

6 Einführung in die Komplexitätstheorie

In den vorherigen Kapiteln haben wir bereits einige theoretische Probleme kennengelernt. Wir haben insbesondere gesehen, dass sich einige Probleme schneller lösen (bzw. approximieren) lassen als andere. Außerdem scheint es Probleme zu geben, bei denen man schnell den Mut verliert, eine *exakte* Lösung in Polynomialzeit zu berechnen. Diese Probleme scheinen *schwerer* zu sein als andere. Was ist die *Schwere* eines Problems? Die Komplexitätstheorie befasst sich noch weitergehend mit der Frage nach der *Komplexität* von Problemen, dh. man versucht zu klassifizieren *wie* schwer Probleme sind.

In diesem Kapitel werden wir die Komplexitätsklassen P und NP einführen und wollen dann die Begriffe der NP -schwere und NP -vollständigkeit verstehen. Im Anschluss werden wir einige Beispiele NP -vollständiger Probleme studieren.

6.1 Einleitung: Cliquesproblem

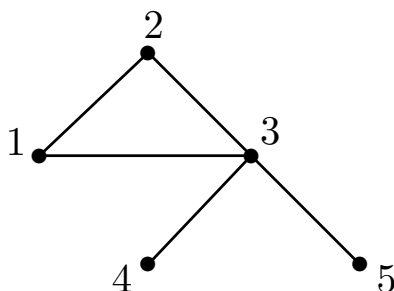
Als erstes Beispiel betrachten wir das sogenannte Cliquesproblem auf Graphen. Eine *Clique* ist eine Teilmenge $C \subseteq V$, wobei $\{u, v\} \in E$ für alle $u, v \in C$ mit $u \neq v$. Das Cliquesproblem können wir nun wie folgt angeben:

Gegeben: Ein ungerichteter Graph $G = (V, E)$ und eine Zahl $k \geq 1$.

Entscheide: Hat der Graph G eine Clique $C \subseteq V$ mit mindestens k Knoten (d.h. mit $|C| \geq k$)?

Dies ist ein sogenanntes *Entscheidungsproblem*, welches also als Antwort Ja oder Nein gibt.

Beispiel 6.1. Man betrachte den folgenden Graphen $G = (V, E)$ mit $V = \{1, 2, 3, 4, 5\}$ und $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}, \{3, 5\}\}$:



Hier ist $C = \{1, 2, 3\}$ eine Clique der Größe 3, aber $C = \{2, 3, 4\}$ ist keine Clique.

Offensichtlich löst der folgende Algorithmus das Problem exakt:

Algorithmus:

Teste für alle $V' \subseteq V$ mit $|V'| = k$, ob V' eine Clique bildet.

Wir bezeichnen mit $T(G, k)$ seine Laufzeit für eine Eingabe (G, k) .

Lemma 6.2. *Der obige Algorithmus hat im schlimmsten Fall mindestens exponentielle Laufzeit; d.h. $T(G, k) \geq 2^{|V|/2}$ für $k = |V|/2$.*

Beweis. Für $|V| = n$, $k = n/2$ und n gerade: Der Test für jedes V' erfordert $\Omega(k^2)$ Schritte. Es gibt $\binom{n}{n/2}$ viele k -elementige Teilmengen. Dann erhalten wir hier mit

$$\binom{n}{n/2} = \frac{n(n-1) \cdot (n/2+1)}{n/2(n/2-1) \cdot 1} \geq 2^{n/2}$$

mindestens eine exponentielle Laufzeit. □

Frage: Gibt es schnellere (bzw. insbesondere polynomielle) Algorithmen?

Definition 6.3. (i) Ein Alphabet Σ ist eine endliche Menge an Symbolen. Σ^* ist die Menge aller endlichen Wörter über Σ .

(ii) Für jedes $x \in \Sigma^*$ bezeichne $|x|$ die Länge von x .

Bemerkung 6.4. Jeder Algorithmus arbeitet auf einem Alphabet (d.h. seine Eingaben und Ausgaben sind Elemente aus Σ^*). Für einen Algorithmus A , bezeichnet $A(x)$ die Ausgabe zur Eingabe $x \in \Sigma^*$.

Die Laufzeit eines Algorithmus A zur Eingabe x (gegeben durch die Anzahl der ausgeführten Operationen) wird durch $T_A(x)$ bezeichnet. Die Worst Case Laufzeit zu Eingaben der Größe n ist

$$T_A(n) = \max\{T_A(x) | x \in \Sigma^*, |x| = n\}.$$

Definition 6.5. (i) Eine Teilmenge $L \subseteq \Sigma^*$ heißt Sprache.

(ii) Ein Entscheidungsproblem ist ein Tripel (L, U, Σ) wobei L, U Sprachen sind und $L \subseteq U$.

(iii) Ein Algorithmus A löst das Entscheidungsproblem (L, U, Σ) (oder auch A entscheidet die Sprache L bezüglich U), wenn für alle $x \in U$ gilt:

- $x \in L$ impliziert $A(x) = 1$,
- $x \notin L$ impliziert $A(x) = 0$.

Häufig ist $U = \Sigma^*$. Dann sagt man, dass A die Sprache $L \subseteq \Sigma^*$ entscheidet. Das Cliquesproblem etwa können wir als Sprache (Menge von Wörtern) wie folgt darstellen:

$$\begin{aligned} \text{CLIQUE} = \{ & u\#v \mid u \in \{0,1\}^* \text{ stellt Adjazenzmatrix von } G \text{ dar,} \\ & v \in \{0,1\}^* \text{ stellt } k \text{ dar,} \\ & G \text{ hat Clique mit } k \text{ Knoten } \}. \end{aligned}$$

Hierbei ist $\Sigma^* = \{0, 1\}^*$ die Menge aller endlichen Wörter (inklusive dem leeren Wort ϵ) über dem Alphabet $\Sigma = \{0, 1\}$.

Bemerkung 6.6. u ist ein Wort der Länge $\mathcal{O}(|V|^2)$, v ist ein Wort der Länge $\mathcal{O}(\log |V|)$.

6.2 Komplexitätsklassen P, NP

Definition 6.7. Die Menge aller polynomiell entscheidbaren Sprachen bzw. Entscheidungsproblemen wird mit P bezeichnet:

$$P = \{L \subseteq \Sigma^* \mid \text{es existiert ein Algorithmus } A, \text{ der } L \text{ in Laufzeit } T_A(n) = \mathcal{O}(n^d) \text{ mit } d = \mathcal{O}(1) \text{ entscheidet}\}.$$

Neben P gibt es unter anderem auch die Klasse NP . Diese wird häufig über *nicht-deterministische Algorithmen* eingeführt (bei denen es zu jeder Eingabe eine endliche Menge von Rechenwegen gibt).

Definition 6.8. Ein nicht-deterministischer Algorithmus entscheidet eine Sprache L ,

- wenn es für jedes Wort $w \in L$ mindestens einen Rechenweg gibt, der w akzeptiert,
- wenn jeder akzeptierende Rechenweg in polynomieller Zeit läuft und
- wenn für jedes Wort $w \notin L$ jeder Rechenweg w ablehnt.

Die Anzahl der Rechenwege kann dabei exponentiell von der Eingabe abhängen.

Die Klasse NP ist nun genau die Menge aller Sprachen, die durch einen nicht-deterministischen Algorithmus in Polynomialzeit entschieden werden können. Wir bedienen uns hier allerdings einer alternativen Definition der Klasse NP über *polynomielle Verifizier*. Die so definierten Klassen von Problemen sind aber gleich.

Definition 6.9. Es sei $L \subseteq \Sigma^*$. Ein (deterministischer) Algorithmus, der auf $\Sigma^* \times \Sigma^*$ arbeitet, heißt ein Verifizierer für L , wenn

$$L = \{x \in \Sigma^* \mid \exists c \in \Sigma^* \text{ mit } A(x, c) = 1\}.$$

Hierbei nennt man ein Wort c mit $A(x, c) = 1$ ein Zertifikat für x (bzw. für die Aussage $x \in L$).

Definition 6.10. Ein Algorithmus A ist ein polynomieller Verifizierer für eine Sprache L , wenn eine Konstante d existiert, so dass es für jedes $x \in L$ ein Zertifikat c gibt mit $T_A(x, c) = \mathcal{O}(|x|^d)$.

Definition 6.11. Die Klasse aller polynomiell verifizierbaren Sprachen wird mit NP bezeichnet; d.h.

$$NP = \{L \subseteq \Sigma^* \mid \text{es existiert ein polynomieller Verifizierer für } L\}.$$

Satz 6.12. *Es gilt $P \subseteq NP$.*

Beweis. Zu einem Entscheidungsproblem L aus P gibt es einen Algorithmus A , der L entscheidet. Diesen Algorithmus kann man als Verifizierer auffassen, der die zweite Eingabe c einfach ignoriert. Damit ist L auch in NP . $\square$

Nun können wir auf das Cliquesproblem zurückkommen und die folgende Aussage beweisen.

