# 10 Kombinatorische Suche und Rekursionsgleichungen

Bisher meistens Probleme in polynomieller Zeit (und damit auch Platz). Obwohl es exponentiell viele mögliche Wege von u ○⟶○ v gibt, gelingt es mit Dijkstra oder Floyd-Warshall, einen kürzesten Weg systematisch aufzubauen, ohne alles zu durchsuchen. Das liegt daran, dass man die richtige Wahl lokal erkennen kann. Bei Ford-Fulkerson haben wir prinzipiell unendlich viele Flüsse, trotzdem können wir einen maximalen systematisch aufbauen, ohne blind durchzuprobieren. Dadurch erreichen wir polynomiale Zeit.

Jetzt: Probleme, bei denen man die Lösung nicht mehr zielgeirchtet aufbauen kann, sondern im Wesentlichen exponentiell viele Lösungskandidaten durchsuchen muss. (Das bezeichnet man als kombinatorische Suche.)

**Zunächst: Aussagenlogische Probleme.**

Aussagenlogische Probleme, wie zum Beispiel $x \wedge y \vee (\neg((u \wedge \neg(v \wedge \neg x)) \rightarrow y), x \vee y, x \wedge y, (x \vee y) \wedge (x \vee \neg y)$.

Also wir haben eine Menge von aussagenlogischen Variablen zur Verfügung, wie x, y, v, u, .... Formeln werden mittels der üblichen aussagenlogischen Operationen $\wedge$ (und), $\vee$ (oder, lat. vel), $\neg$, $^-$(nicht), $\Rightarrow$ (Implikation), $\Leftrightarrow$ (Äquivalenz) aufgebaut.

Variablen stehen für die Wahrheitswerte 1 (= wahr) und 0 (= falsch). Die Implikation hat folgende Bedeutung:

| x | y | x ⇒ y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Das heißt, das Ergebnis ist nur dann falsch, wenn aus Wahrem Falsches folgen soll.

Die Äquivalenz $x \Leftrightarrow y$ ist genau dann wahr, wenn $x$ und $y$ beide den gleichen Wahrheitswert haben, also beide gleich 0 oder gleich 1 sind. Damit ist $x \Leftrightarrow y$ gleichbedeutend zu $(x \Rightarrow y) \wedge (y \Rightarrow x)$.

Ein Beispiel verdeutlicht die Relevanz der Aussagenlogik:

> **Frage:** Worin besteht das Geheimnis Ihres langen Lebens?
>
> **Antwort:** Folgender Diätplan wird eingehalten:
>
> - Falls es kein Bier zum Essen gibt, dann wird in jedem Fall Fisch gegessen.
> - Falls aber Fisch, gibt es auch Bier, dann aber keinesfalls Eis.

- Falls Eis oder auch kein Bier, dann gibts auch keinen Fisch.

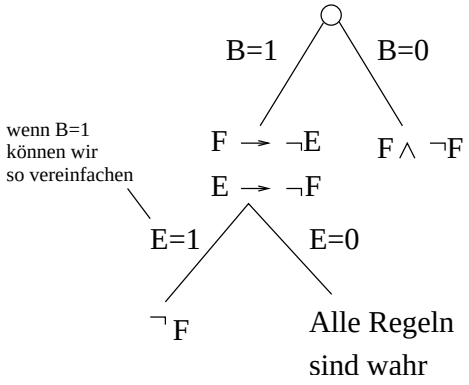
Dazu Aussagenlogik:

$$\begin{aligned} B &= \text{Bier beim Essen} \\ F &= \text{Fisch} \\ E &= \text{Eis} \end{aligned}$$

Ausagenlogisch werden obige Aussagen nun zu:

$$\begin{aligned} \neg B &\Rightarrow F \\ F \wedge B &\Rightarrow \neg E \\ E \vee \neg B &\Rightarrow \neg F \end{aligned}$$

Aussagenlogik erlaubt die direkte Darstellung von Wissen (Wissensrepräsentation - ein eigenes Fach). Wir wollen nun feststellen, was verzehrt wird:




Also: Immer Bier und falls Eis, dann kein Fisch.

$$B \wedge (E \Rightarrow \neg F)$$

Das ist gleichbedeutend zu $B \wedge (\neg E \vee \neg F)$ und gleichbedeutend zu $B \wedge (\neg(E \wedge F))$.

Immer Bier, Eis und Fisch nicht zusammen.

Die Syntax der aussagenlogischen Formeln sollte soweit klar sein. Die Semantik (Bedeutung) kann erst dann erklärt werden, wenn die Variablen einen Wahrheitswert haben, d.h. wir haben eine Abbildung

$$a : \text{Variablen} \to \{0, 1\}.$$

Das heißt eine Belegung der Variablen mit Wahrheitswerten ist gegeben.

Dann ist $a(F)$ = Wahrheitswert von $F$ bei Belegung $a$.

Ist $a(x) = a(y) = a(z) = 1$, dann $a(\neg x \vee \neg y) = 0$, $a(\neg x \vee \neg y \vee z) = 1$, $a(\neg x \wedge (y \vee z)) = 0$.

Für eine Formel $F$ der Art $F = G \wedge \neg G$ gilt $a(F) = 0$ für jedes $a$, für $F = G \vee \neg G$ ist $a(F) = 1$ für jedes $a$.

Einige Bezeichnungen:

- $F$ ist erfüllbar, genau dann, wenn es eine Belgung $a$ mit $a(F) = 1$ gibt.
- $F$ ist unerfüllbar (widersprüchlich), genau dann wenn für alle $a$ $a(F) = 0$ ist.
- $F$ ist tautologisch, genau dann, wenn für alle $a$ $a(F) = 1$ ist.

Beachte: Ist F nicht tautologisch, so heißt das im Allgemeinen nicht, dass $F$ unerfüllbar ist.

## 10.1 Algorithmus (Erfüllbarkeitsproblem)

**Eingabe:** $F$

1. Erzeuge hintereinander alle Belegungen $a(0\ldots0, 0\ldots1, 0\ldots10, \ldots, 1\ldots1)$. Ermittle $a(F)$. Ist $a(F) = 1$, return „$F$ erfüllbar duch $a$.“
2. return „$F$ unerfüllbar.“

**Laufzeit:** Bei $n$ Variablen $O(2^n \cdot |F|)$, wobei $|F|$ = Größe von $F$ ist.

Dabei muss $F$ in einer geeigneten Datenstruktur vorliegen. Wenn $F$ erfüllbar ist, kann die Zeit wesentlich geringer sein! Reiner worst-case. Verbesserung durch backtracking:

$\Rightarrow$ Schrittweises Einsetzen der Belegung und Vereinfachen von $F$ (Davis-Putnam-Prozedur, siehe später).

Die Semantik einer Formel $F$ mit $n$ Variablen lässt sich auch verstehen als eine Funktion $F : \{0,1\}^n \to \{0,1\}$.

Frage: Wieviele derartigen Funktionen gibt es?
Jede derartige Funktion lässt sich in konjunktiver Normalform (KNF) darstellen. Auch in disjunktiver Normalform.

*Konjunktive Normalform ist:*

$$(x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_2 \vee x_3 \vee x_2)$$

*Disjunktive Normalform ist:*

$$(x_1 \wedge x_2 \wedge \neg x_3 \wedge x_4) \vee (x_2 \wedge x_3 \wedge \neg x_4) \vee (\neg x_1 \wedge x_5 \wedge \cdots \wedge x_n) \vee \ldots$$

Im Prinzip reichen DNF oder KNF aus. Das heißt, ist $F : \{0,1\}^n \to \{0,1\}$ eine boolsche Funktion, so lässt sich $F$ als KNF oder auch DNF darstellen.

**KNF:** Wir gehen alle $(b_1, \ldots, b_n) \in \{0,1\}^n$, für die $F(b_1, \ldots, b_n) = 0$ ist, durch.

Wir schreiben Klauseln nach folgendem Prinzip:

$$\begin{aligned} F(0-0) = 0 &\rightarrow x_1 \vee x_2 \vee \cdots \vee x_n \\ F(110-0) = 0 &\rightarrow \neg x_1 \vee \neg x_2 \vee x_3 \vee \cdots \vee x_n \\ &\vdots \end{aligned}$$

Die Konjunktion dieser Klauseln gibt die Formel, die $F$ darstellt.

**DNF:** Alle $(b_1, \ldots, b_n) \in \{0,1\}^n$ für die $F(b_1, \ldots, b_n) = 1$ Klauseln analog oben:

$$\begin{aligned} F(0,0,\ldots,0) = 1 &\rightarrow \neg x_1 \wedge \neg x_2 \wedge \cdots \wedge \neg x_n \\ F(1,1,0\ldots,0) = 1 &\rightarrow x_1 \wedge x_2 \wedge \neg x_3 \wedge \ldots \neg x_n \\ &\vdots \end{aligned}$$

**Beachte:**

- Erfüllbarkeitsproblem bei KNF $\Longleftrightarrow$ Jede (!) Klausel muss ein wahres Literal haben.
- Erfüllbarkeitsproblem bei DNF $\Longleftrightarrow$ Es gibt eine (!) Klausel, die wahr gemacht werden kann.
- Erfüllbarkeitsproblem bei DNF leicht: Gibt es eine Klausel, die nicht $x$ und $\neg x$ enthält.

Schwierig bei DNF:
Gibt es eine Belegung, so dass 0 rauskommt. Das ist nun wieder leicht bei KNF (Wieso?). Eine weitere Einschränkung ist die k-KNF, k-DNF, $k = 1, 2, 3 \ldots i$ : Klauselgröße (= # Literale) $\leq k$ pro Klausel.

1-KNF: $x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \ldots$
2-KNF: $(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_1 \vee \neg x_1) \wedge \ldots$
$x_1 \vee \neg x_1$ - tautologische Klausel, immer wahr, eigentlich unnötig
3-KNF: $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee x_3) \wedge x_1 \wedge \ldots$

Eine unerfüllbare 1-KNF ist $(x_1 \wedge \neg x_1)$. Eine unerfüllbare 2-KNF ist $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2)$.

Interessant ist, dass sich unerfüllbare Formeln auch durch geeignete Darstellung mathematischer Aussagen ergeben können. Betrachten wir den Satz:

> Ist $f : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ injektive Abbildung, dann ist diese Abbildung auch surjektiv.

