Effiziente kombinatorische Algorithmen - Zusammenfassung

Autor: Linda Schneider

Depth-First-Search

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

Algorithmus 1 Systematisches Durchsuchen von G

```
1: Starte in beliebigen Knoten s ∈ V.
2: S = {s}.
3: Markiere alle Kanten e ∈ E als unbenutzt.
4: while ∃ unbenutzte Kante e von u ∈ S nach v do
5:     Wähle Knoten u und unbenutzte Kante e.
6:     S = S ∪ {v}.
7:     Markiere e als benutzt.
```

→ Gestartet in s gilt für die Knoten in $\mathcal{S}$ nach Terminierung:

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

Beweis:
„⇒ “ : Offensichtlich nach Konstruktion.
„⇐“ : Annahme: $v \in V$ und v ist von s aus erreichbar.
⇒ Zu zeigen: $v \in \mathcal{S}$
→ 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: $i = 0,\ v_0 = s \in \mathcal{S}$
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: Kodierung durch Inzidenzliste ⇒ Speicherplatz: $\mathcal{O}(|V| + |E|)$.
- Darstellung $\mathcal{S}$: Array mit $\mathcal{S}[v] \in \{besucht, unbesucht\}$.
- Reihenfolge von $u \in \mathcal{S}$: Auswahl mittels Keller und Inzidenzliste in Verbindung mit den Knotenmarkierungen.

Algorithmus 2 DFS

```
1: Starte in beliebige Knoten s ∈ V.
2: for v ∈ V do
3:     if v ≠ s then S[v] = nichtbesucht
4:     else S[v] = besucht
5: Lege s auf Keller.
6: while Keller nicht leer do
7:     Nimm obersten Knoten u von Keller.
8:     if Es gibt unbenutzte Kante u -e-> v then
9:         Markiere e als benutzt.
10:        Lege u zurück auf den Keller.
11:        if S[v] = nichtbesucht then
12:            S[v] = besucht
13:            Lege v auf Keller.
```

⇒ Laufzeit: $\mathcal{O}(|V| + |E|)$

Definition (Maximale Zusammenhangskomponente)**.**
Ein Teilgraph H von G heißt maximale ZHK, falls H zhgd ist und G keinen H echt enthaltenen zhgd Graphen enthält.

Beweise: Knoten in G haben geraden Grad ⇒ ∃ Eulerkreis
→ Suche Greedy Kreis im Graphen und lösche dessen Kanten
→ Knoten haben weiterhin geraden Grad!
⇒ Mache das bis alle Kanten gelöscht sind und stöpsel Kreis zusammen

---

Berechnung der 2fachen ZHK

→ G sei im Folgenden immer ein zhgd Graph.
→ Reihenfolge des Besuchs der Knoten wird in dfnr gespeichert.
→ Benutzte Kanten werden ohne Richtung in $T \subseteq E$ gespeichert. (für Tiefensuchbaum)
→ In B werden unbenutzte Rückkanten gespeichert.

Algorithmus 3 DFS_u: Tiefensuche in ungerichteten Graphen

```
1: procedure S(v)
2:     dfnr[v] = zaehler, zaehler++ ▷ Setze dfnr des Knotens
3:     Markiere v als besucht.
4:     for ∀ Knoten w ∈ L[v] do ▷ Knoten aus Inzidenzliste
5:         if w = nichtbesucht then
6:             T = T ∪ (v, w) ▷ Füge Kante zu Tiefenbaum
7:             S(w) ▷ Suche von w aus
8:
9: T = ∅, zaehler = 1
10: for ∀v ∈ V do ▷ Initialisierung des Graphen
11:     Markiere v als nichtbesucht.
12: Wähle r ∈ V. ▷ Wähle Wurzel
13: S(r). ▷ Starte Suche von Wurzel aus
14: return T, r, dfnr
```

→ Korrektheit des Algo kann über Rückkanten geprüft werden.

Lemma (Korrektheit des Algorithmus)**.**
DFS_u berechnet in $\mathcal{O}(|V| + |E|)$ die Funktion $dfnr$ und eine Zerlegung von E in T und $B = E \setminus T$ mit

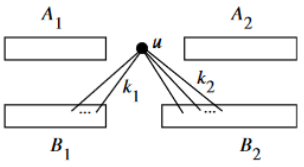
1. Der gerichtete Graph $H = (V, T)$ ist ein Baum mit Wurzel r und $dfnr[r] = 1$.
2. Ist $(v \xrightarrow{e} w) \in B$, so ist v in H Nachfolger von w.

→ Das die erste Aussage gilt ist offensichtlich.
→ Wenn die zweite Aussage falsch wäre, wäre v bis und beim Durchlauf von $L[w]$ nicht betrachtet worden, was per Konstruktion unmöglich ist.

Definition (Artikulationspunkt)**.**
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.)

Definition (Zweifach Zusammenhängend)**.**
G heißt zweifach zusammenhängend, wenn G keinen AP besitzt.

Beispiel. (Reguläre, bipartite Graphen)
Ist G ein zusammenhängender, regulärer, bipartiter Graph von Grad k, so muss G aufgrund der Regularität zweifach Zusammenhängend sein. 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.



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.

Ansatz 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 von v nach u erreichbar ist mit höchstens einer Rückkante.