Lemma 6.13. *Das Entscheidungsproblem k -Clique ist in NP .*

Beweis. Durch Angabe eines polynomiellen Verifizierers für k -Clique. Der Algorithmus bekommt neben der Eingabe (einen Graph $G = (V, E)$ und eine Zahl k) ein Zertifikat (u.z. eine Teilmenge $C \subseteq V$). Der Algorithmus testet dann, ob C eine Clique der Größe $\geq k$ ist. Der Test geht wie folgt:

- (1) prüfe für alle zweielementigen Teilmengen $\{u, v\} \subseteq C$, ob $\{u, v\} \in E$ liegt.
- (2) gebe Ausgabe **Ja** aus, wenn jeder Test in (1) positiv ausfiel und $|C| \geq k$ ist. Ansonsten Ausgabe **Nein**.

Schritt (1) geht in Zeit $\mathcal{O}(|C|^2) = \mathcal{O}(|V|^2)$ und die Kardinalitätsprüfung in (2) in $\mathcal{O}(|C|) = \mathcal{O}(|V|)$ Zeit. Da die Eingabelänge $\Omega(|V| + |E|)$ ist, ist die Laufzeit des Algorithmus polynomiell. $\square$

Nachdem wir nun gesehen haben, wie die Klassen P und NP definiert haben, können wir den Begriff der *Schwere* genauer beleuchten. Dafür ist es notwendig zu verstehen, wann ein Problem auf ein anderes Problem *reduziert* werden kann. Wir benötigen die folgende Definition.

Definition 6.14. *Es seien $L_1 \subseteq \Sigma_1^*$, $L_2 \subseteq \Sigma_2^*$ zwei Sprachen.*

- (i) *Eine Funktion $f : \Sigma_1^* \rightarrow \Sigma_2^*$ mit der Eigenschaft*

$$w \in L_1 \Leftrightarrow f(w) \in L_2 \quad \forall w \in \Sigma_1^*$$

heißt Transformation von L_1 auf L_2 .

- (ii) *Eine solche Transformation heißt polynomielle Transformation, wenn es ein Polynom p und einen Algorithmus A gibt, der für alle $w \in \Sigma_1^*$ die Ausgabe $f(w)$ der Länge $\leq p(|w|)$ in (polynomieller) Zeit $T_A(w) \leq p(|w|)$ berechnet.*

Dann heißt L_1 polynomiell reduzierbar auf L_2 . Als Notation verwenden wir $L_1 \leq L_2$.

Endlich können wir den Begriff der *NP-Schwere* bzw. *NP-Vollständigkeit* formal fassen:

Definition 6.15. (i) *Eine Sprache L_0 heißt NP-schwer, g.d.w. $\forall L \in NP : L \leq L_0$.*

- (ii) *Eine Sprache L_0 heißt NP-vollständig, g.d.w. $L_0 \in NP$ und $\forall L \in NP$ gilt: $L \leq L_0$.*

Eine fundamentale Eigenschaft der NP -vollständigkeit ist die folgende:

Satz 6.16. *Es sei L_0 NP -vollständig. Dann gilt $P = NP$, g.d.w. $L_0 \in P$.*

Beweis. $\Rightarrow$: Es sei $P = NP$. Zeige dann, dass $L_0 \in P$.

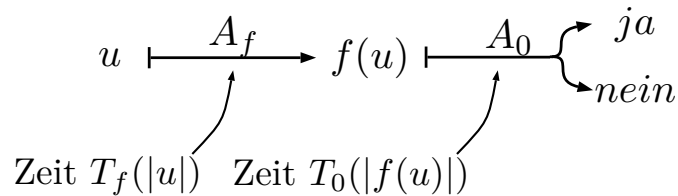
L_0 ist NP -vollständig. Dann gilt insbesondere $L_0 \in NP$. Wegen $P = NP$ gilt dann $L_0 \in P$.

$\Leftarrow$: Es sei $L_0 \in P$. Zeige hier, dass $P = NP$.

Da $L_0 \in P$ ist, gibt es einen Algorithmus A_0 , der L_0 in Polynomzeit T_0 entscheidet. Es sei nun $L \in NP$ beliebig gewählt. **Zeige:** $L \in P$.

Da L_0 NP -vollständig ist und $L \in NP$, gilt $L \leq L_0$. Sei also A_f ein Algorithmus, der die polynomielle Transformation f von L auf L_0 in Polynomzeit T_f berechnet mit $u \in L \Leftrightarrow f(u) \in L_0$.

Wir konstruieren A aus A_0 und A_f durch Hintereinanderschalten; siehe folgendes Bild:



Dann akzeptiert A ein Wort u , g.d.w. A_0 das Wort $f(u)$ akzeptiert. Dies ist äquivalent zu $f(u) \in L_0$ bzw. $u \in L$. D.h. Algorithmus A entscheidet L mit Gesamtlaufzeit $\leq T_f(|u|) + T_0(|f(u)|) \leq T(|u|)$ für ein Polynom T . Hierbei verwende, dass $|f(u)| \leq \text{poly}(|u|)$ und dass die Komposition von Polynomen wieder ein Polynom ergibt. $\square$

Konsequenz von Satz 6.16: Wenn ein NP -vollständiges Entscheidungsproblem in P liegt, dann sind alle NP Entscheidungsprobleme in P .

Die gleiche Aussage gilt auch für ein NP -schweres Problem.

Offenes Problem der Komplexitätstheorie:

Gilt $P = NP$ oder $P \neq NP$?

Dies ist eines der wichtigsten offenen Probleme der Informatik. Es wurde vom Clay Mathematics Institute in die Liste der Millennium-Probleme aufgenommen; mit einem Preisgeld von 1-Million Dollar: www.claymath.org.

Bemerkung 6.17. Wenn $L_1 \leq L_2$ und $L_2 \leq L_3$, dann gilt auch $L_1 \leq L_3$.

Beweis. siehe Übung. $\square$

Korollar 6.18. Ist L_0 NP -vollständig, $L_0 \leq L_1$ und $L_1 \in NP$, so ist auch L_1 NP -vollständig.

Beweis. Zeige die folgende Behauptung: $\forall L \in NP$ gilt $L \leq L_1$.

Sei dazu $L \in NP$ beliebig gewählt. Zeige dann die Aussage $L \leq L_1$. Wegen der NP -vollständigkeit von L_0 und da $L \in NP$ ist, gilt $L \leq L_0$. Nach Voraussetzung gilt zusätzlich $L_0 \leq L_1$. Da die Relation $\leq$ transitiv ist (siehe Bemerkung 6.17), gilt dann: $L \leq L_1$.

Da $L_1 \in NP$ wegen der Voraussetzung, ist daher L_1 auch NP -vollständig. $\square$

Das obige Ergebnis in Korollar 6.18 ermöglicht uns, NP -vollständige Probleme aus bekannten NP -vollständigen Problemen zu konstruieren (bzw. deren NP -vollständigkeit zu beweisen). Für diese Entscheidungsprobleme gibt es keine polynomiellen Algorithmen, außer $P = NP$.

6.3 Definition von P und NP per Turingmaschine

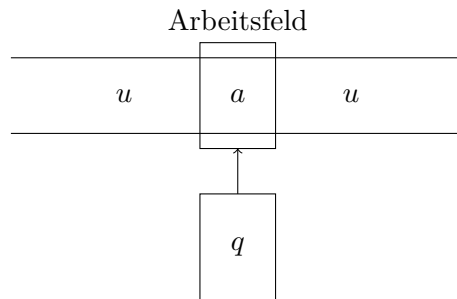
Eine nicht-deterministische Turingmaschine (NDTM) hat die Form:

$$M = (Q, \Gamma, q_0, \delta, F).$$

Hierbei beschreibt Q die nicht-leere endliche Zustandsmenge, Γ das Arbeitsalphabet, wobei $\Gamma \supset \Sigma$ gilt (dabei ist Σ das Eingabealphabet). Die Zustandsmenge enthält den Anfangszustand $q_0 \in Q$ und die Endzustandsmenge $F \subset Q$. Zuletzt beschreibt δ die Übergangsfunktion:

$$\delta : Q \times \Gamma \longrightarrow 2^{Q \times \Gamma \times \{-1, 0, 1\}}.$$

Es gilt, dass $|\delta(q, a)| = 0, 1, 2, \dots$ sein kann, man also mehrere mögliche Folgezustände haben kann.



Ein Übergang $\delta(q, a) \rightarrow (q', a', -1/1/0)$ besagt: im Zustand q mit a auf dem Arbeitsfeld (AF) drucke a' auf AF, bewege den Schreiblesekopf (bzw. das AF) nach links, rechts oder bleibe stehen und gehe danach in Zustand q' über.