Dazu nehmen wir $n^2$ viele Variablen. Diese stellen wir uns so vor:

$$\begin{array}{lllll} x_{1,1}, & x_{1,2}, & x_{1,3}, & \dots & x_{1,n} \\ x_{2,1}, & x_{2,2}, & \dots & & x_{2,n} \\ \vdots & & & & \vdots \\ x_{n,1}, & & \dots & & x_{n,n} \end{array}$$

Jede Belegung $a$ der Variablen entspricht einer Menge von Paaren $M$:

$$a(x_{i,j}) = 1 \Longleftrightarrow (i,j) \in M.$$

Eine Abbilding ist eine spezielle Menge von Paaren. Wir bekommen eine widersprüchliche Formel nach folgendem Prinzip:

1. $a$ stellt eine Abbildung dar **und**
2. $a$ ist eine injektive Abbildung **und**
3. $a$ ist nicht surjektive Abbildung.

$$\left.\begin{array}{l} (x_{1,1} \vee x_{1,2} \vee \dots \vee x_{1,n}) \\ \wedge \\ (x_{2,1} \vee x_{2,2} \vee \dots \vee x_{2,n}) \\ \wedge \\ \dots \\ \wedge \\ (x_{n,1} \vee x_{n,2} \vee \dots \vee x_{n,n}) \end{array}\right\} \begin{array}{l} \text{Jedem Element aus } 1, \dots, n \\ \text{ist eines zugeordnet,} \\ n \text{ Klauseln} \end{array}$$

$$\underbrace{(\neg x_{1,1} \vee \neg x_{1,2}) \wedge (\neg x_{1,1} \vee x_{1,3}) \wedge \dots \wedge (\neg x_{1,n-1} \vee \neg x_{1,n})}_{\begin{array}{c} 1 \text{ ist höchstens 1 Element zugeordnet, } \binom{n}{k} \text{ Klauseln} \end{array}}$$

Ebenso für $2, \dots, n$. Damit haben wir gezeigt, dass $a$ eine Abbildung ist.

$$\underbrace{(\neg x_{1,1} \vee \neg x_{2,1}) \wedge (\neg x_{1,1} \vee x_{3,1}) \wedge (\neg x_{1,1} \vee \neg x_{4,1}) \wedge \dots \wedge (\neg x_{1,1} \vee \neg x_{n,1})}_{1 \text{ wird höchstens von einem Element getroffen.}}$$

$\vdots$

Ebenso für $2, \dots, n$. Haben jetzt: $a$ injektive Abbildung.

$$\left.\begin{array}{l}\wedge\\ ((\neg x_{1,1} \wedge x_{2,1} \wedge x_{3,1} \wedge \cdots \wedge \neg x_{n,1})\\ \vee\\ (\neg x_{1,2} \wedge \neg x_{2,2} \ldots)\\ \vdots\\ \vee\\ (\neg x_{1,n} \wedge \cdots \wedge \neg x_{n,n})\end{array}\right\} \; a \text{ nicht surjektiv}$$

Im Falle des Erfüllungsproblems für KNF lässt sich der Backtracking-Algorithmus etwas verbessern. Dazu eine Bezeichnung: $F_{x=1} \Rightarrow x$ auf 1 setzen und $F$ vereinfachen.

$\Rightarrow$ Klauseln mit $x$ löschen (da wahr). In Klauseln mit $\neg x$ das $\neg x$ löschen, da es falsch ist. Analog für $F_{x=0}$.

Es gilt: $F$ erfüllbar $\Longleftrightarrow$ $F_{x=1}$ oder $F_{x=0}$ erfüllbar.

Der Backtracking-Algorithmus für KNF führt zur *Davis-Putman-Prozedur*, die so aussieht:

## 10.2 Algorithmus (Davis-Putnam)

**Eingabe:** $F$ in KNF, Variablen $x_1, \ldots x_n$,
Array $a[1, \ldots, n]$ of *boolean* (für die Belegung)

```
DP(F){
1. if F offensichtlich wahr      // leere Formel
     return "F erfüllbar"        // ohne Klausel
2. if F offensichtlich unerfüllbar
     return "F unerfüllbar"      // Enthält leere Klausel
                                 // oder Klausel x und ¬x.
3. Wähle eine Variable x von F gemäß einer Heuristik.
   Wähle (b1, b2) mit b1 = 1, b2 = 0 oder b1 = 0, b2 = 1
4. H := F_{x=b1}; a[x]:=b1;
   if (DP(H) == "H erfüllbar")
     return "F erfüllbar"
5. else{                         // Nur, wenn in 4. "unerfüllbar"
     H := F_{x=b2}; a[x]:=b2
     return DP(H)}
}
```

Ist `DP(`$F$`)` = „$F$ erfüllbar“, so ist in $a$ die gefundene erfüllende Belegung gespeichert.

**Korrektheit:** Induktiv über die #Variablen in $F$, wobei das $a$ noch einzubauen ist.

## 10.3 Algorithmus (Pure literal rule, unit clause rule)

In die einfache Davis-Putman-Prozedur wurden noch folgedne Heuristiken in 3. eingebaut:

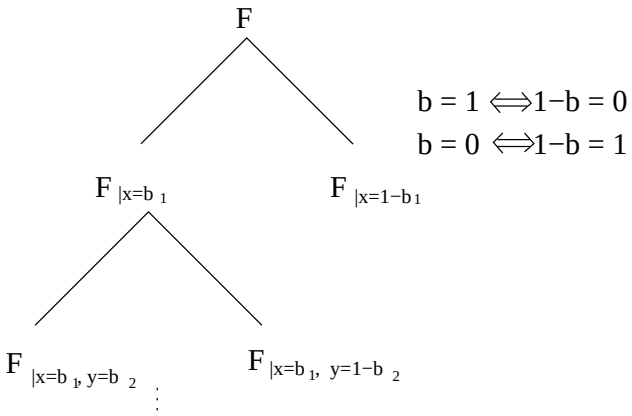
- Pure literal rule ($x$ ist ein „pures Literal“)
  if es gibt ein $x$ in $F$, so dass $\neg x$ nicht in $F\{$
  $H := F_{x=1}$; a[$x$]:=1; return DP($H$);$\}$
  if $\neg x$ in $F$ aber $x$ nicht in $F\{$
  $H := F_{x=0}$; a[$x$]=0; return DP($H$); // Hier kein Backtracking
- Unit clause rule
  if es gibt eine Einerklausel ($x$) in $F\{$
  $H := F_{x=1}$;a[$x$]:=1; return DP($H$);$\}$
  if ($\neg x$) in $F\{$
  $H := F_{x=0}$; a[$x$]=0; return DP($H$);$\}$ // kein Backtracking

Erst danach geht das normale 3. des Davis-Putman-Algorithmus weiter.

**Korrektheit:**

- Pure literal: Es ist $F_{x=1} \subseteq F$, heißt jede Klausel in $F_{x=1}$ tritt auch in $F$ auf. Damit gilt: $F_{x=1}$ erfüllbar $\Longleftarrow$ $F$ erfüllbar und $F_{x=1}$ unerfüllbar $\Longleftarrow$ $F$ unerfüllbar (wegen $F_{x=1} \subseteq F$). Also $F_{x=1}$ erfüllbar $\Longleftrightarrow$ $F$ erfüllbar, Backtracking nicht nötig.
- Unit clause: $F_{x=0}$ ist offensichtlich unerfüllbar,1 wegen leerer Klausel, also kein backtracking nötig.

**Laufzeit:** Prozedurbaum




Sei $T(n)$ = maximale #Blätter bei $F$ mit $n$ Variablen, dann gilt:

$$\begin{aligned} T(n) &\leq T(n-1) + T(n-1) \quad \text{für } n \gneq 1 \\ T(1) &= 2 \end{aligned}$$

Dann das „neue $T(n)$“($\geq$ „altes $T(n)$“):

$$\begin{aligned} T(n) &= T(n-1) + T(n-1) \quad \text{für } n \gneq 1 \\ T(1) &= 2 \end{aligned}$$

Hier sieht man direkt $T(n) = 2^n$.

Eine allgemeine Methode läuft so:

Machen wir den Ansatz (= eine Annahme), dass $T(n) = 2^n$ für ein $\alpha \geq 1$ ist. Dann muss für $n > 1$ $\alpha^n = \alpha^{n-1} + \alpha^{n-1} = 2\alpha^{n-1}$ sein. Also teilen wir durch $\alpha^{n-1}$ :

$$\alpha = 1 + 1 = 2.$$

Muss durch Induktion verifiziert werden, da der Ansatz nicht stimmen muss.

Damit Laufzeit:

$$O((2^n - 1 + 2^n) \cdot |F|) = O(2^n \cdot |F|),$$

$F \ldots$ Zeit fürs Einsetzen.

Also: Obwohl Backtracking ganze Stücke des Lösungsraums, also der Menge $\{0,1\}^n$ rausschneidet, können wir zunächst nichts Besseres als beim einfachen Durchgehen aller Belegungen zeigen.

Bei k-KNF's für ein festes k lässt sich der Davis-Putman-Ansatz verbessern. Interessant ist, dass in gewissem Sinne $k = 3$ reicht (konkret beim Erfüllbarkeitsproblem).

**Satz 10.1:** *Sei $F$ eine beliebige aussagenlogische Formel. Wir können zu $F$ eine 3-KNF $G$ konstruieren mit:*

- *$F$ erfüllbar $\Longleftrightarrow$ $G$ erfüllbar ($F, G$ erfüllbarkeitsäquivalent)*
- *Die Konstruktion lässt sich in der Zeit $O(|F|)$ implementieren. (Insbesondere $|G| = O(|F|)$)*

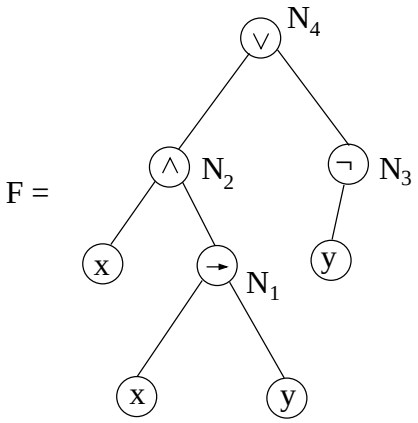
**Beachte:** Durch Ausmultiplizieren lässt sich jedes $F$ in ein äquivalentes $F$ in KNF transformieren. Aber

- Exponentielle Vergrößerung ist möglich.
- keine 3-KNF

***Beweis.*** [des Satzes] Am Beispiel wird eigentlich alles klar. Der Trick besteht in der Einfürhung neuer Variablen!

$$F = (x \wedge (x \Rightarrow y)) \vee \neg y)$$

Schreiben $F$ als Baum: Variablen = Blätter, Operationen = innere Knoten




$N_1, N_2, N_3, N_4$ : neue Variablen, für jeden inneren Knoten eine neue Variable.

Idee: $N_i$ = Wert an den Knoten, bei gegebener Belegung der alten Variablen x, y.

Das drückt $F'$ aus:

$$F' = (N_1 \Leftrightarrow (x \Rightarrow y)) \wedge (N_2 \Leftrightarrow (x \wedge N_1)) \wedge (N_3 \Leftrightarrow \neg y) \wedge (H_4 \Leftrightarrow (N_3 \vee N_2))$$

Es ist F' immer erfüllbar. Brauchen nur die $N_i$ von unten noch eben zu setzen.