Lemma (Hinreichende Bedingung)**.**
Gegeben sind B, T, R von $DFS_u(G)$. Falls eine Baumkante $(u \to v) \in T$ existiert mit $dfnr[u] > 1$ und $tief(v) \geq dfnr[u]$, so ist u ein AP.

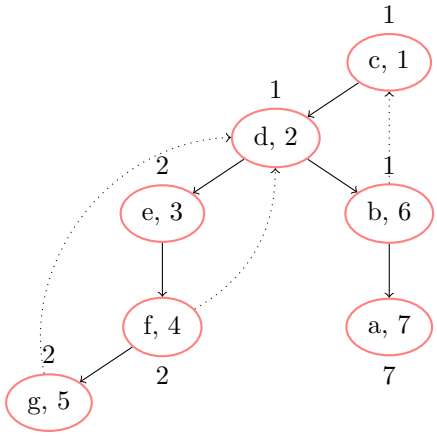
Beweis:
→ Sei $S \setminus \{u\}$ Menge der Knoten in $H = (V, T)$ auf dem Weg von r zu u.
→ V_v sei Menge der Knoten die zu dem Teilbaum gehören, der an v hängt.
→ Aus Lemma 1: Wegen Ast-Eigenschaft gibt es keine Kanten, die zwischen V_v und $V \setminus \{\mathcal{S} \cup \{u\} \cup V_v\}$ verlaufen.
⇒ 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 $w \in \mathcal{S}$ geben.
⇒ $tief(v) \leq dfnr[w] < dfnr[u]$ ↯

□

Lemma (Notwendige Bedingung)**.**
Gegeben sind B, T, R von $DFS_u(G)$. Sei u ein AP von G mit $u \neq r$. Dann gibt es eine Baumkante $(u \to v) \in T$ mit $tief(v) \geq dfnr[u]$.

Beweis:
→ $G \setminus \{u\}$ besteht aus ZHK $G_1, \ldots, G_m,\ m \geq 2$.
⇒ Jeder Weg zwischen den ZHK führt über u.
→ Sei oBdA. $r \in V(G_1)$.
⇒ Jede Suche von DFS_u gelangt über Baumkanten von $r \to u$.
⇒ Sei $(u \to v)$ 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 $tief(v) \geq dfnr[u]$.

□



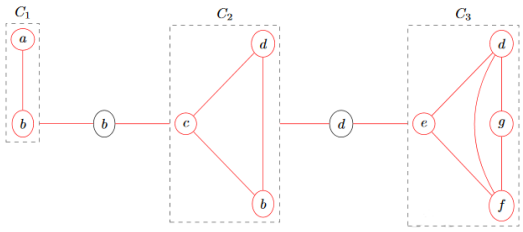
→ Bemerke: Die Wurzel ist genau dann AP, wenn Ausgangsgrad der Wurzel des Tiefenbaumes ≥ 2 ist.

Definition (Superstrukturgraph)**.**
Sei G ein zhgd. Graph. Seien u_i APs und C_j zweifache ZHK von G. Dann heißt der Graph $\bar{G}$ mit

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

Superstrukturgraph von G.

⇒ 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.



Es gibt 2 Fälle zu unterscheiden:
- Startknoten r ist kein AP:
  → DFS_u gelangt von r über Baumkanten zu einer ersten 2-fachen ZHK C, die Blatt von $\bar{G}$ ist.
  → C wird über AP u, gefolgt von $(u \to v) \in T$ betreten und über $v \leftarrow u$ rückwärts verlassen.
  ⇒ Ist beim verlassen von C der Wert $tief(v)$ berechnet, so ist u AP.
  → Bemerkung: Sobald C betreten wird können die Berechnungen von DFS_u als Berechnungen auf eigenem Graphen gestartet in u aufgefasst werden.
  ⇒ C enthält nur eine Baumkante $u \to v$. D.h. in die Komponente läuft nur eine Kante vom AP u hinein.
  ⇒ Erkennen von ZHK: Gebe alle Knoten oberhalb von u aus *Besichtigte-Knoten* aus mit u, aber u bleibt auf dem Keller. Für *Besichtigte-Kanten* das gleiche, aber $u \to v$ wird mit vom Keller genommen.
  → Anschließend gleiches Verfahren ab Anfangsknoten u.
- Startknoten r ist ein AP:
  → Alle 2-fachen ZHK ohne r werden wie vorher beschrieben erkannt.
  → Wenn zu Baumkante $(r \to v) \in T$ die Tiefensuche von v aus beendet ist und noch eine zweite, nicht besuchte Kante $(r \to u)$ existiert, so ist r AP.
  ⇒ Gib jeweils alle Knoten bis zu v und alle Kanten bis zu $r \to v$ aus für 2-fache ZHK.
  → Wiederhole dies, bis alle Kanten besucht worden sind.