Die möglichen Berechnungen einer NDTM wird über Konfigurationen (die die Bandinschrift plus den Zustand enthalten) beschrieben, wobei eine Konfiguration c wie folgt definiert ist:

$$c \in \Gamma^* \times Q \times \Gamma \times \Gamma^*$$

$$u \quad q \quad a \quad v,$$

wobei u und v Wörter sind, q ein Zustand und a der Buchstabe auf dem AF ist.

Für $u = u_0b$, v sei die Folgekonfiguration von $uqav$ definiert durch:

$$\begin{array}{ll} u_0q'ba'v & \text{falls } \delta(q, a) = (a', 1, q') \\ u_0ba'q'v & \text{falls } \delta(q, a) = (a', -1, q') \\ u_0bq'a'v & \text{falls } \delta(q, a) = (a', 0, q') \end{array}$$

Als Notation verwenden wir $C \vdash C'$; dabei ist C' die Folgekonfiguration von C .

Berechnung zu Wörtern bzw. Eingaben: Hierbei ist die Startkonfiguration durch q_0bw gegeben, wobei b ein Sonderzeichen 'Blank' (bzw. ein Leerzeichen) darstellt mit $b \in \Gamma \setminus \Sigma$. Eine Konfiguration C nennt man Stopkonfiguration, falls keine Folgekonfiguration von C existiert. Eine Berechnung zu einer Eingabe $w \in \Sigma^*$ kann dann durch eine Folge $C_0, \dots, C_k$ von Konfigurationen dargestellt werden, wobei C_0 die Startkonfiguration zu einer Eingabe w ist, $C_i \vdash C_{i+1}$, für alle $i < k$ gilt und C_k eine Stopkonfiguration ist. Eine Berechnung heißt genau dann akzeptierend, wenn der Zustand von C_k in F liegt.

Sei $L \subseteq \Sigma^*$ und $T : \mathbb{N} \rightarrow \mathbb{N}$. M akzeptiert L [in nicht deterministischer Zeit T] genau dann wenn $\forall w \in \Sigma^* : (w \in L \iff \exists \text{ akzeptierende Berechnung von } M \text{ zu } w \text{ [mit Länge } \leq T(|w|)])$.

Damit können wir die Zugehörigkeit zu der Komplexitätsklasse NP nun definieren.

Definition 6.19. Eine Sprache $L \subset \Sigma^*$ ist in NP $\iff \exists$ eine NDTM M über Alphabet $\Gamma \supset \Sigma$ und ein Polynom T , so dass gilt: M akzeptiert L in nicht deterministischer Zeit T .

Deterministische Turingmaschine (DTM) Falls $|\delta(q, a)| \leq 1 \ \forall (q, a) \in Q \times \Gamma$, es also höchstens einen gültigen Übergang aus jedem Zustand gibt, nennt man eine Turingmaschine *deterministisch*. Die Berechnung der DTM ist analog zu NDTMs über Konfigurationen definiert.

Eine DTM M entscheidet L [in deterministischer Zeit T] genau dann wenn für alle Wörter $w \in \Sigma^*$ die Berechnung von w mit einer Stopkonfiguration endet [und Länge $\leq T(|w|)$ hat]. Dabei liegt der Zustand der Stopkonfigurationen in F , genau dann wenn w in L liegt.

Die Zugehörigkeit zu der Komplexitätsklasse P wird nun analog definiert.

Definition 6.20. $L \subset \Sigma^* \in P \iff \exists$ eine DTM M über Alphabet $\Gamma \supset \Sigma$ und ein Polynom T , so dass M die Sprache L in Zeit T entscheidet.

Nun bleibt zu zeigen, dass beide gezeigten Definitionen der Komplexitätsklasse NP äquivalent sind, sie also die gleiche Klasse beschreiben.

Satz 6.21. Die Klassen der Sprachen bezüglich Verifizierer und NDTM sind äquivalent und beschreiben beide die gleiche Klasse NP.

Beweis. " $\supset$ " : Sei N eine NDTM, die eine Sprache L in Zeit n^c für eine Konstante c entscheidet. Für jedes $x \in L$ gibt es eine Folge nicht-deterministischer Übergänge,

so dass N bei Eingabe x einen akzeptierenden Zustand q_{acc} erreicht. Diese Folge kann durch einen String u der Länge n^c über einem Alphabet σ' mit $O(1)$ vielen Buchstaben repräsentiert werden (hierbei hängt die Kardinalität von σ' von der maximalen Anzahl von Verzweigungen (d.h. $|\delta(q, a)|$) ab. Dann folgt direkt, dass es eine DTM M gibt, die bei Eingabe $\langle x, u \rangle$ genau dieselben Berechnungen durchführt, also die gleichen Übergänge wie N macht. Für die Eingabe $x \in L$ folgt also auch, dass M einen akzeptierenden Zustand erreichen muss. Dann ist M also auch ein Verifizierer für L .

” $\subset$ ” : Sei $L \subseteq \Sigma^*$ gegeben und sei M ein Verifizierer für L . Sei $p : \mathbb{N} \rightarrow \mathbb{N}$ ein Polynom, das die Länge der Zertifikate angibt. Sei n^c die Laufzeit der DTM M , wobei c wieder eine Konstante ist. Eine NDTM N , die w in Polynomzeit entscheidet, kann bei Eingabe $x \in \Sigma^*$ wie folgt vorgehen:

Phase (1): Nutze nicht-deterministische Entscheidungen um einen beliebigen String u der Länge $\text{poly}(|x|)$, also polynomiell in $|x|$, auf ein Arbeitsband zu schreiben.

Phase (2): Simuliere M bei Eingabe $\langle x, u \rangle$. Dann folgt direkt, dass N in Polynomzeit entscheidet, da M diese Eingabe in Polynomzeit verifizieren kann. $\square$

6.4 Satisfiability Problem (SAT)

Der Kanadier Stephen A. Cook begründete 1971 die Klasse der NP -vollständigen Probleme in NP , indem er zeigte, dass ein NP -vollständiges Problem existiert. Dieses Problem ist das sogenannte Satisfiability Problem (SAT). Leonid Levin hatte einen vergleichbaren Satz 1973 veröffentlicht.

Satisfiability Problem (SAT)

Das Satisfiability Problem (SAT) formalisiert die Frage nach der Erfüllbarkeit eines aussagenlogischen Ausdrucks. Aussagenlogische Ausdrücke sind zusammengesetzt aus Variablen und den *elementaren Operatoren/Junktoren* $\wedge$ (*und*), $\vee$ (*oder*) und $\neg$ (*nicht*). Die Variablen sind Boolesche Variablen und wir notieren sie als x_i , dh. $x_0, x_1, x_{10}, x_{11}, \dots$ und so weiter. Für jede Variable x_i existieren zwei *Literale*. Diese sind das *positive Literal* x_i und das *negative Literal* $\neg x_i$. Aus den Literalen setzt man mit dem elementaren Operator $\vee$ (*oder*) sogenannte *Klauseln* zusammen, dh. disjunktive Ausdrücke der Form $(y_1 \vee \dots \vee y_k)$ mit Literalen $y_1, \dots, y_k$.

Ein Ausdruck $\alpha = C_1 \wedge \dots \wedge C_m$ mit Klauseln $C_1, \dots, C_m$ heißt Ausdruck in *Konjunktiver Normalform* (KNF). Ein Ausdruck α heißt *erfüllbar*, g.d.w. eine Belegung ϕ der Variablen in α mit *true*, *false* existiert, so dass α unter der Belegung ϕ zu *true* ausgewertet wird (siehe Beispiel 6.23).

Nun können wir das SAT-Problem als Sprache wie folgt angeben:

$$\text{SAT} = \{ \alpha \mid \alpha \text{ Boolescher Ausdruck in KNF, } \alpha \text{ erfüllbar} \}$$

Bemerkung 6.22. $\text{SAT} \subset \{x, 1, 0, \vee, \wedge, \neg, (,)\}^*$.

Beispiel 6.23. Betrachte $\alpha = (x_1 \vee x_{10}) \wedge (x_1 \vee \neg x_{10}) \wedge (x_{10})$. Wir erhalten die folgende Wahrheitstabelle:

$x1$	$x10$	α
<i>false</i>	<i>false</i>	<i>false</i>
<i>true</i>	<i>false</i>	<i>false</i>
<i>false</i>	<i>true</i>	<i>false</i>
<i>true</i>	<i>true</i>	<i>true</i>

Die Belegung $\psi(x1) = \psi(x10) = \text{true}$ ist eine erfüllende Belegung.

Satz 6.24. (Cook, 1971/Levin, 1973): SAT ist NP-vollständig.

Beweis. $\text{SAT} \in \text{NP}$ Wir skizzieren dies für eine entsprechende nichtdeterministische Turingmaschine mit den Zwischenergebnissen auf dem Turingband in Abb. 6.1. Salopp gesagt „raten“ wir nichtdeterministisch eine Belegung der Variablen und prüfen dann, ob sie den Ausdruck erfüllt.