**Aber:**

$$F'' = F' \wedge N_4,$$

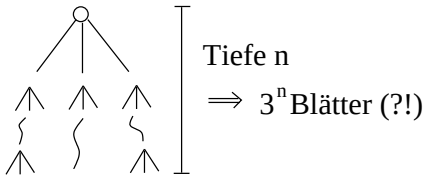
$N_4$ Variable der Wurzel erfordert, dass alle $N_i$ richtig und $N_4 = 1$ ist. Es gilt: Ist $a(F) = 1$, so können wir eine Belegung $b$ konstruieren, in der die $b(N_i)$ richtig stehen, so dass $b(F'') = 1$ ist. Ist andererseits $b(F'') = 1$, so $b(N_4) = 1$ und eine kleine Überlegung zeigt uns, dass die $b(x)$, $b(x)$ eine erfüllende Belegug von $F$ sind.

3-KNF ist gemäß Prinzip auf Seite 127 zu bekommen. Anwendung auf die Äquivalenzen. □

**Frage:** Warum kann man so keine 2-KNF bekommen?

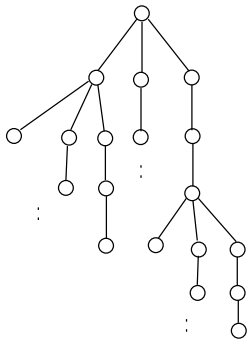
Davis-Putman basiert auf dem Prinzip $F$ erfüllbar $\Longleftrightarrow$ $F_{x=1} oder F_{x=0}$ erfüllbar. (Verzweigen nach dem Wert von $x$) Haben wir eine 3-KNF, etwa $F = \cdots \wedge (x \vee \neg y \vee z) \wedge \ldots$, dann ist $F$ erfüllbar, gdw. $F_{x=1}$ oder $F_{y=0}$ oder $F_{z=1}$ erfüllbar (Verzweigen nach der Klausel).

Ein Backtracking-Algorithmus nach diesem Muster führt zu einer Aufrufstruktur der Art




Aber es gilt auch: $F$ erfüllbar $\Longleftrightarrow$ $F_{x=1}$ erfüllbar oder $F_{x=0,y=0}$ erfüllbar oder $F_{x=0,y=1,z=1}$ erfüllbar.

Damit Aufrufstruktur im Backtracking:



Sei wieder $T(n) :=$ maximale # Blätter bei $b$ Variablen, dann gilt:

$$\begin{aligned} T(n) &\geq T(n-1) + (T(n-1) + T(n-3) \quad \text{für } n \geq n_0, n_0 \text{ eine Konstante} \\ T(n) &\leq d = O(1) \quad \text{für } n < n_0. \end{aligned}$$

Was ist das?

Lösen $T(n) = T(n-1) + T(n-2) + T(n-3)$.

Ansatz:

$$T(n) = c \cdot \alpha^n \quad \text{für alle } n \text{ groß genug.}$$

Dann

$$c \cdot \alpha^n = c \cdot \alpha^{n-1} + c \cdot \alpha^{n-2} + c \cdot \alpha^{n-3},$$

also

$$\alpha^3 = \alpha^2 + \alpha + 1.$$

Wir zeigen: Für $\alpha > 0$ mit $\alpha^3 = \alpha^2 + \alpha + 1$ gilt $T(n) = O(\alpha^n)$.

Ist $n > n_0$, dann $T(n) \leq d \cdot \alpha^n$ für geeignete Konstante $d$, da $\alpha > 0$! Sei $n \geq n_0$.

Dann:

$$\begin{aligned} T(n) = T(n-1) + T(n-2) + T(n-3) &\leq d \cdot \alpha^{n-1} + d \cdot \alpha^{n-2} + d \cdot \alpha^{n-3} \\ &= d(\alpha^{n-3} \cdot \overbrace{(\alpha^2 + \alpha + 1)}^{=\alpha^3 \text{nach Wahl von } \alpha}) \\ &= d \cdot \alpha^n = O(\alpha^n). \end{aligned}$$

Genauso $T(n) = \Omega(\alpha^n)$. Da die angegebene Argumentation für jedes(!) $\alpha > 0$ mit $\alpha^3 = \alpha^2 + \alpha + 1$ gilt, darf es nur ein solches $\alpha$ geben! Es gibt ein solches $\alpha < 1.8393$.

Dann ergibt sich, dass die Laufzeit $O(|F| \cdot 1.8393^n)$ ist, da die inneren Knoten durch den konstanten Faktor mit erfasst werden können.

Frage: Wie genau geht das?

Bevor wir weiter verbessern, diskutieren wir, was derartige Verbesserungen bringen. Zunächst ist der exponentielle Teil maßgeblich (zumindest für die Asymptotik ($n$ groß) der Theorie):

Es ist für $c > 1$ konstant und $\epsilon > 0$ konstant (klein) und $k$ konstant (groß)

$$n^k \cdot c^n \leq^{(1+\epsilon)n},$$

sofern $n$ groß genug ist, denn $c^{\epsilon \cdot n} \geq c^{log_c n \cdot k} = n^k$ für $\epsilon > 0$ konstant und $n$ groß genug (früher schon gezeigt).

Haben wir jetzt zwei Algorithmen

- $A_1$ Zeit $2^n$
- $A_2$ Zeit $2^{\frac{1}{2} \cdot n} = (\sqrt{2})^n = (1.4\ldots)^n$

für dasselbe Problem. Betrachten wir eine vorgegebene Zeit $x$ (fest). Mit $A_1$ können wir alle Eingaben der Größe $n$ mit $2^n \leq x$, d.h. $n \leq log_2 x$ in Zeit $x$ sicher bearbeiten. Mit $A_2$ dagegen alle mit $2^{\frac{1}{2} \cdot n} \leq x$, d.h. $n \leq 2 \cdot log_2$, also Vergrößerung um einen konstanten Faktor! Lassen wir dagegen $A_1$ auf einem neuen Rechner laufen, auf dem jede Instruktion 10-mal so schnell abläuft, so $\frac{1}{10} \cdot 2^n \leq x$, d.h. $n \leq log_2 10x + log_2 10$ nur die Addition einer Konstanten.

**Fazit:** Bei exponentiellen Algorithmen schlägt eine Vergrößerung der Rechengeschwindigkeit weniger durch als eine Verbesserung dr Konstante $c$ in $c^n$ der Laufzeit.

**Aufgabe:** Führen Sie die Betrachtung des schnelleren Rechners für polynomielle Laufzeiten $n^k$ durch. Haben wir das literal $x$ pur in $F$ (d.h. $\neg a$ ist nicht dabei), so reicht es, $F_{z=1}$ zu betrachten. Analoges gilt, wenn wir mehrere Variablen gleichzeitig ersetzen. Betrachten wir die Variablen $x_1, \ldots, x_k$ und Wahrheitswerte $b_1, \ldots, b_k$ für diese Variablen und gilt $H = F_{x_1=b1, x_2=b_2, \ldots, x=b_k} \subseteq F$, d.h. jede Klausel $C$ von $H$ ist schon in $F$, d.h., wenn das Setzen von einem $x_i = b_j$ wahr wird, so gilt

$$F \text{ erfüllbar} \iff H \text{ erfüllbar}$$

wie beim Korrektheitsbeweis der *pure literal rule*. Kein backtracking nötig!

Wir sagen, die Belegung $x_1 = b_1, \ldots, x_k = b_k$ ist *autark* für $F$. Etwa $F = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge G, G$ ohne $x_3, x_1, \neg x_2$.

Dann $F_{x_1=0, x_2=0, x_3=0} \subseteq F$, also $x_1 = 0, x_2 = 0, x_3 = 0$ autark für $F$. (In $G$ fallen höchstens Klauseln weg.)

## 10.4 Algorithmus (Monien, Speckenmeyer)

```
MoSP(F){
1. if F offensichtlich wahr
     return "erfüllt"
2. if F offensichtlich unerfüllbar
     return "unerfüllbar"
3. Wähle eine kleinste Klausel C,
     C=l_1 ∨ ··· ∨ l_i in F                    // l_i Literal, x oder ¬ x
4. Betrachte die Belegungen
     l_1 = 1; l_1 = 0, l_2 = 1; ... l_1 = 0, l_2 = 0, ..., l_s = 1
5. if (eine der Belegungen autark in F)
     b := eine autarke Belegung; return MoSP(F_b)
6. Teste = MoSP(F_i) für i=1, ..., s          // Hier ist keine der
   und F_i jeweils durch Setzen einer der     // Belegungen autark
   obigen Belegungen; return "erfüllbar",
   wenn ein Aufruf "erfüllbar" ergibt,
   "unerfüllbar" sonst.
```

Wir beschränken uns bei der Analyse der Laufzeit auf den 3-KNF-Fall. Sei wieder $T(n) = \#$ Blätter bei kleinster Klausel mit 3 Literalen und $T'(n) = \#$ Blätter bei kleinster Klausel mit $\leq 2$ Literalen.

Für $T(n)$ gilt:

$T(n) \leq T(n-1)$ (Autarke Belegung gefunden, minestens 1 Variable weniger)
$T(n) \leq T'(n-1) + T'(n-2) + T'(n-3)$ (Keine autarke Belegung gefunden,

dann aber Klauseln der Größe $\leq 2$, Algorithmus nimmt immer die kleinste Klausel).

Ebenso bekommen wir

$$\begin{aligned} T'(n) &\leq T(n-1) \\ T'(n) &\leq T'(n-1) + T'(n-2) \quad \text{(Haben } \leq \text{2er Klausel ohne autarke Belegung)} \end{aligned}$$

Zwecks Verschwinden des $\leq$-Zeichens neue $T(n), T'(n) \leq$ die alten $T(n), T'(n)$.

$$\begin{aligned} T(n) &= d \quad \text{für } n \leq n_0, d \text{ Konstante} \\ T(n) &= \max\{T(n-1), T'(n-1) + T'(n-2) + T'(n-3)\} \quad \text{für } n > n_0 \\ T'(n) &= d \quad \text{für } n \leq n_0, d \text{ Konstante} \\ T'(n) &= \max\{T(n-1), T'(n-1) + T'(n-2)\} \end{aligned}$$

Zunächst müssen wir das Maximum loswerden. Dazu zeigen wir, dass für $S(n) = d$ für $n \leq n_0$.