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

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

- Falls v kein Blatt ist

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

Algorithmus 4 DFS-2ZK

```
1:  procedure S(v)
2:      tief(v) = dfnr[v] = zaehler, zaehler++
3:      Markiere v als besucht
4:      for ∀ w ∈ L[v] do                    ▷ Knoten in Inzidenzliste
5:          if w = nichtbesucht then
6:              T = T ∪ (v → w)
7:              vater[w] = v                 ▷ Hierarchie merken
8:              Lege w auf Besichtigte-Knoten
9:              (v → w) auf Besichtigte-Kanten
10:             S(w)                         ▷ Suche in Knoten weiter
11:             if tief(w) ≥ dfnr[v] und v ≠ r then
12:                 Gib 2-fache ZHK bis v aus und leere Stack\v
13:             tief(v) = min{tief(v), tief(w)}
14:         else if w ≠ vater[v] und dfnr[v] > dfnr[w] then
        ▷ Wir haben eine Rückkante von v nach w gefunden
15:             Lege (v → w) auf Besichtigte-Kanten
16:             tief(v) = min{tief(v), dfnr[w]}
17:
18: T = ∅, zaehler = 1
19: for ∀v ∈ V do
20:     Markiere v als nicht-besucht
21: Wähle r ∈ V.
22: S(r)
23: while Besichtigte-Kanten nicht leer do
24:     Gib alle Kanten auf Besichtigte-Kanten bis (r → v) ∈ T
25:     Gib alle Knoten von Besichtigte-Knoten bis r aus
```

→ Laufzeit: $\mathcal{O}(|V| + |E|)$.

⇒ 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

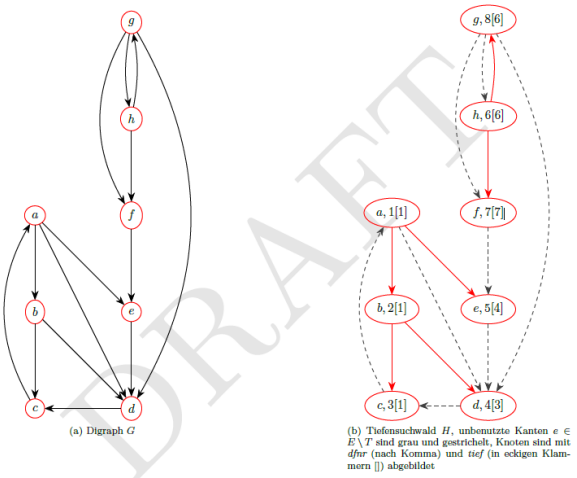
Definition (Brücke)**.**
Eine Kante $e \in E$ heißt Brücke, wenn $G' = (V, E \setminus \{e\})$ mehr maximale Zusammenhangskomponenten hat als G.

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:
while ∃ *nicht-besuchte Knoten* $r \in V$ *do: suche(r)*
⇒ DFS berechnet zu G den Tiefensuchwald H:



(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

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.
2. **Vorwärtskanten:** Kanten $(u \to v) \in E \setminus T$ mit Weg in H.
3. **Rückkanten:** Kanten $(u \to v) \in E \setminus T$ mit v → … → u ist Weg in H. (⇒ Bsp: c → a, g → h)
4. **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.
   ⇒ Ist u → v eine Crosskante, so gilt: $dfnr[u] > dfnr[v]$.
   ⇒ Bsp: d → c, E → d, g → f

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

Beispiel (Ist die Sprache endlich?)**.**
Suche in Automat starke ZHK. Hat diese ≥ 2 Zustände, so muss Endzustand enthalten sein, da wir sonst endlose Zyklen haben können.

Beispiel (Brückenlos und starke ZHK)**.**
Aufgabe: Mache aus brückenlosen, ungerichteten, zusammenhängendem Graphen eine starke ZHK
→ Führe Tiefensuche durch und merke Baum- und Rückkanten
→ Da G brückenlos: $\nexists u \in V \setminus \{r\} : dfnr[u] = tief(u)$
→ Auch Wurzel muss also über Rückkante erreichbar sein!
⇒ Erstelle gerichteten Kreis für Kriterium der starke ZHK

Definition (Superstrukturgraph)**.**
Seien $C_1, \ldots, C_k$ starke Zusammenhangskomponenten von G. Dann heißt der Digraph $\tilde{G}$ mit $V(\tilde{G}) = \{C_1, \ldots, C_k\}$ und $E(\tilde{G}) = \{C_i \to C_j \mid i \neq j,$ falls ∃ Kanten zwischen i und j$\}$ der Superstrukturgraph von G.

⇒ **Fakt:** $\tilde{G}$ ist ein kreisfreier Graph (DAG)!

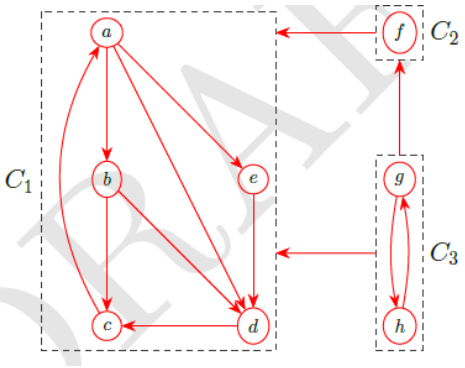


Abb. 2: Superstrukturgraph zu Abbildung 1

Vorgehen von DFS im Superstrukturgraphen:
→ Start in Knoten r in ZHK C_1: Wird C_1 verlassen, ehe C_1 vollständig durchsucht ist ⇔ $d^+_{\tilde{G}}(C_1) \geq 1$.
→ In diesem Fall gelangt DFS zur nächsten starken ZHK, bis eine ZHK C_k mit $d^+_{\tilde{G}}(C_k) = 0$ erreicht ist.
⇒ Diese muss es geben, da $\tilde{G}$ ein DAG ist.
→ C_k wird vollständig durchsucht, bevor es rückwärts wieder verlassen wird.