- α
(1) $\downarrow \text{det.}$
 $\alpha \# \text{Liste der auftretenden Variablen}$
- (2) $\downarrow \text{nichtdet.}$
 $\alpha \# \text{Belegung der Variablen durch wahr/falsch:}$
 $x101 \rightarrow T101 \text{ oder } F101.$
- (3) $\downarrow \text{det.}$
 $\alpha \# \text{Wert von } \alpha$
- (4) $\downarrow \text{det.}$
 F , wenn $\alpha = \text{wahr}$; dummy-Zustand, falls $\alpha = \text{falsch}$.

Abb. 6.1: $\text{SAT} \in \text{NP}$: Alle vier Schritte sind in polynomieller Zeit ausführbar.

$L \in \text{NP} \implies L \leq \text{SAT}$ Sei $\mathfrak{M}$ eine NDTM und akzeptiere L in nichtdeterministischer Zeit T (für ein geeignetes Polynom T). Wir suchen eine Transformation $u \rightarrow \alpha_u$, die polynomzeitberechenbar ist mit $u \in L \iff \alpha_u \in \text{SAT}$.

Idee: α_u wird so beschrieben, dass $\mathfrak{M}$ für u in genau $T(u)$ Schritten eine akzeptierende Stoppkonfiguration erreicht. Sei also

$$\mathfrak{M} = (Q, \Gamma, q_0, \delta, F),$$

wobei $Q = \{q_0, \dots, q_s\}$, $F = \{q_r, \dots, q_s\}$, $\Gamma = \{a_0, \dots, a_m\}$, $a_0 = \flat$. Ohne Einschränkung der Allgemeinheit gelte: Akzeptiert $\mathfrak{M}$ das Wort $u = a_{j_1} \dots a_{j_{|u|}}$ mit einer Berechnung der Länge *höchstens* $T(|u|)$, so akzeptiere $\mathfrak{M}$ das Wort mit einer Berechnung der Länge *genau gleich* $T(|u|)$.

Wir stellen δ durch eine Folge von Zeilen (Fünftupeln) $z_1, \dots, z_\ell$ der Form $qaq'a'\beta$ dar (für $(q', a', \beta) \in \delta(q, a)$). O.B.d.A betrachten wir nur Turingmaschinen, die nicht links vom Feld 0 arbeiten. Der zu konstruierende Ausdruck in KNF α enthält Variablen, deren Bedeutung in Tabelle 6.1 zusammengefasst ist.

Variablen in α_k	Bedeutung (wahr, falls...)
z_{tk}	nach t Schritten wird Zustand q_k erreicht
s_{ti}	nach t Schritten ist Feld Nr. i das Arbeitsfeld
b_{tl}	nach t Schritten wird Zeile z_l ausgeführt
a_{tij}	nach t Schritten steht auf Feld Nr. i der Buchstabe a_j ,

Hierbei ist $0 \leq t, i \leq T(|u|)$, $0 \leq k \leq S$, $1 \leq l \leq \varrho$, $0 \leq j \leq m$.

Tab. 6.1: Variablen in α_k

Die Anzahl der Variablen ist nach oben durch $c \cdot T(|u|)^2$ beschränkt, wobei c eine von der Turingmaschine abhängige Konstante ist.

Wir bauen nun einen Ausdruck α_u auf mit

$$\alpha_u = \alpha_{\text{Anfang}} \wedge \alpha_{\text{Ende}} \wedge \alpha_{\text{Eindeutig}} \wedge \alpha_{\text{Übergang}}.$$

Das Ziel ist, dass erfüllende Belegungen von α_u den akzeptierenden Berechnungen in Zeit $T(|u|)$ von $\mathfrak{M}$ für $u = a_{j_1} \dots a_{j_{|u|}}$ entsprechen sollen. Wir setzen

$$\alpha_{\text{Anfang}} = z_{00} \wedge s_{00} \wedge a_{000} \wedge a_{01j_1} \wedge a_{02j_2} \wedge \dots \wedge a_{0|u|j_{|u|}} \wedge a_{0(|u|+1)0} \wedge \dots \wedge a_{0T(|u|)0}$$

$$\alpha_{\text{Ende}} = z_{T(|u|)r} \vee \dots \vee z_{T(|u|)s}$$

$$\begin{aligned} \alpha_{\text{Eindeutig}} = & \bigwedge_{0 \leq t \leq T(|u|)} \left(\text{„genau ein Zustand“} \right. \\ & \wedge \text{„genau ein Arbeitsfeld“} \\ & \left. \wedge \text{„für alle Felder } i = 0, \dots, T(|u|) \text{ genau ein Buchstabe“} \right) \end{aligned}$$

Den ersten Teil, „genau ein Zustand“, schreiben wir als

$$(z_{t0} \vee \dots \vee z_{ts}) \wedge \bigwedge_{i \neq j} \underbrace{\neg(z_{ti} \wedge z_{tj})}_{(\neg z_{ti} \vee \neg z_{tj})},$$

die anderen Bedingungen analog. Wir setzen

$$\alpha_{\text{Übergang}} = \bigwedge_{0 \leq t \leq T(|u|)} \alpha_t,$$

wobei α_t Folgendes modelliert:

- Buchstaben auf nicht-Arbeitsfeldern bleiben unverändert
- Falls das Arbeitsfeld Nr. i hat und die l . Zeile $q_{k_l} a_{j_l} q_{\tilde{k}_l} a_{\tilde{j}_l} \beta_l$ ausgeführt wird, gilt:
 - a) nach t Schritten wird q_{k_l} erreicht
 - b) nach t Schritten steht auf Feld i der Buchstabe a_{j_l}
 - c) nach $t+1$ Schritten wird $q_{\tilde{k}_l}$ erreicht

- d) nach $t + 1$ Schritten steht auf Feld i Buchstabe $a_{\tilde{j}_l}$
- e) nach $t + 1$ Schritten hat das Arbeitsfeld die Nummer $i + \beta_l$.

Wir erhalten insgesamt:

$$\alpha_t = \bigwedge_{0 \leq i \leq T(|u|)} \left(\bigwedge_{0 \leq j \leq m} ((\neg s_{ti}) \wedge a_{tij}) \rightarrow a_{(t+1)ij} \right) \\ \wedge \bigwedge_{1 \leq l \leq \varrho} ((s_{ti} \wedge b_{tl}) \rightarrow (z_{tk_l} \wedge a_{tj_l} \wedge z_{(t+1)\tilde{k}_l} \wedge a_{(t+1)i\tilde{j}_l} \wedge s_{(t+1)(i+\beta_l)}))$$

Insgesamt kann man $\alpha_u = \alpha_{\text{Anfang}} \wedge \alpha_{\text{Ende}} \wedge \alpha_{\text{Eindeutig}} \wedge \alpha_{\text{übergang}}$ in KNF darstellen. Wir müssen noch zeigen:

- a) die Transformation $u \mapsto \alpha_u$ ist polynomiell, und
- b) es existiert eine akzeptierende Berechnung von $\mathfrak{M}$ zu u der Länge $T(|u|)$ dann und nur dann, wenn α_u erfüllbar ist.

a) Wir bestimmen die Anzahl bzw. Häufigkeit der Variablen in α_u :

- in α_{Anfang} : $T(n) + 3$
- in α_{Ende} : $\leq s + 1 = |Q|$
- in $\alpha_{\text{Eindeutig}}$: $\leq (T(n) + 1)(s + 1 + 2(s + 1)^2 + (T(n) + 1)^2 + (T(n) + 1)(m + 1)^2 + (\varrho + 1)^2) \leq c(T(n))^3$
- in $\alpha_{\text{übergang}}$: $(T(n) + 1)^2((3(m + 1)) + (15(\varrho + 1))) \leq c'T(n)^2$
- insgesamt: $\leq dT(n)^3$

(für geeignete, von $\mathfrak{M}$ abhängige Konstanten c, c', d). Die Länge einer Variablen ist nach oben durch $\bar{c} \log T(n)$ beschränkt (für geeignetes $\bar{c}$), also ist die Länge von α_u durch $\bar{d}T(n)^3 \log T(n)$ beschränkt, also polynomiell in $n = |u|$.

- b) $\Rightarrow$ Sei $C_0, \dots, C_{T(|u|)}$ eine akzeptierende Berechnung. Dann liefert das eine Belegung der Variablen von α_u so, dass α_u den Wert wahr erhält, und zwar:

$$\begin{aligned} \text{Wert}(z_{tk}) &= \text{wahr} \iff C_t \text{ hat Zustand } q_k \\ \text{Wert}(s_{ti}) &= \text{wahr} \iff \text{in } C_t \text{ hat das Arbeitsfeld die Nr. } i \\ &\vdots \end{aligned}$$

$\Leftarrow$ Betrachte eine erfüllende Belegung von α_u . Wegen $\alpha_{\text{Eindeutig}} = \text{wahr}$ existiert zu jedem Zeitpunkt t genau ein k mit $z_{tk} = \text{wahr}$. Bezeichne diese k mit $k(t)$. Analog definiere $i(t), l(t), j(t, i)$. Für alle Zeitpunkte t wird eine Konfiguration C_t eindeutig definiert durch den Zustand $q_{k(t)}$, Arbeitsfeld Nr. $i(t)$, Bandinschrift $a_{j(t,0)}, \dots, a_{j(t,T(|u|))}$.