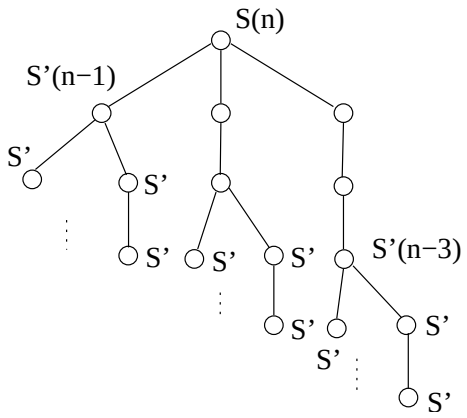
$$\begin{aligned} S(n) &= S'(n-1) + S'(n-2) + S'(n-3) \quad \text{für } n > n_0, S'(n) = d \text{ für } n \leq n_0 \\ S'(n) &= S'(n-1) + S'(n-2) \quad \text{für } n > n_0 \text{ gilt, dass } T(n) \leq S(n), T'(n) \leq S'(n). \end{aligned}$$

Ind.-Anfang ✓

Induktionsschluss

$$\begin{aligned} T(n) &= \max\{T'(n-1) + T'(n-2) + T'(n-3)\} \\ &\leq \text{(durch Ind.-Voraussetzung)} \\ &\quad \max\{S(n-1), S'(n-1) + S'(n-2) + S'(n-3)\} \\ &\leq \text{(durch Monotonie)} \\ &\quad S'(n-1) + S'(n-2) + S'(n-3)\} = S(n). \\ T'(n) &= \max\{T(n-1), T'(n-1) + T'(n-2) \\ &\leq \text{(durch Ind.-Voraussetzung)} \\ &\quad \max\{S(n-1), S'(n-1) + S'(n-2)\} \\ &= \max\{\overbrace{S'(n-2) + S'(n-3)}^{=S'(n-1)} + S'(n-4), S'(n-1) + S'(n-2)\} \\ &= \text{(durch Monotonie)} \\ &\quad S'(n-1) + S'(n-2) = S'(n). \end{aligned}$$

Der Rekursionsbaum für $S(n)$ hat nun folgende Struktur:




Ansatz: $S'(n) = \alpha^n, \alpha^{n-1} + \alpha^{n-2} \Longrightarrow \alpha^2 = \alpha + 1$

Sei $\alpha > 0$ Lösung der Gleichung, dann $S'(n) = O(\alpha^n)$.

Ind.-Anfanfang: $n \leq n_0$, da $\alpha > 0$.

Induktionsschluss:

$$\begin{aligned} S'(n+1) &= S'(n) + S'(n-1) \\ &\text{Induktionsvoraussetzung} \\ &\leq c \cdot \alpha^n + c \cdot \alpha^{n-1} \\ &\text{Wahl von } \alpha \\ &= c \cdot \alpha^{n-1}(\alpha + 1) \\ &= c \cdot \alpha^{n-1} \cdot \alpha^2 \\ &= c \cdot \alpha^{n+1} = O(\alpha^{n+1}). \end{aligned}$$

Dann auch $S(n) = O(\alpha^n)$ und Laufzeit von Monien-Speckenmayer ist $O(\alpha^n \cdot |F|)$. Für $F$ in 3-KNF ist $|F| \leq (2n)^3$, also Zeit $O(\alpha^{(1+\epsilon)n})$ für $n$ groß genug. Es ist $\alpha < 1.681$.

Die Rekursionsgleichung von $S'(n)$ ist wichtig.

**Definition 10.1:** *Die Zahlen* $(F_n)_{n \geq 0}$ *mit*

$$\begin{aligned} F_0 &= 1, \\ F_1 &= 1, \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

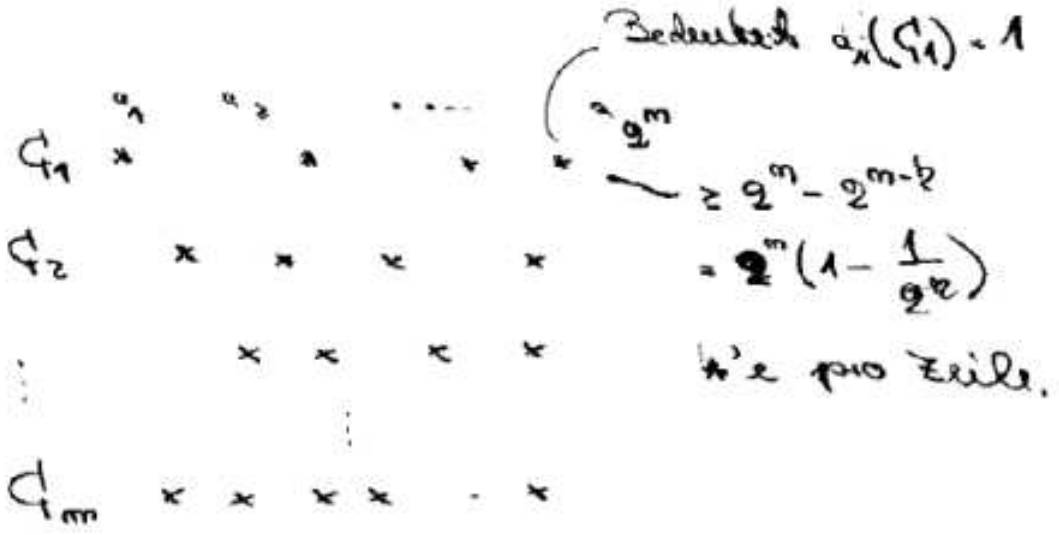
*für alle* $n \geq 2$ *heißen Fibonacci-Zahlen. Es ist* $F_n = O(1.681^n)$.

Verwandt mit dem Erfüllbarkeitsproblem ist das Max-Sat- und das Max-k-Sat-Problem. Dort hat man als Eingabe eine Formel $F = C_1 \wedge \cdots \wedge C_m$ in KNF und sucht

eine Belegung, so dass $|\{i|a(C) = 1\}|$ maximal ist. Für das Max-Ek-Sat-Problem, d.h. jede Klausel besteht aus k verschiedenen Literalen, hat man folgenden Zusammenhang.

**Satz 10.2:** *Sei $m = \#F = C_1 \wedge \cdots \wedge C_m$. Formel gemäß Max-Ek-Sat. ($C_i = C_j$ ist zugelassen.) Es gibt eine Belegung $a$, so dass $|\{j|a(C_j) = 1\}| \geq (1 - \frac{1}{2^k}) \cdot m$.*

***Beweis.*** Eine feste Klausel $C \subseteq \{C_1, \ldots, C_m\}$ wird von wievielen Belegungen falsch gemacht? $2^{n-k}, n = \#$ Variablen, denn die $k$ Literale der Klausel müssen alle auf 0 stehen. Also haben wir folgende Situation vorliegen:




Zeilenweise Summation ergibt:

$$m \cdot 2^n \cdot (1 - \frac{1}{2}) \leq \sum_{j=1}^{m} \{a_i|a_i(C_j) = 1\}| = \#\text{ der } \times \text{ insgesamt.}$$

Spaltenweise Summation ergibt:

$$\sum_{i=1}^{2^n} |\{j|a_i(C_j) = 1\}| = \#\text{ der } \times \text{ insgesamt}, \geq 2^n \cdot (m \cdot (1 - \frac{1}{2^k}))$$

Da wir $2^n$ Summanden haben, muss es ein $a_i$ geben mit

$$|\{j|a_i(C_j) = 1\}| \leq m \cdot (1 - \frac{1}{2^k}).$$

□

Wie findet man eine Belegung, so dass $\{j|a(C_j) = 1\}$ maximal ist? Letztlich durch Ausprobieren, sofern $k \geq 2$, also Zeit $O(|F| \cdot 2^n)$. (Für $k = 2$ geht es in $O(c^n)$ für ein $c < 2$.)

Wir haben also 2 ungleiche Brüder:

1. 2-Sat: polynomiale Zeit
2. Max-2-Sat: nur exponentielle Algorithmen vermutet

Vergleiche bei Graphen mit Kantenlängen $\geq 0$ : kürzester Weg polynomial, längster kreisfreier Weg nur exponentiell bekannt.

Auch 2-färbbar: poly

3-färbbar: nur exponentiell

Ein ähnliches Phäomen liefert das Problem des maximalen Schnittes:

Für einen Graphen $G = (V, E)$ ist ein Paar $(S_1, S_2)$ mit $(S_1 \cup S_2 = V)$ ein Schnitt $(S_1 \cap S_2 = \emptyset, S_1 \cup S_2 = V)$. Eine Kante $\{v, w\}, v \neq w$,liegt im Schnitt $(S_1, S_2)$, genau dann, wenn $v \in S_1, w \in S_2$ (oder auch umgekehrt).

Es gilt: $G$ 2-färbbar $\Longleftrightarrow$ Es gibt einen Schnitt, so dass jede Kante in diesem Schnitt liegt. Beim Problem des maximalen Schnittes geht es darum, einen Schnitt $(S_1, S_2)$ zu finden, so dass $|\{e \in E | e \text{ liegt in } (S_1, S_2)\}|$ maximal ist. Wie beim Max-Ek-Sat gilt:

**Satz 10.3:** *Zu $G = (V, E)$ gibt es einen Schnitt $(S_1, S_2)$ so, dass $|\{e \in E | e \text{ in } (S_1, S_2)\}| \geq \frac{|E|}{2}$*
*Beweis: Übungsaufgabe.*

## 10.5 Algorithmus

(Findet den Schnitt, in dem $\geq \frac{|E|}{2}$ Kanten liegen.)

Eingabe: $G = (V, E), V\{1, \ldots, n\}$

```
1. for v=1 to n{
2. if Knoten v hat mehr direkte Nachbarn in S2 als in S1{
      S1 = S1 ∪ {v}; break;
   }
3. S2 = S2 ∪ {v}}
```

**Laufzeit:** $O(n^2), S_1, S_2$ als boolsches Array

Wieviele Kanten liegen im Schnitt $(S_1, S_2)$?

**Invariante:**
Nach dem j-ten Lauf gilt: Von den Kanten $v, w \in E$ mit $v, w \in S_{1,j} \cup S_{2,j}$ liegt mindestens die Hälfte im Schnitt $(S_{1,j}, S_{2,j})$. Dann folgt die Korrektheit. So findet man aber nicht immer einen Schnitt mit einer maximalen Anzahl Kanten.

**Frage:** Beispiel dafür.

Das Problem minimaler Schnitt dagegen fragt nach einem Schnitt $(S_1, S_2)$, so dass $|\{e \in E | e \text{ in } (S_1, S_2)\}|$ minimal ist. Dieses Problem löst Ford-Fulkerson in polynomialer Zeit.