Identifizieren von ZHK mittels tief:
→ 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!
⇒ Induktives Berechnungsschema für tief im Tiefensuchwald:
1. $v \in V : d^+_{\tilde{G}}(v) = 0$:
   $tief(v) = \min(\{dfnr[v]\} \cup \{dfnr[u] \mid v \to u$ Rück- oder Crosskante, u in gleicher ZHK wie $v\})$
2. $v \in V : d^+_{\tilde{G}}(v) \geq 1$:
   $tief(v) = \min(\{dfnr[v]\} \cup \{dfnr[u] \mid v \to u$ Rück- oder Crosskante, u in gleicher ZHK wie $v\} \cup$ $\{tief(u) \mid (v \to u) \in T\})$

Algorithmus 5 DFS-SZK

```
1:  procedure S(v)
2:      tief(v) = dfnr[v] = zaehler,  zaehler++
3:      Markiere v als besucht.
4:      Lege v auf Keller K, auf_Keller[v] = true.
5:      for ∀ w ∈ L[v] do                ▷ Knoten in Inzidenzliste
6:          if w = nicht_besucht then
7:              S(w), tief(v) = min{tief(v), tief(w)}
8:          else if w = besucht then
9:              tief(v) = min{tief(v), dfnr[w]}
10:     if tief(v) = dfnr[v] then
11:         while auf_Keller[v] do
12:             Nimm obersten Knoten x vom Keller K.
13:             Gebe x aus.
14:             auf_Keller[x] = false
15:         „Ende der SZK “.
16:
17: zaehler = 1,  K = ∅.
18: for v ∈ V do
19:     Markiere v als nicht-besucht.
20:     Setze auf_Keller[v] = false.
21: while ∃ nicht besuchter Knoten r ∈ 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 $z \in V(C)$, die nicht über t auf K liegen, haben diese Eigenschaft.
   → Mindestens ein solcher Knoten z' ist über Rück- oder Crosskanten erreichbar.
   ⇒ $x \to z'$ von x oberhalb von t auf K.
   → Da $x, z', t \in V(C)$ sind, gilt
   $$tief(t) \leq tief(x) \leq dfnr[z'] < dfnr[t] \quad \lightning$$

□

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 Π 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 $f : S(I) \to \mathbb{N}$
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.

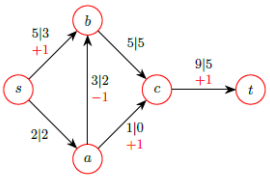
Definition (Flussproblem).
Ein Netzwerk N ist ein Digraph G mit einer Kapazitätsfunktion c und zwei besonderen Knoten s (Quelle) und t (Senke). Wir schreiben $N = (G, s, t, c)$.

→ **Fluss:** Wert der über Kante fließt. Das Gleiche muss auch zurück fließen können (Konservierungsgesetz)
→ **Wert des Flusses:** Kapazität, die darüber fließt, Differenz muss noch fließen können.
→ **s,t-Schnitt:** Partition der Knoten, für maximalen Fluss
⇒ Kann an Quelle und Senke bestimmt werden.
⇒ Jeder s,t-Schnitt ist kleiner gleich der Summe der Kapazitäten der geschnittenen Knoten

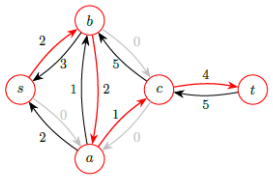
Definition (Erweiternder Weg).
Ein Weg von s nach t über Kanten aus $E \cup \overleftarrow{E}$ heißt erweiternder Weg p bezüglich f, falls

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

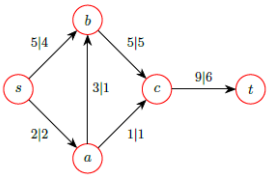
→ Verbesserung durch erweiternden Weg: Finde Weg p, suche schmalste Stelle und verbessere Weg um diesen Wert!



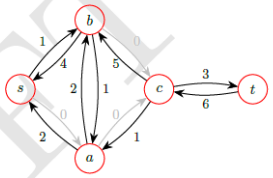
(a) Netzwerk mit einem vorhandenem Fluss und Erweiterungsmöglichkeit



(b) Residualgraph mit erweiterndem Weg

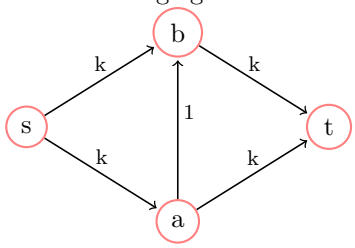


(a) Netzwerk mit maximalem Fluss



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

→ Bemerke: Bei reellen Kantenkapazitäten kann es passieren, dass es unendliche viele erweiternde Wege gibt und man somit nie auf den maximalen Fluss kommt.
→ Bemerke: Folgendes Netzwerk hat eine exponentielle Laufzeit, da 2k erweiternde Wege gefunden werden können:



→ Bemerke: 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 ⇔ 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 echt verbesserbar und somit war f nicht maximal.
„⇐ “: Angenommen ∄ erweiternden Weg bezüglich f.
→ Aufteilen der Knotenmenge in zwei Partitionen:
$X = \{s\} \cup \{v \in V \mid v \neq t \text{ ist von } s \text{ aus im Residualgr. erreichbar}\}$
$\bar{X} = V \setminus X$.
→ Ist $e = (u \to v)$ Kante in G aber nicht im Residualgraphen und $u \in X,\ v \in \bar{X}$, dann ist $c(e) = f(e)$.
→ Ist e eine Kante in G aber $\overleftarrow{e}$ nicht im Residualgraphen, dann ist $f(e) = 0$.
⇒ Nettofluss von X nach $\bar{X}$ ist daher die Summe der einzelnen Flüsse von X nach $\bar{X}$.
→ Dies beweist auch das Theorem und somit ist f maximal.

□

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

Algorithmus von Dinic

→ Laufzeit: $\mathcal{O}(|V|^2|E|)$

→ Restkapazität $\tilde{c}(e) = \begin{cases} c(e) - f(e) & falls\ e \in E \\ f(e) & falls\ e \in \overleftarrow{E} \end{cases}$

→ Kanten mit $\tilde{c}(e) > 0$ heißen nützliche Kanten.

⇒ Erweiternde Wege bzgl. f enthalten nur nützliche Kanten!

→ f heißt **Sperrfluss**, falls mindestens eine Kante im Schichtgraph voll ausgelastet ist.

→ $\tilde{\delta}(v)$ = Anzahl Kanten eines kürzesten Weges von s zu v 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

- $\tilde{V} = \{v \in V \mid v$ ist auf kürzestem erweiterndem Weg von s nach t, $\tilde{\delta}(v) \leq \tilde{\delta}(t)\}$
- $\tilde{E} = \{e = (u, v) \in E \cup \overleftarrow{E} \mid e$ 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.

→ $(\tilde{V}, \tilde{E})$ 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

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

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

→ Gilt wegen Erfülltheit des Konservierungsgesetzes.

Algorithmisches Schema zur Berechnung eines max. Flusses

Algorithmus 6 Dinic - Algorithmisches Schema

Input: Netzwerk N = (G,s,t,c)
Output: Maximaler Fluss f auf N