Wir zeigen nur noch: $C_0, \dots, C_{T(|u|)}$ ist eine akzeptierende Berechnung. Wegen $\alpha_{\text{Anfang}} = \text{wahr}$ ist C_0 die Startkonfiguration zu u . Wegen $\alpha_{\text{Ende}} =$

wahr ist $C_{T(|u|)}$ eine akzeptierende Konfiguration. Wegen $\alpha_{\text{Übergang}} =$ wahr geschieht der Übergang $C_t \rightarrow C_{t+1}$ gemäß Zeile $z_{l(t)}$. Insgesamt erhalten wir eine akzeptierende Berechnung von $\mathfrak{M}$ auf u . $\square$

Im Folgenden werden wir nun einige NP -vollständige Probleme betrachten und ihre Schwere insbesondere beweisen. Es zeigt sich etwa, dass SAT sogar dann NP -vollständig bleibt, wenn man pro Klausel nur maximal 3 Literale erlaubt. Dieses Problem nennen wir 3-SAT.

6.5 SAT mit kleinen Klauseln (3-SAT)

3-SAT = $\{\alpha \mid \alpha \text{ Boolescher Ausdruck in KNF mit } \leq 3 \text{ Literalen pro Klausel, } \alpha \text{ erfüllbar}\}$.

Satz 6.25. 3-SAT ist NP -vollständig.

Beweis. Es reicht zu zeigen: SAT $\leq$ 3-SAT. Gesucht ist eine polynomzeit-berechenbare Transformation $\phi : \alpha \rightarrow \bar{\alpha}$ mit

$$\alpha \in \text{SAT} \Leftrightarrow \bar{\alpha} \in \text{3-SAT}.$$

Idee: $y_1 \vee y_2 \vee y_3 \vee y_4$ ist erfüllbar, g.d.w. $(y_1 \vee y_2 \vee x) \wedge (\neg x \vee y_3 \vee y_4)$ ist erfüllbar. Hierbei ist x eine neue Hilfsvariable.

Begründung: $y_1 \vee \dots \vee y_4$ erfüllbar $\Leftrightarrow y_1 \vee y_2$ erfüllbar oder $y_3 \vee y_4$ erfüllbar $\Leftrightarrow y_1 \vee y_2 \vee x$ erfüllbar **und** $y_3 \vee y_4 \vee \neg x$ erfüllbar.

Allgemein: $(y_1 \vee \dots \vee y_n)$ erfüllbar $\Leftrightarrow (y_1 \vee y_2 \vee x_1) \wedge (\neg x_1 \vee y_3 \vee x_2) \wedge \dots \wedge (\neg x_{n-3} \vee y_{n-1} \vee y_n)$ erfüllbar.

$\Rightarrow$: Es sei $(y_1 \vee \dots \vee y_n)$ wahr. Sind $(y_1 \vee y_2)$ oder $(y_{n-1} \vee y_n)$ wahr, so ist die rechte Seite erfüllbar durch Wahl $x_1 = \dots = x_{n-3} = \text{false}$ falls $y_1 \vee y_2$ wahr ist (bzw. alle Hilfsvariablen gleich true falls $y_{n-1} \vee y_n$ wahr ist). Andernfalls sei y_i wahr für $i \notin \{1, 2, n-1, n\}$. In diesem Fall setze $x_1, \dots, x_{i-2} = \text{true}$ und $x_{i-1}, \dots, x_{n-3} = \text{false}$.

$\Leftarrow$: Es sei eine Belegung gegeben, die die rechte Seite erfüllt. Falls alle $y_i = \text{false}$ sind, können nicht alle Disjunktionen true werden.

$$(y_1 \vee y_2 \vee x_1) \wedge (\neg x_1 \vee y_3 \vee x_2) \wedge \dots \wedge (\neg x_{n-3} \vee y_{n-1} \vee y_n).$$

Wenn $y_1 \vee y_2$ falsch ist, so muss x_1 wahr sein. Dann ist $\neg x_1 \vee y_3$ auch falsch und x_2 muss wahr sein. Am Ende ist dann aber $\neg x_{n-3}$ und $y_{n-1} \vee y_n$ falsch und wir haben einen Widerspruch.

Definiere $\alpha \rightarrow \bar{\alpha}$ entsprechend klauselweise. Dann gilt $|\bar{\alpha}| \leq c|\alpha|$ mit $c = \mathcal{O}(1)$ und $\alpha \rightarrow \bar{\alpha}$ ist polynomzeit-berechenbar. $\square$

Wir betrachten erneut das Cliquesproblem und analysieren dieses Mal also auch die Schwere des Problems.

6.6 Cliquesproblem (k -Clique)

Gegeben: Ein ungerichteter Graph $G = (V, E)$ und eine Zahl $k \geq 1$.

Eine *Clique* ist eine Teilmenge $C \subseteq V$ mit $\{u, v\} \in E$ für alle $u, v \in C$ mit $u \neq v$.

Entscheide: Hat der gegebene Graph G eine Clique $C \subseteq V$ mit mindestens k Knoten (d.h. mit $|C| \geq k$)?

Satz 6.26. *Das Problem k -Clique ist NP-vollständig.*

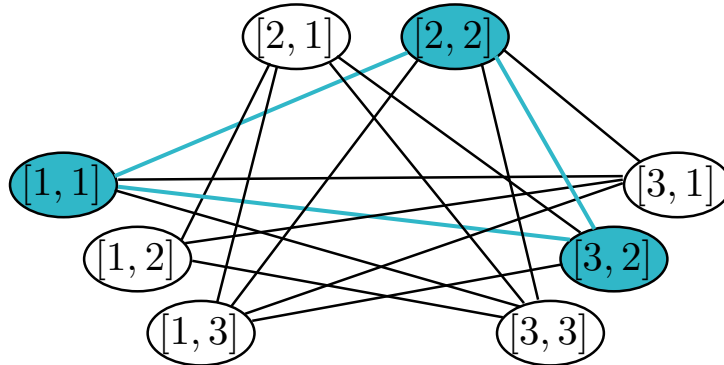
Beweis. (a) k -Clique $\in NP$: siehe Lemma 6.13 (b) Zeige: SAT $\leq k$ -Clique.

Es sei F ein Boolescher Ausdruck in KNF, wobei $F = F_1 \wedge \dots \wedge F_m$ und $F_i = (y_{i1} \vee \dots \vee y_{i\ell_i})$ eine Klausel mit ℓ_i Literalen ist. Konstruktion von $G = (V, E)$ (dies geht in poly. Zeit):

$$\begin{aligned} V &= \{[i, j] \mid 1 \leq i \leq m, 1 \leq j \leq \ell_i\} \\ E &= \{\{[i, j], [i', j']\} \mid i \neq i' \text{ und } y_{ij} \neq \neg y_{i'j'}\} \end{aligned}$$

Behauptung: F ist erfüllbar $\Leftrightarrow G$ enthält eine Clique mit $k = m$ Knoten.

Beispiel 6.27. Betrachte $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$; ist erfüllbar mit $\psi(x_1) = \psi(x_3) = \text{true}$ und $\psi(x_2) = \text{false}$. Zur Konstruktion von G und Clique C siehe folgendes Bild:



$\Rightarrow$: Es sei F erfüllbar. Dann existiert ein ψ mit $\psi(F) = \text{true}$. Für diese Belegung gilt $\psi(F_i) = \text{true}$ für alle $i = 1, \dots, m$. Dann $\exists \psi \forall i = 1, \dots, m \exists r_i \in \{1, \dots, \ell_i\}$ mit $\psi(y_{ir_i}) = \text{true}$.

Setze nun $C = \{[i, r_i] \mid 1 \leq i \leq m\}$ und zeige, dass C eine m -Clique ist.

Falls $\{[i, r_i], [j, r_j]\} \notin E$ für ein Paar i, j mit $i \neq j$, dann muss (wegen der Definition von E) $y_{ir_i} = \neg y_{jr_j}$ sein. Dagegen gilt aber $\psi(y_{ir_i}) = \psi(y_{jr_j}) = \text{true}$; ein Widerspruch. Also bildet C eine Clique.

$\Leftarrow$: Es sei C eine m -Clique in G . Da es keine Kante zwischen Knoten mit gleicher erster Komponente gibt, gilt:

$$C = \{[1, r_1], [2, r_2], \dots, [m, r_m]\}$$

für gewisse $r_1, r_2, \dots, r_m$. Definiere eine Belegung ψ mit $\psi(y_{1r_1}) = \psi(y_{2r_2}) = \dots = \psi(y_{mr_m}) = \text{true}$.