**Frage:** Wie? $\rightsquigarrow$ Übungsaufgabe.

Für den maximalen Schnitt ist ein solcher Algorithmus nicht bekannt. Wir haben also wieder:
Minimaler Schnitt: polynomial
Maximaler Schnitt: nur exponentiell bekannt

Die eben betrachteten Probleme sind kombinatorische Optimierungsprobleme. Deren bekanntestes ist das Problem des Handlungsreisenden *TSP- Travelling Salesmen Problem.*

**Definition 10.2(TSP, Problem des Handlungsreisenden):** *gegeben: gerichteter, gewichteter Graph*
*gesucht: eine Rundreise (einfacher Kreis) mit folgenden Kriterien:*

- *enthält alle Knoten*
- *Summe der Kosten der betretenen Kanten minimal*

Der Graph ist durch eine Distanzmatrix gegeben. ($\infty \Leftrightarrow$ keine Kante). Etwa $M = (M(u,v)), 1 \leq u,v \leq 4$

$$\begin{array}{cc} & \begin{array}{cccc} \multicolumn{4}{c}{1 \leq u,v \leq 4} \end{array} \\ M = & \begin{pmatrix} \infty & 10 & 15 & 20 \\ 5 & \infty & 9 & 10 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{pmatrix} \\ \textit{Spalte} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \end{array} \quad \begin{array}{c} \textit{Zeile} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}$$

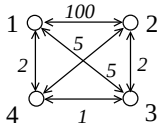
Die Kosten der Rundreise $R = (1,2,3,4,1)$ sind $K(R) = M(1,2) + M(2,3) + M(3,4) + M(4,1) = 10 + 9 + 12 + 8 = 39$ Für $R' = (1,2,4,3,1)$ ist $K(R') = 35$ Ist das minimal?
1. Versuch:
Knoten 1 festhalten, alle $(n-1)!$ Permutationen auf $2, \ldots n$ aufzählen, jeweils Kosten ermitteln.
**Zeit:**

$$\Omega((n-1)! \cdot n) = \Omega(n!) \geq \{(\frac{n}{2}\})^{(\frac{n}{2})}$$
$$= 2^{((logn)-1)\cdot\frac{n}{2}} = 2^{\frac{(logn)-1}{2}\cdot n} = 2^{\Omega(logn)\cdot n} \gg 2^n$$

**Beachte:**

$$n^n = 2^{(logn)\cdot n}, \quad n! \geq 2^{\frac{logn-1}{2}\cdot n}, \quad n! \ll n^n$$

**2. Versuch:**
Greedy. Gehe zum jeweils nächsten Knoten, der noch nicht vorkommt (analog Prim). Im Beispiel mit Startknoten 1 $R = (1,2,3,4,1), K(R) = 39$ nicht optimal. Mit Startknoten 4 $R' = (4,2,1,3,4), K(R') = 35$ optimal.
Aber: Greedy geht es zwingend in die Irre.




| Greedy | |
|---|---|
| (2,3,4,1,2) | Kosten = 105 |
| (3,4,1,2,3) | Kosten = 105 |
| (4,3,2,1,4) | Kosten = 105 |
| (1,4,3,2,1) | Kosten = 105 |

Immer ist $1 \circ\!\longleftrightarrow\!\circ 2$ dabei. Optimal ist (1,4,2,3,1) mit Kosten von 14. Hier ist $4 \circ\!\longrightarrow\!\circ 2$ nicht greedy, sondern mit Voraussicht(!) gewählt. Tatsächlich sind für TSP nur Exponentialzeitalgorithmen bekannt.
Weiter führt unser backtracking. Wir verzweigen nach dem Vorkommen einer Kante in der Rundreise. Das ist korrekt, denn: Entweder eine optimale Rundreise enthält diese Kante $4 \circ\!\longrightarrow\!\circ 2$ oder eben nicht.

Gehen wir auf unser Beispiel von Seite 141 Wir haben die aktuelle Matrix

$$M = \begin{pmatrix} \infty & 10 & 15 & 20 \\ 5 & \infty & 9 & 10 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{pmatrix}$$

Wählen wir jetzt zum Beispiel die Kante $2 \circ\!\longrightarrow\!\circ 3$, dann gilt:

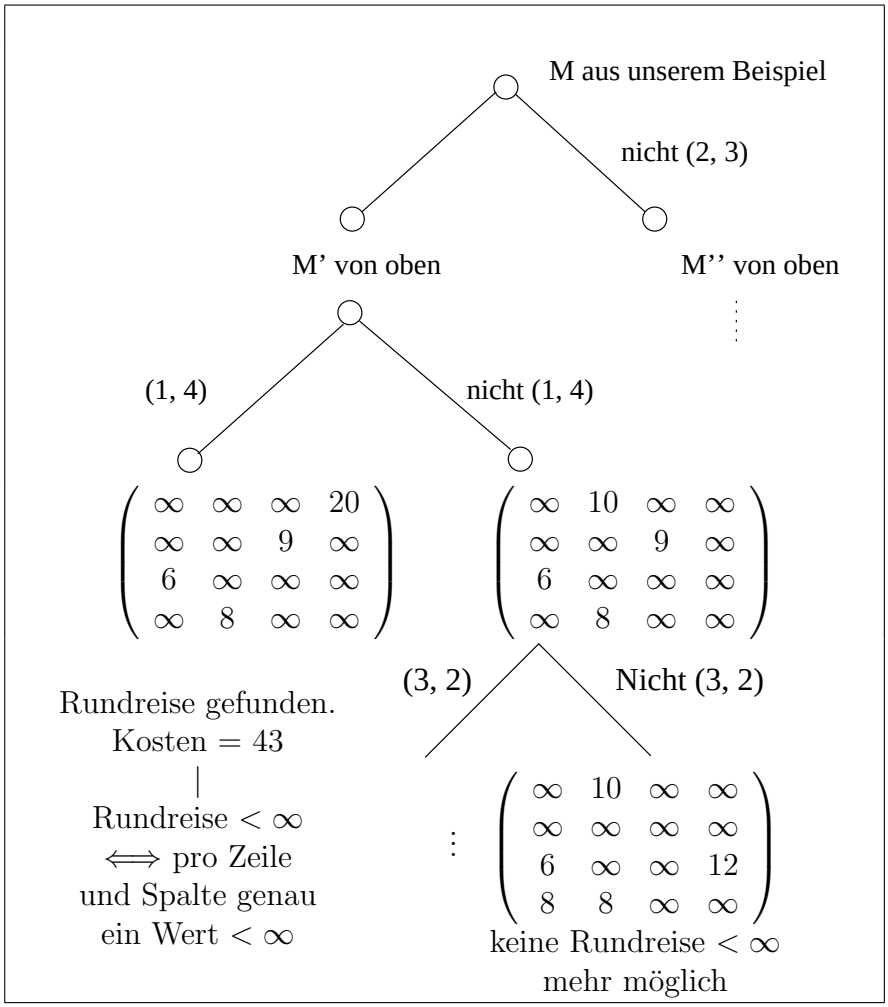
- keine Kante $u \circ\!\longrightarrow\!\circ 3$ , $u \neq 2$ muss noch betrachtet werden
- keine $2 \circ\!\longrightarrow\!\circ v$ , $v \neq 3$
- nicht $3 \circ\!\longrightarrow\!\circ 2$

Das heißt, wir suchen eine optimale Reise in

$$M' = \begin{pmatrix} \infty & 10 & \infty & 20 \\ \infty & \infty & 9 & \infty \\ 6 & \infty & \infty & 12 \\ 8 & \infty & \infty & \infty \end{pmatrix}$$

Alle nicht mehr benötigten Kanten stehen auf $\infty$. Ist dagegen $2 \circ\!\longrightarrow\!\circ 3$ nicht gewählt, dann

$$M' = \begin{pmatrix} \infty & 10 & 15 & 20 \\ 5 & \infty & \infty & 10 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{pmatrix}$$




## 10.6 Algorithmus (Backtracking für TSP)

Eingabe: M=(M(u,v)), $1 \leq$ u,v $\leq$ n
Ausgabe: Eine optimale Rundreise dargstellt als Modifikation von M.

```
1. if M stellt Rundreise dar
     return (M, Kosten von M)
2. if M hat keine Rundreise ≤ ∞
     return (M, ∞)                  // Etwa eine Zeile voller ∞,
                                    // eine Spalte voller ∞
3. Wähle (u,v), u≠v mit M(u,v) ≤ ∞,
   wobei in Zeile von u oder Spalte von v mindestens ein Wert ≠ ∞
4. M':=M modifiziert, so dass u ○——→○ v gewählt
                                    //vgl. Seite 142
5. M'':= M modifiziert, dass M(u,v) = ∞
6. Führe TSP(M') aus
   Führe TSP(M'') aus
```

```
7. Vergleiche die Kosten;
  return (M, K), wobei M die Rundreise der kleineren Kosten ist.
```

**Korrektheit:**
Induktion über die # Einträge in $M = \infty$ (nimmt jedesmal zu). Zeit zunächst $O(2^{n(n-1)}), 2^{n(n-1)} = 2^{\Omega(n^2)} \gg n^n$!

Verbesserung des backtracking durch branch-and-bound.
Dazu:
Für Matrix M(=Knoten im Baum): $S(M)$ = untere(!) Schranke an die Kosten aller Rundreisen zu M, d.h. $S(M) \leq$ Kosten aller Rundreisen unter $M$ im Baum.

Prinzip: Haben wir eine Rundreise der Kosten $K$ gefunden, dann keine Auswertung (Expansion) der Kosten $M$ auf $S(M) \geq K$ mehr.
Allgemein ist es immer so:
$S(k)$ untere Schranke bei Minimierungsproblemen
$S(k)$ obere Schranke bei Maximierungsproblemen (etwa Max-k-Sat),
wobei $k$ Knoten im backtracking-Baum.
Es folgen vier konkrete $S(M)$s für unser TSP-Backtracking:

1. $S_1(M)$ = minimale Kosten einer Rundreise unter $M$. (Das ist die beste (d.h. größte) Schranke, aber nicht polynomial zu ermitteln.)

2. $S_2(M)$ = Summe der Kosten der bei M gewählten Kanten.

   Ist untere Schranke bei $M(u,v) \geq 0$ (Das können wir aber hier annehmen, im Unterschied zu den kürzesten Wegen. (Wieso?))

   Ist $M$ = Wurzel des backtracking-Baumes, dann im allgemeinen $S_2(M) = 0$, sofern nicht $M = (\infty\infty\infty m\infty\infty)$ oder analog für Spalte.

3. $S_3(M) = \frac{1}{2} \cdot \sum_v \min\{M(u,v) + M(v,\infty) | u, v \in V\}$ (Jedesmal das Minimum, um durch den Knoten $v$ zu gehen.) Wieso ist $S_3(M)$ untere Schranke?

   Sei $R$ eine Rundreise unter $k$. Dann gilt: $(R = (1, v_1, \ldots, v_{n-1}, 1))$

$$
\begin{aligned}
K(R) &= M(1,v_1) + M(v_1,v_2) + \cdots + M(v_{n-1},1) \\
&= \frac{1}{2}\Big(M(1,v_1) + M(1,v_1) + \cdots + M(v_{n-1},1) + M(v_{n-1},1)\Big) \\
&\qquad \text{(Shift um 1 nach rechts.)} \\
&= \frac{1}{2}\Big(M(v_{n-1},\underline{1}) + M(\underline{1},v_1) + M(1,\underline{v_1}) + M(\underline{v_1},v_2) + \\
&\qquad M(v_1,\underline{v_2}) + M(\underline{v_2},v_3) + \\
&\qquad\qquad \cdots + M(v_{n-2},\underline{v_{n-1}}) + M(\underline{v_{n-1}},1)\Big) \\
&\geq \frac{1}{2} \cdot \sum_v \min\{M(u,v) + M(v,w) | u, w \in V\}
\end{aligned}
$$

Also ist $S_3(M)$ korrekte Schranke.

Es ist $S_3(M) = \infty \Longleftrightarrow$ Eine Zeile oder Spalte voller $\infty$. (Ebenso für $S_2(M)$).

Betrachten wir $M$ von Seite 141 Wir ermitteln $S_2(M)$.

v=1 5+10 $2 \circ\!\!\xrightarrow{1}\!\!\circ 2$

v=2 8+5 $4 \circ\!\!\xrightarrow{2}\!\!\circ 1$

v=3 9+6 $2 \circ\!\!\xrightarrow{3}\!\!\circ 1$ oder $4 \circ\!\!\xrightarrow{3}\!\!\circ 1$

v=4 10+8 $2 \circ\!\!\xrightarrow{4}\!\!\circ 1$ oder $2 \circ\!\!\xrightarrow{4}\!\!\circ 2$

Also $S_3(M) = \dfrac{1}{2} \cdot 61 = 30,5$.