```
1: Setze Startfluss f(e) = 0  ∀e ∈ E.
2: while Ṽ ≠ ∅ do
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: return f
```

Satz (Laufzeit)**.**
Die while-Schleife wird höchstens $|V| - 1$ mal durchlaufen.

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

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

→ Sei $p = (s \to u_1 \to \cdots \to u_k = t)$ Weg mit $\delta'(u_i) = i$.

→ Zeige durch Induktion: $\delta'(u_i) \geq \tilde{\delta}(u_i)$

→ Annahme: $\delta'(u_i) < \tilde{\delta}(u_i)$

⇒ ∃ nützliche Kante bezüglich dem leeren Fluss.

Fall 1: $u_i \to u_{i+1}$ **ist nützliche Kante**

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

Fall 2: $u_i \to u_{i+1}$ **ist keine nützliche Kante**

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

→ Die Kante $u_{i+1} \to u_i$ lag in $\tilde{N}(f)$ auf dem kürzesten Weg von s nach t.

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

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

Noch zu zeigen: $\delta'(t) > \tilde{\delta}(t)$

→ Unter Annahme von Gleichheit wären alle Knoten in der nächsten Iteration gleich und somit wäre f kein Sperrfluss gewesen!

□

Wir starten mit leerem Fluss auf $\tilde{N}(f)$ und der Graph habe Durchmesser k.

Algorithmus 7 Sperrfluss

