (Gemeinsamer Worst Case ist kleiner, Bild malen)

## Kapitel 4: Das Erfüllbarkeitsproblem

1. Definieren Sie SAT.
2. Erklären Sie den polynomiellen Algorithmus für 2SAT.
3. Wie interpretiert man einen Pfad im resultierende Graphen? (Implikationskette)
4. Erklären Sie den Algorithmus von Monien-Speckenmayer. Analysieren Sie die Laufzeit.
5. Wieso kann man (bei MS) ein Literal auf False setzen?
6. Erklären Sie den Zusammenhang von LocalSearch und Hamming-Ball.
7. Wie kann man LocalSearch randomisieren.
8. Was bedeutet Preprocessing in diesem Kontext?

## Kapitel 5: Dynamische Programmierung

1. Erklären Sie das TSP-Problem und den Nutzen der dynamischen Programmierung.
2. Erklären Sie das Scheduling-Problem und den Nutzen der dynamischen Programmierung.
3. Erklären Sie Matrixmultiplikation und den Nutzen der dynamischen Programmierung.
4. Erklären Sie ein dynamisches Verfahren für Subsetsum.
5. Wie und wieso funktionieren die Algorithmen?
6. Welche Laufzeiten haben die Algorithmen?
7. Woher kommen die exponentiellen Laufzeiten?
8. Was besagt die Bellmansche Optimalitätsbedingung?