Wegen ganzzahliger Kosten sogar $S_3(M) \geq 31$. Also heißt das $K(R) \geq 31$ für jede Reise $R$.

Ist $2 \circ\!\!\longrightarrow\!\!\circ 3$ bei $M$ gewählt, dann

$$M' = \begin{pmatrix} \infty & 10 & \infty & 20 \\ \infty & \infty & 9 & \infty \\ 6 & \infty & \infty & 12 \\ 8 & 8 & \infty & \infty \end{pmatrix}$$

v=1 6+10 $3 \circ\!\!\xrightarrow{1}\!\!\circ 2$

v=2 8+9 $4 \circ\!\!\xrightarrow{2}\!\!\circ 3$

v=3 9+6 $2 \circ\!\!\xrightarrow{3}\!\!\circ 1$

v=4 12+8 $3 \circ\!\!\xrightarrow{4}\!\!\circ 2$ oder $3 \circ\!\!\xrightarrow{4}\!\!\circ 1$

$S_3(M') = \frac{1}{2}(16 + 17 + 15 + 20) = 34$, wogegen $S_2(M') = 9$.

4. Betrachten wir eine allgemeine Matrix $M = \big(M(u,v)\big)_{1 \leq u,v \leq n}$. Alle Einträge $M(u,v) \geq 0$.

   Ist $R$ eine Rundreise zu $M$, dann ergeben sich die Kosten von $R$ als $K(R) = z_1 + z_2 + \cdots + z_n$, wobei $z_n$ geeignete Werte aus Zeile n darstellt, alternativ ist $K(R) = s_1 + s_2 + \cdots + s_n$, wobei $s_n$ einen geeigneten Wert aus Spalte n darstellt. Dann gilt wieder

$$\begin{aligned} S_4'(M) \;=\; & \min\{\overbrace{M(1,1), M(1,2), \ldots, M(1,n)}^{\text{Zeile 1}}\} + \\ & \min\{M(2,1), M(2,2), \ldots, M(2,n)\} + \\ & \ldots + \\ & \min\{M(n,1), M(n,2), \ldots, M(n,n)\} \end{aligned}$$

nimmt aus jeder Zeile den kleinsten Wert. Ist korrekte Schranke. Wenn jetzt noch nicht alle Spalten vorkommen, können wir noch besser werden. Und zwar so:

Sei also für $1 \leq u \leq n$ $M_u = \min\{M(u,1), M(u,2), \ldots, M(u,n)\}$ = kleinster Wert der Zeile u. Wir setzen

$$\hat{M} = \begin{pmatrix} M(1,1) - M_1 & \ldots & M(1,n) - M_1 \\ M(2,1) - M_2 & \ldots & M(2,n) - M_2 \\ & \vdots & \\ M(n,1) - M_n & \ldots & M(n,n) - M_n \end{pmatrix}$$

Alles $\geq 0$. Pro Zeile ein Eintrag $= 0$.

Es gilt für jede Rundreise $R$ zu $M$, $K_M(R) = K_{\hat{M}}(R) + S_4'(M) \geq S_4'(M)$. mit $K_M$ = Kosten in $M$ und $K_{\hat{M}}$ = Kosten in $\hat{M}$, in $\hat{M}$ alles $\geq 0$.

Ist in $\hat{M}$ in jeder Spalte eine 0, so $S_4'(M)$ = die minimalen Kosten einer Rundreise. Ist das nicht der Fall, iterieren wir den Schritt mit den Spalten von $\hat{M}_u$.

Ist also $S = \min\{\overbrace{\hat{M}(1,u), \hat{M}(2,u), \ldots, \hat{M}(n,u)}^{\text{Spalte von } \hat{M}}\}$, dann

$$\hat{\hat{M}} = \begin{pmatrix} \hat{M}(1,1) - S_1 & \ldots & \hat{M}(1,n) - S_n \\ \hat{M}(2,1) - S_1 & \ldots & \hat{M}(2,n) - S_n \\ & \vdots & \\ \hat{M}(n,1) - S_1 & \ldots & \hat{M}(n,n) - S_n \end{pmatrix}$$

Alles $\geq 0$. Pro Spalte eine 0, pro Zeile eine 0.

Für jede Rundreise $R$ durch $\hat{M}$ gilt $K_{\hat{M}}(R) = K_{\hat{\hat{M}}}(R) + S_1 + \cdots + S_n$.

Also haben wir insgesamt:

$$\begin{aligned} K_M(R) &= K_{\hat{M}}(R) + M_1 + \cdots + M_n \\ &= K_{\hat{\hat{M}}}(R) + \overbrace{S_1 + \cdots + S_n}^{\text{aus } \hat{M}} + \underbrace{M_1 + \cdots + M_n}_{\text{aus M}} \\ &\geq M_1 + \cdots + M_n + S_1 + \cdots + S_n. \quad (\text{In } \hat{\hat{M}} \text{ alles} \geq 0.) \end{aligned}$$

Wir definieren offiziell:

$$S_4(M) = M_1 + \cdots + M_n + S_1 + \ldots S_n$$

Aufgabe: $S_1(M) \geq S_4(M) \geq S_3(M) \geq S_2(M)$. ($S_4(M) \geq S_3(M)$ ist nicht ganz so offensichtlich.)

In unserem Eingangsbeispiel von Seite 141

$$M = \begin{pmatrix} \infty & 10 & 15 & 20 \\ 5 & \infty & 9 & 10 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{pmatrix}$$

$$M_1 = 10, M_2 = 5, M_3 = 6, M_4 = 8$$

$$\hat{M} = \begin{pmatrix} \infty & 0 & 5 & 10 \\ 0 & \infty & 4 & 5 \\ 0 & 7 & \infty & 6 \\ 0 & 0 & 1 & \infty \end{pmatrix}$$

$$S_1 = 0, S_2 = 0, S_3 = 1, S_4 = 5$$

$$\hat{\hat{M}} = \begin{pmatrix} \infty & 0 & 4 & 10 \\ 0 & \infty & 3 & 0 \\ 0 & 7 & \infty & 1 \\ 0 & 0 & 0 & \infty \end{pmatrix}$$

$$\text{Also} S_4(M) = 35 > S_3(M) = 31.$$

Die Schranken lassen sich für jede Durchlaufreihenfolge des backtracking-Baumes zum Rausschneidem weiterer Teile verwenden.

## 10.7 Algorithmus (Offizielles branch-and-bound)

Front des aktuellen Teils des backtracking-Baumes im Heap geordnet nach $S(M)$ ($S(M)$ = die verwendete Schranke).

Immer an dem $M$ mit $S(M)$ minimal weitermachen. Minimale gefundene Rundreise vermerken. Anhalten, wenn $S(M) \geq$ Kosten der minimalen Reise für $S(M)$ minimal im Heap.

Offizielles Branch-and-Bound mit Beispiel von Seite 141 und Schranke $S_4$.

$$\begin{pmatrix} \infty & 10 & 15 & 20 \\ 5 & \infty & 9 & 10 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{pmatrix}$$

Schranke = 35

nicht (2,1) (2,1)

$$\begin{pmatrix} \infty & & & \\ \infty & \infty & & \\ & & & \\ & & & \end{pmatrix} \qquad \begin{pmatrix} \infty & \infty & & \\ 5 & \infty & \infty & \infty \\ \infty & & & \\ \infty & & & \end{pmatrix}$$

Schranke = 34
Nehmen weiter
Schranke 35!

Schranke = 40

nicht (3,1) (3,1)

$$\begin{pmatrix} \infty & & & \\ \infty & & & \\ \infty & & & \\ & & & \end{pmatrix} \qquad \begin{pmatrix} \infty & & \infty & \\ \infty & \infty & & \\ 6 & \infty & \infty & \infty \\ \infty & & & \end{pmatrix}$$

Schranke = 39

Schranke = 34
weiter 35

nicht (4,2) (4,2)

$$\begin{pmatrix} \infty & 10 & \infty & 20 \\ \infty & \infty & 9 & 10 \\ 6 & \infty & \infty & \infty \\ \infty & \infty & 9 & \infty \end{pmatrix} \qquad \begin{pmatrix} \infty & \infty & \infty & 20 \\ \infty & \infty & 9 & 10 \\ 6 & \infty & \infty & \infty \\ \infty & 8 & \infty & \infty \end{pmatrix}$$

Schranke = 35
(10+9+6+9+1 = 35)

Schranke = 43
(20+9+6+8 = 43)

nicht (2,4) (2,4)

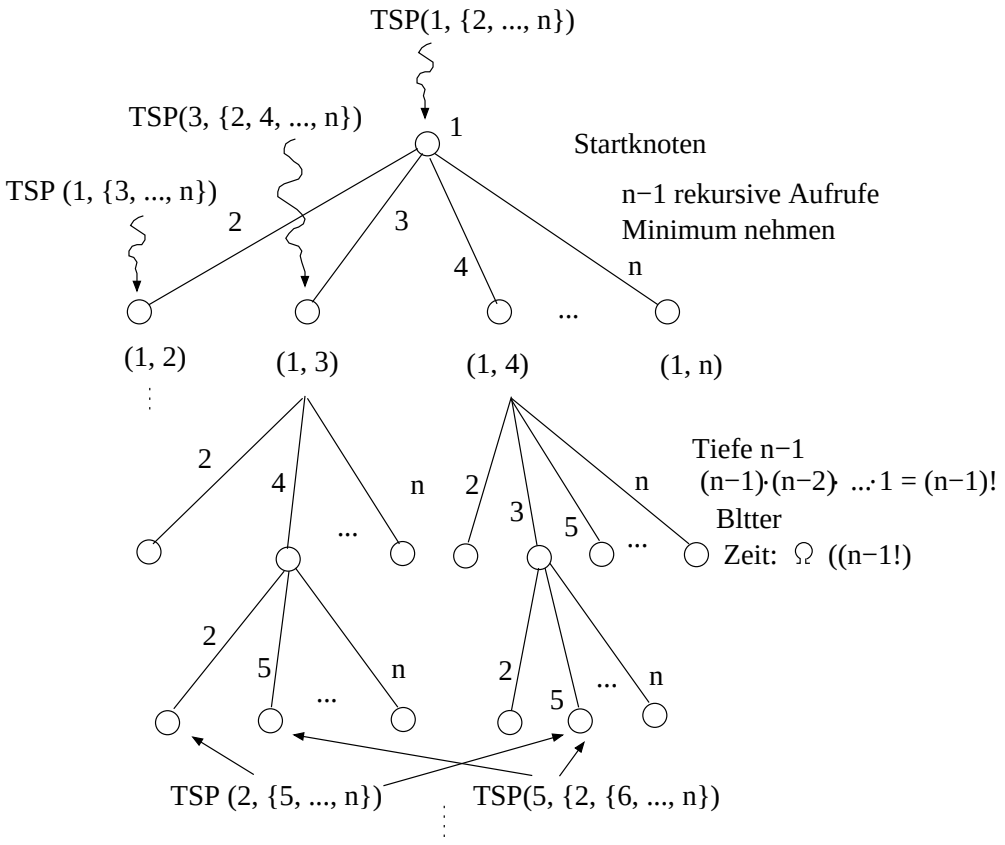
$$\begin{pmatrix} \infty & 10 & \infty & 20 \\ \infty & \infty & 9 & \infty \\ 6 & \infty & \infty & \infty \\ \infty & \infty & 9 & \infty \end{pmatrix} \qquad \begin{pmatrix} \infty & 10 & \infty & \infty \\ \infty & \infty & \infty & 10 \\ 6 & \infty & \infty & \infty \\ \infty & \infty & 9 & \infty \end{pmatrix}$$

Schranke $= 45$ Schranke $= 43$

Es ist wegen der Schranke keine bessere Lösung mehr möglich.