```
1: while t ∉ V_k do                    ▷ Solange Ziel noch nicht in Graph
2:     v = t, a = ∞
3:     for i = k, ..., 1 do             ▷ Berechne erweiternden Weg
4:         Wähle e = (u, v) in G
5:         a = min{a, c̃(e)},  v = u
6:     Aktualisiere in Ñ(f) den Fluss entlang des konstruierten
       Weges um a und reduziere die Kapazitäten um a.
7:     Entferne unnütze Knoten und Kanten.
8: return f
```

→ Laufzeit: $\mathcal{O}(|V||E|)$

⇒ 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 „gesehen " 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.

→ Entscheidungs- und Optimierungsproblem sind polynomiell verknüpft, aber VC in NP.

→ Laufzeitberechnung kann über charakteristisches Polynom vereinfacht werden.

→ Alternativ auch Abschätzung über folgenden Formel möglich:

$$\sum_{i=0}^{\delta n} \binom{n}{i} \geq \frac{1}{n+1} \left(\left(\frac{1}{\delta}\right)^{\delta} \cdot \left(\frac{1}{1-\delta}\right)^{1-\delta} \right)^n \qquad \forall \delta \in \left[0, \frac{1}{2}\right]$$

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

Ein exakter Algorithmus für VC

→ Für jeden Knoten v gilt: v ist im VC oder alle seine Nachbarn!

Algorithmisches Schema für MVC

1. Nachbar v von Knoten u mit Grad 1 ist im minimalen VC, dann werden u und v aus Graph entfernt. Wiederhole dies für alle Knoten mit Grad 1.
2. Haben alle Knoten nun Grad 2, so ist der Restgraph eine Vereinigung von Kreisen und jeder zweite Knoten muss ins VC aufgenommen werden.
3. Falls nicht, so gibt es mindestens einen Knoten v mit Grad ≥ 3. Wähle einen solchen Knoten v:
   (a) Bestimme die Nachbarn $u_1, \ldots, u_k$ von v, da entweder v oder alle seine Nachbarn im VC sind.
   (b) Starte rekursiven Aufruf zur Bestimmung eines min. VC C_1 von $G \setminus \{u\}$
   ⇒ Gesamtes Cover: $C_1 \cup \{u\}$
   (c) Starte rekursiven Aufruf zur Bestimmung eines min. VC C_1 von $G \setminus \{Nachbarn(u) \cup u\}$
   ⇒ Gesamtes Cover: $C_1 \cup \{Nachbarn(u)\}$
4. Gib kleinstes Gesamtcover aus!

Satz (Laufzeit MVC).
MVC löst VC_{opt} in Zeit $\mathcal{O}^*(1.3803^n)$.

Beweis:
→ 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
⇒ $\lambda^4 = \lambda^3 + 1$ mit NS $\lambda \approx 1.3803$
⇒ 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 in Abhängigkeit von Parametern und Eingabelänge angeben!

Definition (Par-parametrisierter Polynomzeit-Algo).
Sei L ein Entscheidungsproblem. Eine beliebige in Polyzeit berechenbare Funktion $Par : L \rightarrowtail \mathbb{N}$ 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 eine Funktion g und ein Polynom p, sodass die Laufzeit von A für jede Eingabe I in $\mathcal{O}(g(Par(I)) \cdot p(|I|))$ ist.

Existiert dieser, so nennt man L „fixed-parameter tracable “ 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 abhängt (Kernelization).
- Löse I' und übertrage Antwort auf I.

→ Hat G ein VC der Größe k, so müssen alle Knoten aus v mit Grad größer k enthalten sein.
→ Für einen Graphen ohne isolierte Knoten gilt: Wenn alle Knoten Grad ≤ k haben und G VC der Größe m besitzt, so hat G höchstens $m \cdot k$ Kanten.

Algorithmus 8 DivideAndConquerVC(G,k)

Input: Ungerichteter Graph G=(V,E), k
Output: Antwort: $|VC_{opt}| \leq k$?

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

→ Laufzeit: $\mathcal{O}(2^k \cdot (|V| \cdot |E|))$
Algorithmus mit Problemkern-Methode:

Algorithmus 9 Buss & Goldsmith

Input: Ungerichteter Graph G, k
Output: Antwort: $|VC_{opt}| \leq k$?

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

→ Laufzeit ohne Algo. 9: $\mathcal{O}(kn + 2^k k^{2k+2})$
→ Laufzeit mit Algo. 9: $\mathcal{O}(kn + 2^k k^4)$
⇒ Kernelization benötigt $\mathcal{O}(kn)$
⇒ Restliche Laufzeit kommt aus der Anzahl der m-elementigen Teilmengen.

Umwandlung parametrisierter Algorithmen in exakte Algorithmen

→ Sei Π 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.
⇒ Brute-Force-Ansatz hat Laufzeit $\mathcal{O}^*(\sum_{i=0}^{n} \binom{n}{i}) = \mathcal{O}^*(2^n)$.

Algorithmus 10 Exakt

Input: Π, k-parametrisierter Algo A mit Laufzeit $\mathcal{O}(c^k)$, Eingabe I und Lösungsuniversum U
Output: $\Pi_{opt}(I)$

1: Berechne größtes λ mit $c^{\lfloor \lambda n \rfloor} \leq \sum_{i=\lfloor \lambda n \rfloor + 1}^{n} \binom{n}{i}$
2: /* *Berechnet maximalen Schnittpunkt* */