Dies geht widerspruchsfrei, da $y_{ir_i} \neq \neg y_{jr_j}$ für alle $1 \leq i \neq j \leq m$. Daraus folgt, dass $\psi(F_i) = \text{true}$ für alle $1 \leq i \leq m$ und damit $\psi(F) = \text{true}$ ist. $\square$

Ein weiteres Problem auf Graphen ist das sogenannte *Färbungsproblem*. Eine k -Färbung eines ungerichteten Graphen $G = (V, E)$ ist eine Abbildung $f : V \rightarrow \{1, \dots, k\}$ mit $f(i) \neq f(j)$ für alle $\{i, j\} \in E$ mit $i \neq j$. Eine k -Färbung f färbt also die Knoten des Graphen G mit k Farben so, dass Kanten nur zwischen *unterschiedlich* gefärbten Knoten existieren.

6.7 Färbungsproblem (k -Color)

Gegeben: Ein ungerichteter Graph $G = (V, E)$ und eine Zahl $k \geq 1$.

Entscheide: Hat der gegebene Graph G eine k -Färbung?

Satz 6.28. *Das Färbungsproblem k -Color ist NP-vollständig.*

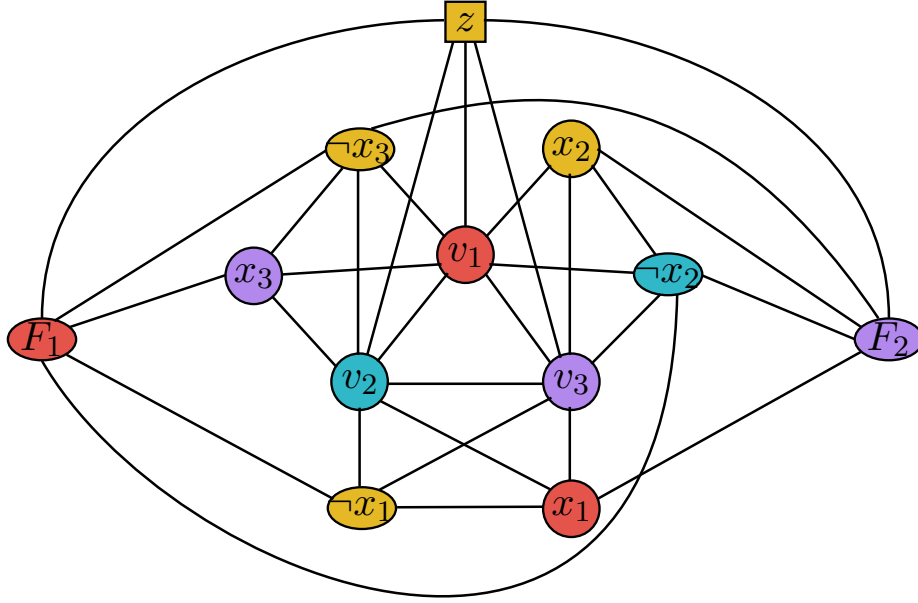
Beweis. (a) k -Color $\in NP$: Wähle als Zertifikat eine Abbildung $f : V \rightarrow \{1, \dots, k\}$ und teste, ob f eine k -Färbung ist. Der Testschritt geht in Zeit $\mathcal{O}(|V| + |E|)$.

(b) zeige: 3-SAT $\leq k$ -Color. Es sei F Boolescher Ausdruck in KNF mit $F = F_1 \wedge \dots \wedge F_m$ und F_i Klausel der Länge ≤ 3 und ohne Paare $x_j \vee \neg x_j$ in einer Klausel (diese Klauseln können eliminiert werden).

Konstruktion von $G = (V, E)$ mit $|V| = 3n + m + 1$:

$$\begin{aligned} V &= \{x_i, \bar{x}_i, v_i \mid 1 \leq i \leq n\} \cup \{F_j \mid 1 \leq j \leq m\} \cup \{z\}. \\ E &= \{\{v_i, v_j\} \mid 1 \leq i \neq j \leq n\} \\ &\quad \cup \{\{v_i, x_j\}, \{v_i, \bar{x}_j\} \mid 1 \leq i \neq j \leq n\} \\ &\quad \cup \{\{x_i, \bar{x}_i\} \mid 1 \leq i \leq n\} \\ &\quad \cup \{\{x_i, F_j\} \mid \text{falls } x_i \text{ kein Literal in } F_j\} \\ &\quad \cup \{\{\bar{x}_i, F_j\} \mid \text{falls } \neg x_i \text{ kein Literal in } F_j\} \\ &\quad \cup \{\{v_i, z\} \mid 1 \leq i \leq n\} \cup \{\{F_j, z\} \mid 1 \leq j \leq m\}. \end{aligned}$$

Beispiel 6.29. *Konstruktion für $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_3)$.*



Dieser Graph ist 4-färbbar und $\psi(x_1) = \psi(x_3) = \text{true}$, $\psi(x_2) = \text{false}$ ist erfüllende Belegung.

Behauptung: F ist erfüllbar $\Leftrightarrow G$ kann mit $n + 1$ Farben gefärbt werden.

Vorüberlegung: $\{v_1, \dots, v_n, z\}$ ist Clique. Daher brauchen wir $n+1$ Farben um $v_1, \dots, v_n, z$ zu färben. O.B.d.A. gilt $f(v_i) = i$ und $f(z) = n + 1$; ansonsten umfärben. Desweiteren gilt $\forall j \ x_j, \bar{x}_j$ sind mit jedem v_i für $i \neq j$ verbunden. Dann folgt $f(x_j), f(\bar{x}_j) \in \{j, n+1\}$. Da $\{x_j, \bar{x}_j\} \in E$ folgt $f(x_j) \neq f(\bar{x}_j)$.

D.h. einer der beiden Knoten wird mit Farbe j und der andere mit $n + 1$ gefärbt, falls eine $n + 1$ Färbung existiert. Dies erzeugt die Variablenbelegung.

Beweis der Behauptung. $\Rightarrow$: F ist erfüllbar. Dann existiert eine Belegung ψ , so dass F_j ein Literal y mit $\psi(y) = \text{true}$ enthält. Färbe nun

$$f(x_i) = \begin{cases} i & \text{falls } \psi(x_i) = \text{true} \\ n + 1 & \text{sonst} \end{cases}$$

und $f(\bar{x}_i)$ entsprechend. Dann ist $\{x_i, \bar{x}_i, v_i | 1 \leq i \leq n\} \cup \{z\}$ korrekt mit $n + 1$ Farben gefärbt. Weiter ist F_j mit x_i und $\bar{x}_i$ verbunden, falls x_i bzw. $\neg x_i$ kein Literal in F_j ist. Für ein Literal $y \in \{x_i, \neg x_i\}$ in F_j gilt $\psi(y) = \text{true}$ und $\{y, F_j\} \notin E$. Färbe dann $f(F_j) = i$. Dies ergibt dann korrekte Färbung.

$\Leftarrow$: Gegeben sei eine Färbung, die o.B.d.A. gegeben ist durch $f(v_i) = i$ und $f(z) = n + 1$. Dann sind $x_j, \bar{x}_j$ wie oben beschrieben mit Farbe j und $n + 1$ gefärbt. Betrachte Variablenbelegung

$$\psi(x_j) = \begin{cases} \text{true} & \text{falls } x_j \text{ Farbe } j \text{ hat} \\ \text{false} & \text{sonst} \end{cases}$$

und setze $\psi(\neg x_j) = \neg\psi(x_j)$. F_j hat ≤ 3 Literale. Daher ist F_j mit $\geq 2n - 3$ Knoten aus $\{x_i, \bar{x}_i | 1 \leq i \leq n\}$ (und mit z) verbunden.

Annahme: $\psi(F_j) = false$; d.h. alle drei Literale in F_j sind nach Definition mit $n + 1$ gefärbt. Desweiteren ist F_j verbunden mit Knoten aus $\{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$ und Farben $1, \dots, n$ (wegen der Voraussetzung über die Paare $x_j \vee \neg x_j$) und mit z und Farbe $f(z) = n + 1$. Daraus folgt $f(F_j) \notin \{1, \dots, n + 1\}$ und wir erhalten einen Widerspruch (da f eine $n + 1$ Färbung ist). $\square$

6.8 3-dimensionales Matching

Gegeben: Mengen U, V, W mit $|U| = |V| = |W|$ und Teilmenge $T \subseteq V \times W \times U$.

Entscheide: Gibt es eine Teilmenge $M \subseteq T$ mit $|M| = |U|$ mit der Eigenschaft: Für jede zwei unterschiedliche $(v, w, u), (v', w', u') \in M$ gilt: $u \neq u', v \neq v'$ und $w \neq w'$?