Aufgabe: Zeigen Sie für $M$ und $M'$ mit M' im Baum, dass $S_2(M') \geq S_(M), S_3(M') \geq S_3(M)$.

Das backtracking mit Verzweigen nach dem Vorkommen einer Kante ist praktisch wichtig und führt ja auch zum branch-and-bound. Jedoch, viel Beweisbares ist dort bisher nicht herausgekommen. Wir fangen mit einer anderen Art des backtracking an: Verzweigen nach dem nächsten Knoten, zu dem eine Rundreise gehen soll. Der Prozeduraufrufbaum hat dann folgende Struktur:




Aufrufe der Art: $TSP(k, \underbrace{\{i_a, \ldots, i_e\}}_{\subseteq\{2,\ldots,n\}})$

Wieviele verschiedene rekursive Aufrufe TSP(k, S) gibt es prinzipiell?

- Wähle k: n Möglichkeiten
- Wähle S: $\leq 2^{n-1}$ Möglichkeiten

$2^{n-1} \cdot n = 2^{n-1+logn} = 2^{O(n)}$ Wogegen $(n-1)! = 2^{\overbrace{\Omega(logn) \cdot n}^{\gg n}}$.

Wir tabellieren wieder die Ergebnisse der rekursiven Aufrufe, in der Reihenfolge, wobei $TSP(k, S) =$ kürzeste Reise:


$k \neq 1, 1 \notin S$

Zunächst $S = \emptyset$:

$$\begin{aligned} TSP(2, \emptyset) &= M(2,1) \quad // M = (M(u,v)) \text{ Eingangsmatrix} \\ TSP(n, \emptyset) &= M(n,1) \end{aligned}$$

Dann $|S| = 1$:

$$\begin{aligned} TSP(2, \{S\}) &= M(2,3) + TSP(3, S \setminus \{3\}) \\ &\vdots \\ TSP(2, \{n\}) &= M(2,n) + TSP(n, S \setminus \{n\}) \\ &\vdots \end{aligned}$$

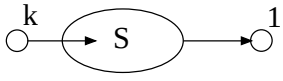
Dann $|S| = 2$

$$\begin{aligned} TSP(2, \{3,4\}) &= \min\{M(2,3) + TSP(3, \{4\}) + M(2,4) + TSP(4, \{3\})\} \\ &\vdots \end{aligned}$$

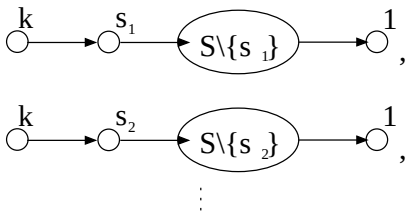
Allgemein:

$$TSP(k, S) = \min\{M(k,s) + TSP(s, S \setminus \{s\}) | s \in S\}$$

Also: Minimum über den Einstiegspunkt in S:




= Minimum von:




Alle Elemente aus S probieren.

## 10.8 Algorithmus (TSP mit dynamischem Programmieren)

Datenstruktur: array $TSP[1,\ldots n, \overbrace{0\ldots0}^{\text{n-1 Bits}}, \ldots \overbrace{1\ldots1}^{\text{n-1 Bits}}]$ of integer

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

Ergebnis ist $TSP(1,\{2,\ldots,n-1\}) = \min\{M(1,s) + TSP(s,\{2,\ldots,n-1\}\setminus)\}$.

Laufzeit: $O(n\cdot 2^n)$ Einträge im array TSP. Pro Eintrag das minimum ermitteln, also Zeit $O(n)$. Also $O(n^2\cdot 2^n)$.

Es ist

$$n^2\cdot 2^n = 2^{n+2logn} \leq 2^{n(1+\varepsilon)} = 2^{(1+\varepsilon)\cdot n} = (2(1+\varepsilon)!)^n \ll (n-1)!$$

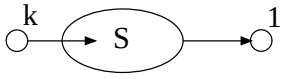
für $\varepsilon, \varepsilon' \geq 0$ geeignet.

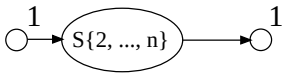
$2^\varepsilon \to 1$ für $\varepsilon \to 0$, da $2^0 = 1$.

Verwandt mit dem Problem des Handlungsreisenden ist das Problem des Hamiltonschen Kreises.

**Definition 10.3(Hamilton-Kreis):** *Sei $G = (V,E)$ ein ungerichteter Graph. Ein Hamilton-Kreis ist ein einfacher Kreis der Art $(1, v_1, v_2, \ldots, v_{n-1}, 1)$. (Also ist dies ein Kreis, in dem alle Knoten genau einmal auftreten.)*

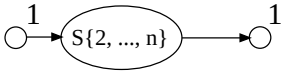
Mit dynamischem Programmieren

H(k, S) für 

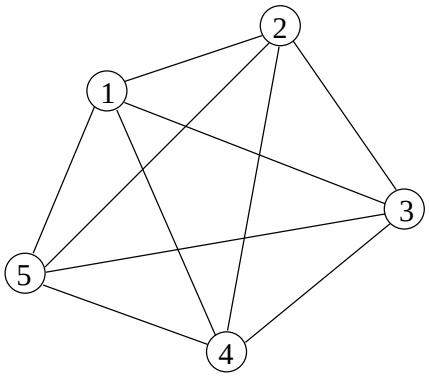
H(1, S) für 


in Zeit $O(n^2 \cdot 2^n)$ lösbar. Besser als $\Omega((n-1)!)$ druch einfaches backtracking.

**Definition 10.4(Eulerscher Kreis):** *Sei $G = (V, E)$ ein ungerichteter Graph. Ein Eulerscher Kreis ist ein geschlossener Weg, in dem jede Kante genau einmal vorkommt.*




Dann sollte $(1, 5, 4, 3, 2, 1, 4, 2, 5, 3, 1)$ ein Eulerscher Kreis sein.



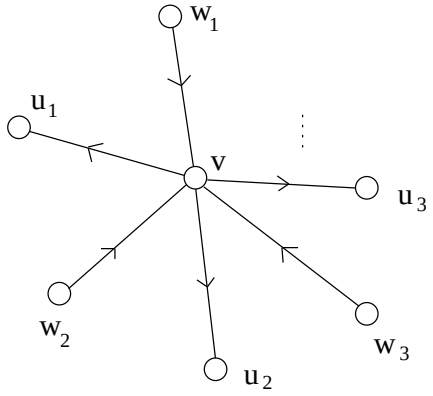

Scheint nicht zu gehen. Dynamisches Programmieren? $O(m \cdot n \cdot 2^m)$, $|V| = n, |E| = m$ sollte klappen. Es geht aber in polyninomialer Zeit und wir haben wieder:

- Hamilton-Kreis: polynomiale Zeit, nicht bekannt
- Eulerscher Kreis: polynomiale Zeit

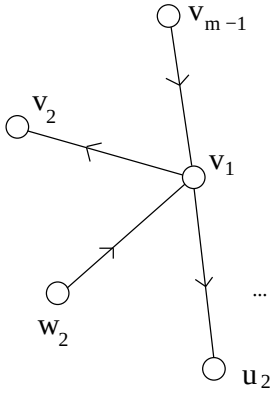
Dazu der

**Satz 10.4:** *Sei $G$ ohne Knoten vom Grad 0 (Grad(v) = # direkter Nachbarn), $G$ hat Eulerschen Kreis $\Longleftrightarrow$ $G$ zusammenhängend und Grad(v) = gerade für alle $v \in V$.*

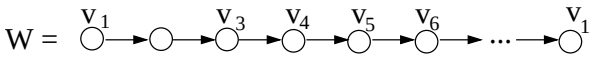
***Beweis.*** „$\Rightarrow$“ Sei also $G = (V, E), |V| = n, |E| = m$. Sei $K = (v_1, v_2, v_3, \ldots, v_m = v_1)$ ein Eulerscher Kreis von $G$. Betrachten wir einen beliebigen Knoten $v \neq v_1$ der etwa $k-$mal auf $K$ vorkommt, dann haben wir eine Situation wie




alle $u_i, w_i$ verschieden, also $Grad(v) = 2k$. Für $v = v_1$ haben wir



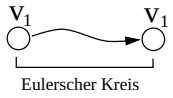

und $Grad(v_1)$ ist gerade
„$\Leftarrow$“ Das folgende ist eine Schlüsselbeobachtung für Graphen, die zusammenhängend sind und geraden Grad haben: Wir beginnen einen Weg an einem beliebigen Knoten $v_1$ und benutzen jede vorkommende Kante nur einmal. Irgendwann landen wir wieder an $v_1$:



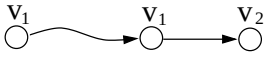

Ist etwa $v_3 = v_5 = u$, so hat $u$ mindestens 3, also $\geq 4$ Nachbarn. Wir können also von $v_5$ wieder durch eine neue kante weggehen. Erst, wenn wir bei $v_1$ gelandet sind, ist das nicht mehr sicher, und wir machen dort Schluss.
Nehmen wir nun $W$ aus $G$ heraus, haben alle Knoten wieder geraden Grad und wir können mit den Startknoten $v_1, v_2, \ldots$ so weitermachen und am Ende alles zu einem Eulerschen Kreis zusammensetzen. konkret sieht das so aus:

1. Gehe einen Weg $W$ wie oben und streiche die Kanten von W aus G. ($W$ gehört nicht direkt zum Eulerschen Kreis)

2. Nach Induktionsvoraussetzung haben wir einen Eulerschen Kreis auf dem Stück, das jetzt von $v_1$ erreichbar ist. Schreibe diesen Eulerschen Kreis hin. Wir bekommen:




3. Erweitere den Eulerschen Kreis auf $W$ zu




Mache von $v_2$ aus dasselbe wie bei $v_1$:


So geht es weiter bis zum Ende von $W$.

□

**Aufgabe:** Formulieren Sie mit dem Beweis oben einen (rekursiven) Algorithmus, der in $O(|V| + |E|)$ auf einen Eulerschen Kreis testet und im positiven Fall einen Eulerschen Kreis gibt.