3: **for** $k = 1, \ldots, \lfloor \lambda n \rfloor$ **do**
4: **if** $A(I, k) = True$ **then return** k
5: **for** $\lfloor \lambda n \rfloor + 1, \ldots, n$ **do**
6: **if** BruteForce(Π,I,U,k) = True **then return** k

→ Laufzeit: $\mathcal{O}^*(\sum_{i=1}^{\lfloor \lambda n \rfloor} c^i + \sum_{i=\lfloor \lambda n \rfloor + 1}^{n} \binom{n}{i}) \overset{\lambda > 1/2}{\equiv} \mathcal{O}^*(c^{\lambda n})$
→ Falls $c < 4$ kann für große n ein $\lambda > 1/2$ gewählt werden, so dass der Algo. asymptotisch schneller ist als Brute-Force.
→ Der Exakte DAC_VC-Algo. hat Laufzeit $\mathcal{O}^*(1.70872^n)$.

Das Erfüllbarkeitsproblem - SAT

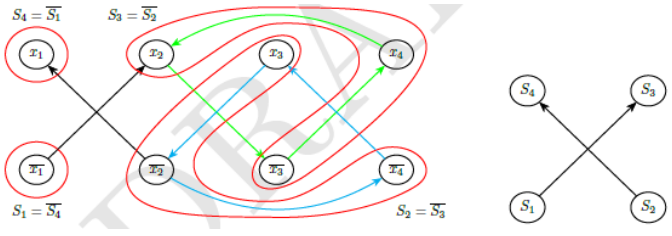
Definition (SAT).
Literale innerhalb von Klauseln haben oder-Verknüpfung, in KNF haben Klauseln und-Verknüpfungen. In SAT wird eine gültige Belegung gesucht. k-SAT beschränkt die Anzahl der Literale in den Klauseln der KNF durch k.

Polynomieller Algorithmus für 2-SAT

Algorithmus 11 Solve2SAT

1: n = Anzahl der Variablen in Φ
2: $V = \{x_1, \ldots, x_n\} \cup \{\bar{x}_1, \ldots, \bar{x}_n\}$, E = $\emptyset$
3: Füge zu E zu allen Klauseln die möglichen Kanten hinzu.
4: Berechne mit DFS_SZK die starken ZK in G_Φ
5: Sind x und $\bar{x}$ in einer ZK, **return** Null
6: **while** ∃ 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: $x_i \vee x_j \Rightarrow (\bar{x}_i \Rightarrow x_j, \bar{x}_j \Rightarrow x_i)$



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_Φ. Anschließend kann S_3 gewählt werden ....
⇒ Erhalte Belegung: $x_1 = x_3 = True,\ x_2 = x_4 = False$

Algorithmus für k-SAT

Algorithmus 12 MonienSpeckenmayer

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

Bestimmung der Laufzeit:
→ Sei t(n) Laufzeit bei beliebiger Eingabe einer 3-KNF über n Variablen.
⇒ $t(n) = const$ für $n \leq 3$
⇒ $t(n) \leq t(n-1) + t(n-2) + t(n-3) + poly(n)$
⇒ Laufzeit in $\mathcal{O}^*(1.84^n)$
→ Verallgemeinert auf k-SAT:
⇒ $t(n) = const$ für $n \leq k$
⇒ $t(n) = t(n-1) + \cdots + t(n-k) + poly(n)$
⇒ $\lambda^k = \lambda^{k-1} + \cdots + \lambda + 1 = \frac{1-\lambda^k}{1-\lambda} \Leftrightarrow \lambda^{k+1} + 1 = 2\lambda^k$
⇒ Lösung für größte Nullstelle ist Basis der exponentiellen Laufzeit!
⇒ $k \to \infty$: Nullstelle → 2

Algorithmen aus den Übungen zu k-SAT

Hamming-Ball Methode

Definition (Hamming-Abstand und Hamming-Kugel)**.**
Der Hamming-Abstand $h(a,b)$ zweier 0-1-Folgen a,b gibt an, an wie vielen Stellen sich die beiden 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.

Algorithmus 13 LocalSearch(a,d)

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

→ Laufzeit: $\mathcal{O}^*(k^d)$

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.
□
→ Bemerke: Startet man LocalSearch(a,d) mit zwei konträren Belegungen, so reicht es zur vollständigen Durchsuchung $d = \frac{n}{2}$ zu setzen.
⇒ Laufzeit hier: $\mathcal{O}^*(k^{n/2})$

Einfacher randomisierter Algorithmus

Algorithmus 14 $GanzEinfach_{\delta \in [0,1/2]}$

1: Rate Belegung $a \in \{0,1\}^n$ für Φ
2: Gib LocalSearch(a,δn) aus

→ Verwende binomielle Abschätzung zur Berechnung, wie häufig der Algorithmus ausgeführt werden muss.
→ Erste Frage: Wie viele b gibt es: $|\mathcal{H}(a, \delta n)| = \sum_{i=0}^{\delta n} \binom{n}{i}$
→ Zweite Frage: Gesamtanzahl Belegungen: 2^n
⇒ $P[\#Algo] = \sum_{i=0}^{\delta n} \binom{n}{i} 2^n = \frac{2^{-n}}{n+1} \left(\left(\frac{1}{\delta}\right)^\delta \cdot \left(\frac{1}{1-\delta}\right)^{1-\delta} \right)^n$
⇒ $E[\#Algo] = {}^1/_{P[\#Algo]}$
→ Erwartete Laufzeit: $E[\#Algo]$ + Laufzeit(LocalSearch)

Der verbesserte Berechnungsbaum

→ Holzhammer: Setze in jedem Level genau ein Literal fest.
⇒ Verzweigungsgrad $2^{\#Literale}$

Algorithmus 15 Verbesserter Berechnungsbaum

1: **for** $i = 1, \ldots, n$ **do**
2: Setze ungesetzte Literale l aus Klausel i fest ▷ $2^{\#l} - 1$
3: Falls Klausel nicht erfüllbar, schneide diesen Ast ab
4: **return** Gültige Belegungen

→ Baum hat im WC nur $\lceil \frac{n}{k} \rceil$ Level.
→ Hilfsmittel: Geometrische Reihe $\sum_{i=0}^{t} a^i = \frac{a^{t+1}-1}{a-1}$
⇒ Laufzeit: $\sum_{i=0}^{\lceil \frac{n}{k} \rceil} (2^k - 1)^i = \frac{(2^k-1)^{\lceil \frac{n}{k} \rceil + 1} - 1}{2^k - 2}$

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: $\mathcal{O}^*((n-1)!) = \mathcal{O}^*((\frac{n}{e})^n)$