Satz 6.30. *Das 3-dimensionale Matchingproblem (3-dim. Matching) ist NP-vollständig.*

Beweis. (a) 3-dim. Matching ist in NP. Wähle als Zertifikat eine entsprechende Teilmenge $M \subseteq T$ und teste die obigen Bedingungen.

(b) SAT $\leq$ 3-dim. Matching. Es sei F ein Boolescher Ausdruck mit Variablen $x_1, \dots, x_n$ und Klauseln $C_1, \dots, C_m$. Konstruiere Instanz (V, W, U, T) mit:

$$\begin{aligned} V &= \{a_i^j | i = 1, \dots, n, j = 1, \dots, m\} \\ &\quad \cup \{v_j | j = 1, \dots, m\} \\ &\quad \cup \{c_k | k = 1, \dots, m(n-1)\} \\ W &= \{b_i^j | i = 1, \dots, n, j = 1, \dots, m\} \\ &\quad \cup \{w_j | j = 1, \dots, m\} \\ &\quad \cup \{d_k | k = 1, \dots, m(n-1)\} \\ U &= \{x_i^j, \bar{x}_i^j | i = 1, \dots, n, j = 1, \dots, m\} \end{aligned}$$

Es gilt $|U| = |V| = |W| = 2nm$. T enthalte die folgenden Vektoren

- (1) (a_i^j, b_i^j, x_i^j) und $(a_i^{j+1}, b_i^j, \bar{x}_i^j)$ für $i = 1, \dots, n, j = 1, \dots, m$ wobei $a_i^{m+1} = a_i^1$.

Beachte: die a und b Knoten tauchen in den anderen Vektoren nicht mehr auf. Hier überdeckt man in M entweder alle x_i^j für $j = 1, \dots, m$ oder alle $\bar{x}_i^j$. Das erzeugt die **Variablensetzung** $x_i = false$ oder $x_i = true$.

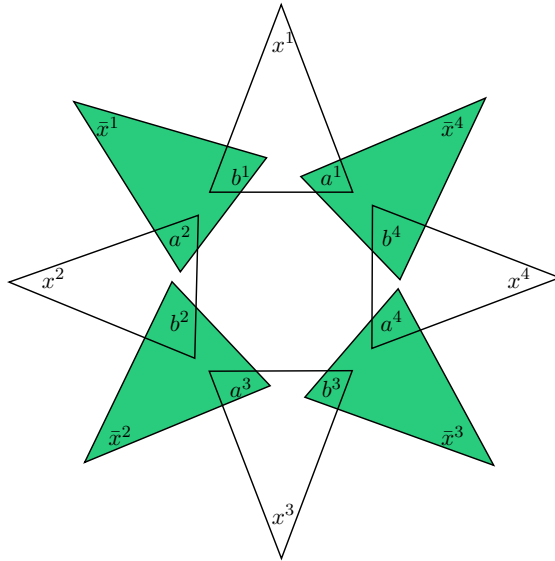
- (2) (v_j, w_j, λ^j) für $j = 1, \dots, m$ und λ Literal von C_j .

Die v und w Knoten tauchen in anderen Vektoren nicht mehr auf. $\Rightarrow$ Für jede Klausel C_j müssen wir ein Literal λ auswählen und einen Vektor $(v_j, w_j, \lambda^j) \in M$. D.h. die Knoten v_j, w_j werden mit dem **wahren** Literal aus C_j gematcht.

- (3) Die c und d Knoten spielen die Rolle einer **Garbage Collection**. Die Vektoren haben die Form $(c_k, d_k, x_i^j), (c_k, d_k, \bar{x}_i^j)$ für $k = 1, \dots, m(n-1)$ und $i = 1, \dots, n$ und $j = 1, \dots, m$.

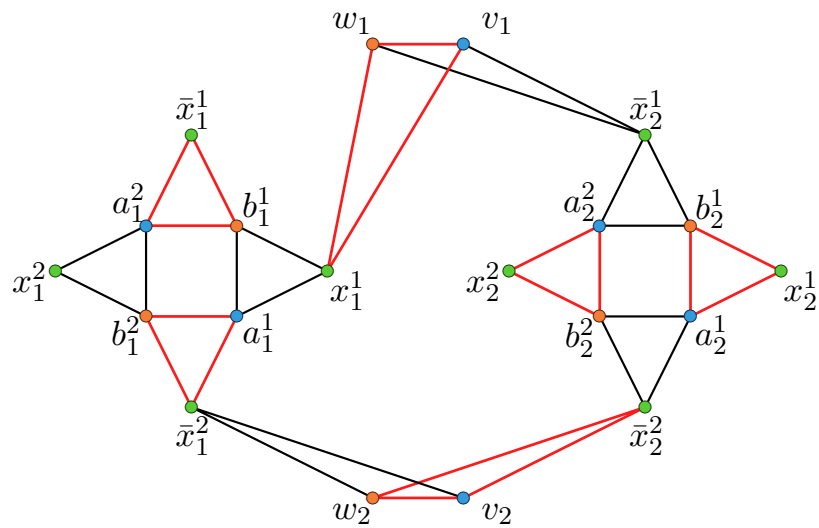
Damit sammelt man $(n-1)m$ Literale auf.

Zeige dann die Behauptung: F ist erfüllbar $\Leftrightarrow (V, W, U, T)$ enthält ein 3-dim. Matching.
Zur Variablensetzung betrachte folgendes Bild:



□

Konstruktion für $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2)$ mit $\psi(x_1) = \text{true}$ und $\psi(x_2) = \text{false}$:



6.9 3-Exact Cover

Gegeben: Familie $F = \{S_1, \dots, S_n\}$ von n Teilmengen einer Menge $U = \{u_1, \dots, u_{3m}\}$
mit $|S_j| = 3$ für alle $j = 1, \dots, n$.

Entscheide:

Gibt es eine Teilfamilie von F mit m Teilmengen $S_{i_1}, \dots, S_{i_m}$ und $\bigcup_{j=1, \dots, m} S_{i_j} = S$.

Satz 6.31. *Das Problem 3-Exact Cover ist NP-vollständig.*

Beweis. Dies folgt, da jede Instanz von 3-dim Matching als eine Instanz von 3-Exact Cover aufgefasst werden kann. $\square$

6.10 SubSet Sum

Gegeben: n ganze Zahlen $c_1, \dots, c_n$ und eine Zahl K .

Entscheide: Gibt es eine Teilmenge $S \subseteq \{1, \dots, n\}$ mit $\sum_{j \in S} c_j = K$?

Satz 6.32. *Das Problem SubSet Sum ist NP-vollständig.*

Beweis. (a) SubSet Sum $\in$ NP: Wähle als Zertifikat eine Teilmenge und teste ob Summe gleich K ist.

(b) 3-Exact Cover $\leq$ SubSet Sum. Gegeben sei eine Familie von n Mengen der Kardinalität 3 und Grundmenge $U = \{u_1, \dots, u_{3m}\}$. Konstruiere Zahlen $c_1, \dots, c_n$.

Idee: Schreibe jede Menge S_j in F als einen Bitvektor der Länge $3m$. Zum Beispiel $\{u_1, u_5, u_6\}$ als 100011. Nun interpretiere jeden Bitvektor als Zahl zur Basis $(n+1)$:

$$c_j = \sum_{u_i \in S_j} (n+1)^{i-1}$$

Dann gibt z.B. die Menge $\{u_1, u_5, u_6\}$ die Zahl $(n+1)^0 + (n+1)^4 + (n+1)^5$.

Sei K die Zahl zum Bitvektor 11...1 der Länge $3m$. Dann gilt

$$K = \sum_{j=0}^{3m-1} (n+1)^j.$$

Behauptung: $\exists m$ Teilmengen in F , die $\{u_1, \dots, u_{3m}\}$ überdeckt $\Leftrightarrow$ es gibt Teilmenge der c_j , deren Summe genau K ergibt.

$\Leftarrow$: Sei $S \subseteq \{1, \dots, n\}$ mit $\sum_{j \in S} c_j = K$.

Beachte: (1) Koeffizienten von $(n+1)^i$ sind immer 0 oder 1. (2) Wir haben $< (n+1)$ Summanden auf der rechten Seite, da $|S| \leq n$. D.h. es gibt keinen Übertrag bei der Addition zur Basis $(n+1)$. Und wir erhalten genau eine 1 in jeder Bitposition. Daraus folgt, dass $C = \{S_j | j \in S\}$ genau die Menge $\{u_1, \dots, u_{3m}\}$ überdeckt.

$\Rightarrow$: Gegeben sei ein Exact Cover C von $\{u_1, \dots, u_{3m}\}$. Dann folgt sofort $\sum_{S_j \in C} c_j = K$. $\square$

6.11 Partition

Gegeben: n ganze Zahlen $c_1, \dots, c_n$.

Entscheide: Gibt es eine Teilmenge $S \subseteq \{1, \dots, n\}$ mit $\sum_{j \in S} c_j = 1/2 \sum_{j=1}^n c_j$?