Eine weitere Möglichkeit, die kombinatorische Suche zu gestalten, ist das Prinzip der lokalen Suche. Beim aussagenlogischen Erfüllbarkeitsproblem für KNF gestaltet es sich etwa folgendermaßen:

## 10.9 Algorithmus (Lokale Suche bei KNF)

Eingabe: $F$ in KNF

1. Wähle eine Belegung $a = (a_1, \ldots, a_n)$ der Variablen.
2. if $a(F) = 1$ return „$a(F) = 1$“
3. Wähle Klausel $C$ von $F$ mit $a(C) = 0$
4. Ändere $a$ so, dass $a(C) = 1$ ist, indem ein (oder mehrere) Werte von $a$ geändert werden.
5. Mache bei 1. weiter.

Wie Nicht-Erfüllbarkeit erkennen?

Eine Modifikation. 2 lokale Suchen, von $a = (0, \ldots, 0)$ und $b = (1, \ldots, 1)$.

Nun gilt: Jede Belegung unterscheidet sich auf $\leq \frac{\lceil n \rceil}{2}$ Positionen von $a$ ($\leq \frac{\lceil n \rceil}{2}$ Einsen) oder $b$ ($\geq \frac{\lceil n \rceil}{2}$ Einsen).

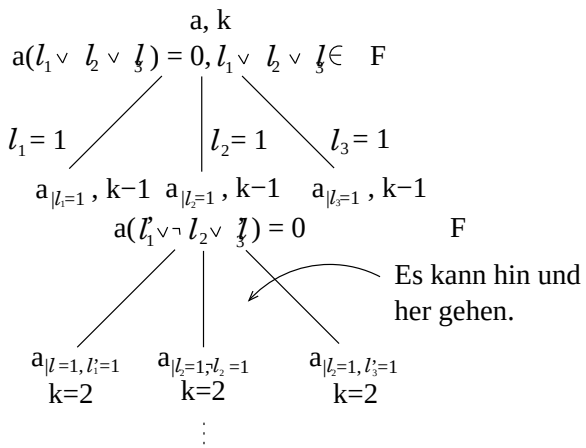
**Bezeichnung:** Für zwei Belegungen $a = (a_1, \ldots, a_n)$, $b = (b_1, \ldots, b_n)$ ist $D(a, b) = |\{i | 1 \leq i \leq n, a_i \neq b_i\}|$. Also die # Positionen, auf denen $a$ und $b$ verschieden sind, die Distanz von $a$ und $b$.

Für $F$ und Belegung $a$ gilt:

Es gibt $b$ mit $b(F) = 1$ und $D(a, b) \leq k \iff$

- $a(F) = 1$ oder
- Für jedes $C = l_1 \vee \cdots \vee l_m \in F$ mit $a(C) = 0$ gibt es ein $l_j$, so, dass für alle $a'(l_j) = 0$ und $a'$ sonst wie $a$ gilt: $D(b, a) \leq k - 1$.

Mit diesem Prinzip rekursiv für 3-KNF $F$:

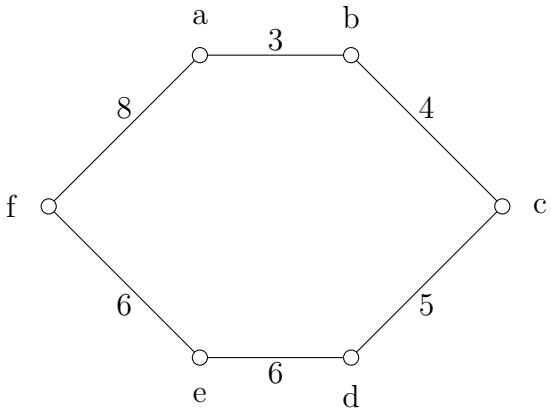



Tiefe k, Zeit: $O(n \cdot 3^k)$

Einmal rekursiv mit $F, a = (0, \ldots, 0), \frac{\lceil n \rceil}{2}$ noch einmal mit $F, b = (1, \ldots, 1), \frac{\lceil n \rceil}{2}$. Dann Zeit $O(n \cdot 3^{\frac{n}{2}} = O(\underbrace{1,7...}_{<2}{}^n)$. (3 wegen 3-KNF) Auf die Art kann bis zu $O(1,5^n)$ erreicht werden.

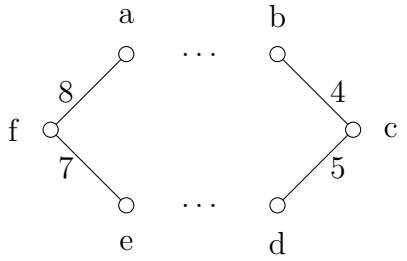
Die lokale Suche basiert auf einem Distanzbegriff zwischen möglichen Lösungen. Bei Belegungen ist dieser klar gegeben. Wie bei Rundreisen für das TSP? Wir betrachten hier nur den Fall, dass der zugrundeliegende Graph ungerichtet ist.

Haben nun also eine Rundreise wie



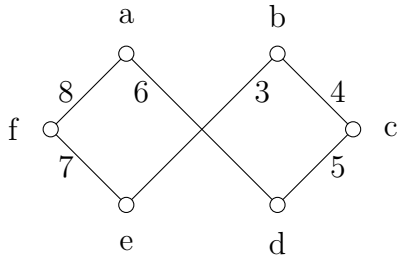
Kosten 33

gegeben. Duch Ändern einer Kante bekommen wir keine neue Rundreise hin. An zwei benachbarten Kanten können wir nichts ändern. Löschen wir einmal zwei nicht benachtbarte Kanten a—b, c—d




Wie können wir eine neue Rundreise zusammensetzen?

d—e (und a—b ) gibt die Alte.
Also d—a ( und b—e ). Das gibt:

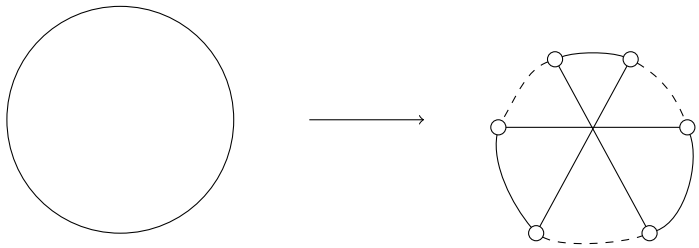


Kosten 32

Man kann in $O(n^2)$ Schritten testen, ob es so zu einer Verbesserung kommt. Ein solcher Verbesserungsschritt wird dann gemacht. Aber es gilt (leider) nicht:

Keine Verbesserung möglich $\Longleftrightarrow$ Minimale Rundreise gefunden.

Allgemein kann man auch drei Kanten löschen und den Rest zusammenbauen.



Man findet jedoch auch nicht unbedingt eine minimale Rundreise.

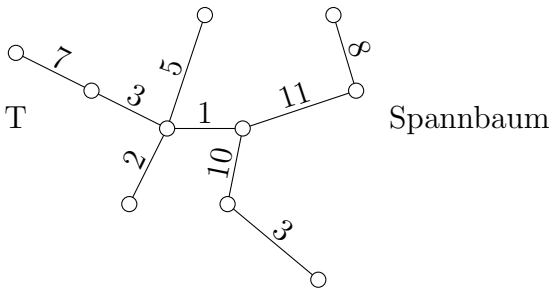
**Bemerkung:**

- Für all die Probleme, für die wir keine Polynomialzeit-Algorithmen angegeben haben, sind auch keine bekannt.
- Das bedeutet, für diese Probleme kann das weitgehend blinde Ausprobieren von (exponentiell) vielen Lösungsmöglichkeiten nicht vermieden werden.
- Es ist kein Beweis bekannt, dass es nicht doch in polynomialer Zeit geht. (Wird aber nicht erwartet)
- Die hier betrachteten Exponentialzeit-Probleme sind ineinander übersetzbar (Theoretische Informatik 2)

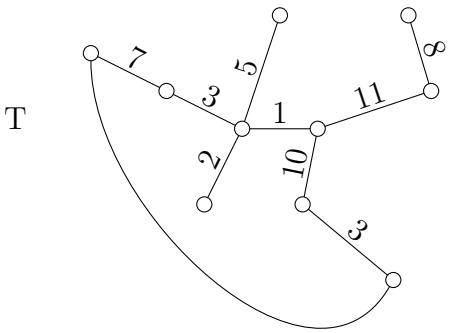
Die lokale Suche erlaubt es dagegen, in Polynomialzeit einen minimalen Spannbaum zu finden.

Erinnerung: Können minimalen Spannbaum in $O(|E|log|V|)$ finden (sogar $O(|E|log*|V|)$, wenn $E$ nach den Kosten vorsortiert ist).

Wir definieren zunächst den grundlegenden Transformationsschritt der lokalen Suche beim minimalen Spannbaum:




Wähle eine Kante $e$, die nicht im Baum ist:




Diese Kante induziert genau einen einfachen Kreis mit dem Spannbaum. Wir prüfen für jede Kante auf diesem Kreis, ob ihre Kosten > Kosten von $e$ sind. Haben wir eine solche Kante gefunden, löschen wir sie und bekommen einen Spannbaum mit geringeren Kosten.

**Satz 10.5:** *Haben wir einen nicht-minimalen Spannbaum, so gibt es immer eine Kante $e$, mit der oben angegebener Transformationsschritt zu einer Verbesserung führt.*

***Beweis.*** Sei also $T$ ein Spannbaum nicht minimal, sei $S$ ein minimaler Spannbaum, dann $S \neq T$. Wir betrachten einmal die Kantenmenge $S \setminus T = \{e_1, \ldots, e_k\}$.

Falls der oben angegebene Transformationsschritt mit $T$ und $e_1$, $T$ und $e_2, \ldots, T$ und $e_k$ jedesmal zu keiner Verringerung der Kosten führt, transformieren wir so:

$$
\begin{array}{lcl}
T_1 & := & T \cup \{e_1\} \\
R_1 & := & T_1 \setminus \{\text{Kante nicht aus S, auf Kreis durch } e_1\} \quad \text{// keine Verbesserung} \\
T_2 & := & R_1 \cup \{e_2\} \quad \text{// noch Annahme} \\
R_1 & := & T_2 \setminus \{\text{Kante nicht aus } S \text{ auf Kreis durch } e_2\} \\
 & & \quad \text{// Keine Verbesserung, da } e_1 \text{ nicht kleiner als alle Kanten auf} \\
 & & \quad \text{// dem vorherigen Kreis.} \\
 & \vdots & \\
T_k & := & R_{k-1} \cup \{e_k\} \\
R_k & := & T_k \setminus \{\text{Kante nicht aus } S \text{ auf Kreis durch } e_k\} \\
 & & \quad \text{// Keine Verbesserung, da vorher keine Verbesserungen.}
\end{array}
$$

Also, es sind die Kosten von $T$ minimal, im Widerspruch zur Annahme. □

Laufzeit eines Algorithmus mit Transformationsschritt:

- Maximal $|E|^2$ Schritte

- Pro Schritt $|E|$ Kanten ausprobieren
- Pro Kante $O(|V| + |E|)$

Insgesamt $O(|E|^4 + |V|)$.