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, $S \subseteq \{2, \ldots, n\}, k \in S$
Berechne dies mittels folgender rekursiver Beziehung:

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

⇒ $c(K^*) = \min\{c(\{2, \ldots, n\}, m) + c(k,1) \mid k \in \{2, \ldots, n\}\}$
→ 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 l Möglichkeiten für k.

→ Dazu $2n-3$ Schritte zur Berechnung von $c(K^*)$.
⇒ Laufzeit:

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

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

Zeitplanung

Gegeben: n Jobs J_i, Zeiten $a_1,\ldots,a_n$, Kosten $c_1(t),\ldots,c_n(t)$
Gesucht: Fahrplan π mit minimalen Kosten
→ Maschine muss J_i vollständig erledigen (steht nie still)
→ Reihenfolge kann durch Permutation π 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 $S \subseteq \{J_1,\ldots,J_n\}$ sei $t_S = \sum_{J_i \in S} a_i$.
→ opt(S) bezeichne Kosten eines minimalen Fahrplans in $[0, t_S]$.

Berechnung von opt(S) mittels rekursiver Beziehung:

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

→ Laufzeit von $opt(\{J_1,\ldots,J_n\})$: $\sum_{i=1}^{n-1} i \cdot \binom{n}{i} = \mathcal{O}^*(2^n)$
→ Auch hier kann die Bellmansche Optimalitätsbeziehung angewandt werden
→ Optimale Lösung lässt sich aus einzelnen Jobs bilden.
→ Rekonstruktion: Optimale Reihenfolge kann dadurch bestimmt werden, dass man sich bei jedem 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, sodass Zahlenmultiplikation minimiert werden.
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 $M_i,\ldots,M_j$ sei $m_{i,j}$ die minimale Anzahl an Zahlenmultiplikationen.
Berechnung von $m_{i,j}$ mittels folgender rekursiver Beziehung:

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

→ 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.

Subsetsum in der dynamischen Programmierung:
Problem: Finde Summe der $a_1,\ldots,a_n$, welche S ergeben.
Basisfall: $f(1,S) = \begin{cases} 1 & ,S = a_1 \; oder \; S = 0 \\ 0 & ,sonst \end{cases}$
Rekursiver Aufruf: $f(i,S) = \max\{f(i-1,S), f(i-1,S-a_i)\}$

Klausurfragen:

Kapitel 1: Tiefensuche
1. Erklären Sie DFS für ungerichtete Graphen. Warum funktioniert dies?
2. Definieren Sie Artikulationspunkt und erklären Sie, wie man einen solchen erkennt.
3. Definieren Sie zweifache Zusammenhangskomponente und erläutern Sie einen Ansatz zum Erkennen.
4. Erklären Sie die tief-Funktion und ihre Zusammenhänge zur dfnr.
5. Was ist ein Superstrukturtgraph? Wieso ist dieser wichtig?
6. Wie findet man über Tiefensuche 2fache ZHK? Wo kann ein AP im Keller liegen?
7. Welche Kantentypen gibt es im Tiefensuchwald für Digraphen?
8. Welche Eigenschaften haben Rückkanten im Tiefensuchbaum?
9. Definieren Sie starke Zusammenhangskomponente und stellen Sie eine Verbindung zum Superstrukturgraphen her.
10. Beschreiben Sie die Tiefensuche zum finden starker ZHK. Warum funktioniert dies?
11. Wieso ist die Laufzeit linear in der Eingabelänge?

Kapitel 2: Flüsse in Netzwerken
1. Definieren Sie ein kombinatorisches Optimierungsproblem.
2. Definieren Sie maximalen Fluss.
3. Erklären Sie das Konzept der erweiternden Wege und des Residualgraphen. Wieso kann es zu einer exponentiellen Laufzeit kommen?
4. Wieso ist das Ausnutzen von Vor- und Rückkanten im Residualgraphen wichtig?
5. Beweisen Sie den Satz über erweiternde Wege.
6. Was besagt das Integrality-Theorem?
7. Erklären Sie den Algorithmus von Dinic. Erläutern Sie die Laufzeit.
8. 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. Erklären Sie den Algorithmus von Buss & Goldsmith. Welche Laufzeit hat dieser?
4. Wann kann es kein VC der Größe k geben?
5. Wie kann ein parametrisierter Algorithmus in einen exakten Algorithmus umwandeln?
6. Mit welchem Ansatz kann man Laufzeiten in $\mathcal{O}^*$ bestimmen?

Kapitel 4: Das Erfüllbarkeitsproblem
1. Definieren Sie SAT.
2. Erklären Sie den polynomiellen Algorithmus für 2SAT.
3. Erklären Sei den Algorithmus von Monien-Speckenmayer. Analysieren Sie die Laufzeit.
4. Wie funktioniert SAT über einen Berechnungsbaum?
5. Erklären Sie den Zusammenhang von LocalSearch und Hamming-Ball.
6. Wie kann man LocalSearch randomisieren.

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. Was besagt die Bellmannsche Optimalitätsbedingung?