Satz 6.33. *Das Problem Partition ist NP-vollständig.*

Beweis. (a) Partition $\in$ NP (klar).

(b) SubSet Sum $\leq$ Partition. Gegeben seien Zahlen $c_1, \dots, c_n, K$. Setze $N = \sum_{j=1}^n c_j + 1$ und konstruiere die folgende Menge von $(n+2)$ Gegenständen: $\{1, \dots, n\} \cup \{b, c\}$ mit Zahlen $c_1, \dots, c_n$ und $c_{n+1} = N - K$ und $c_{n+2} = K + 1$. Dann gilt $\sum_{j=1}^{n+2} c_j = (N-1) + (N-K) + (K+1) = 2N$. D.h.

$$\frac{1}{2} \sum_{j=1}^{n+2} c_j = N.$$

Behauptung: $c_1, \dots, c_n, K$ ist Ja-Eingabe von SubSet Sum $\Leftrightarrow c_1, \dots, c_{n+2}$ ist Ja-Eingabe von Partition.

Vorbemerkung: $\{b, c\}$ können nicht in einer Lösung zusammen vorkommen, da $c_{n+1} + c_{n+2} = (N-K) + (K+1) = N+1$.

$\Rightarrow$: Sei $S \subseteq \{1, \dots, n\}$ mit $\sum_{j \in S} c_j = K$ gegeben. Wähle $S \cup \{b\}$ und erhalte $\sum_{j \in S} c_j + c_{n+1} = K + (N-K) = N$. Dies ist Lösung von Partition.

$\Leftarrow$: Sei $S \subseteq \{1, \dots, n+2\}$ Lösung von Partition mit $\sum_{j \in S} c_j = N$ gegeben. Dann gilt für die Komplementmenge $S^c = \{1, \dots, n+2\} \setminus S$ auch $\sum_{j \in S^c} c_j = N$. Wegen der obigen **Vorbemerkung** können b und c nicht beide in S und auch nicht in S^c liegen. O.B.d.A. liege $b \in S$ und damit $c \notin S$.

Dann gilt

$$\begin{aligned} N &= \sum_{j \in S} c_j = \sum_{j \in S \setminus \{b, c\}} c_j + c_{n+1} \\ &= \sum_{j \in S \setminus \{b, c\}} c_j + (N-K) \end{aligned}$$

Deswegen gilt nun

$$\sum_{j \in S \setminus \{b, c\}} c_j = K;$$

d.h. $S \setminus \{b, c\}$ ist Lösung von SubSet Sum. □

6.12 Rucksackproblem

Gegeben: n Gegenstände mit Größen $c_1, \dots, c_n$ und Gewinnen $p_1, \dots, p_n$ und Rucksackkapazität K und Profitwert P .

Entscheide: Gibt es eine Teilmenge $S \subseteq \{1, \dots, n\}$ der n Gegenstände mit Gesamtgröße $\sum_{j \in S} c_j \leq K$ und Gesamtprofit $\sum_{j \in S} p_j \geq P$?

Satz 6.34. *Das Rucksackproblem ist NP-vollständig.*

Beweis. Spezialfall mit $c_j = p_j$ für alle Gegenstände und $P = K$ entspricht dem SubSet Sum Problem. □

6.13 Exponentialzeit-Hypothese

Die Exponentialzeit-Hypothese (ETH) ist eine, bisher unbewiesene, Rechenhärte-Annahme. Formuliert wurde sie von Impagliazzo, Paturi und Zane im Jahre 2001 [IPZ01].

Satz 6.35 (Exponential Time Hypothesis). *Es existiert eine positive Zahl $\delta \in \mathbb{R}$ so dass 3-SAT mit n Variablen und m Klauseln nicht in Zeit $2^{\delta n}(n+m)^{O(1)}$ gelöst werden kann.*

Dieses Theorem ist dadurch motiviert, dass alle bekannten Algorithmen für 3-SAT eine Laufzeit von $c^n(n+m)^{O(1)}$ besitzen. Die derzeit kleinste Konstante c ist 1.30704

Lemma 6.36 (Sparsifikation-Lemma). *Unter der Annahme der ETH existiert ein $\delta' \in \mathbb{R}_{\geq 0}$, so dass 3-SAT mit m Klauseln nicht in Zeit $O(2^{\delta' m}) = 2^{\delta' m}(n+m)^{O(1)}$ gelöst werden kann.*

Aus obigem Lemma folgt, dass kein Algorithmus mit Laufzeit $2^{o(m)}$ für 3-SAT existieren kann.

Satz 6.37. *Gegeben eine 3-SAT Formel Φ mit n Variablen und m Klauseln ist es möglich einen Graphen G mit $O(n+m)$ Knoten in polynomieller Zeit zu konstruieren, der genau dann 3-färbbar ist, wenn Φ erfüllbar ist.*

Da es möglich ist diesen Graphen in polynomieller Zeit zu erstellen und die Anzahl der Knoten linear in Variablen und Klauseln ist können wir direkt unter Annahme der ETH folgendes schlussfolgern.

Satz 6.38. *Unter Annahme der ETH gibt es keinen $2^{o(n)}$ Algorithmus für 3-Färbung.*

Mittels ähnlichen Transformationen können wir ähnliche Ergebnisse auch für das Cliquesproblem, Vertex Cover und Independent Set folgern.

Satz 6.39. *Unter Annahme der ETH gibt es keinen $2^{o(n)}$ Algorithmus für das Cliquesproblem, Vertex Cover und Independent Set.*

Die ETH hat auch Konsequenzen für die bekannten Subset Sum und Partition-Probleme. Wir zeigen eine Reduktion von 3-SAT auf Subset Sum und schließen damit folgendes:

Satz 6.40. *Partition, Subset Sum können nicht in Zeit $2^{o(n)} \text{poly}(|I|)$ entschieden werden, außer die ETH ist falsch.*

Beweis. (Beweis von Ingo Wegener) Gegeben sei eine Formel mit Variablen $x_1, \dots, x_n$ und Klauseln $c_1, \dots, c_m$. Für jede Variable x_i erzeuge zwei Gegenstände a_i, b_i mit Größe

$$\begin{aligned} s(a_i) &= 10^{i-1} + \sum_{j \in [m], x_i \in c_j} 10^{n+j-1} \\ s(b_i) &= 10^{i-1} + \sum_{j \in [m], \bar{x}_i \in c_j} 10^{n+j-1} \end{aligned}$$

[Alle Zahlen haben $n + m$ Ziffern zur Basis 10] Erzeuge zusätzlich zwei Dummy Items d_j, e_j für Klausel c_j mit $s(d_j) = s(e_j) = 10^{n+j-1}$.

Die Itemmenge $A = \{a_i, b_i | i \in [n]\} \cup \{d_j, e_j | j \in [m]\}$. Target-Wert ist dann $B = \sum_{i=1}^n 10^{i-1} + \sum_{j=1}^m 3 \cdot 10^{n+j-1}$

Bemerkung: (a): Eine Lösung kann nur entweder a_i oder b_i enthalten. Dies folgt aus der Variablensetzung.

(b): Bei den höheren Ziffern benötigt man durch die Variablensetzung Werte 1,2 oder 3 ($\hat{=}$ # Literale, die erfüllt sind pro Klausel) [die Dummy Items kann man nutzen um auf genau 3 zu kommen]

Für den Beweis zeige noch: Die Reduktion ist streng (d.h. $|A| = O(m)$)

a) Wir können annehmen, dass $n \leq 3m$ [beachte: $m \leq O(n^3)$]

$\rightarrow |A| = 2n + 2m \leq_{n \leq 3m} 6m + 2m = 8m$

Der Rest geht dann per indirektem Beweis. Dafür nehmen wir an, dass ein Algorithmus für Subset Sum mit Laufzeit $2^{o(n)} \text{poly}(|I|)$ existiert. Für eine Formel Φ in 3-SAT in KNF mit m Klauseln konstruiere Instanz von Subset Sum mit $\leq 8m = O(m)$ Gegenständen und löse diese Instanz mit obigem Algorithmus.

$$2^{o(8m)} \text{poly}(|I|) = 2^{o(m)} \text{poly}(|I|).$$

Wegen Korrektheit und Laufzeit der Reduktion bestimmt dies die Erfüllbarkeit von Φ in der Laufzeit von $2^{o(m)} \text{poly}(|I|)$. Dies ist ein Widerspruch zum Sparsification Lemma der ETH. Als Anmerkung: Ohne das Sparsification Lemma kann kein Algorithmus für Subset Sum existieren mit Laufzeit von $2^{o(\sqrt[3]{n})}$ $\square$

Einschub: Eine Anmerkung zur o -Notation: Die kleine o -Notation ist wie folgt definiert.

$$\begin{aligned} f(x) = o(g(x)) & \Leftrightarrow \\ \forall c > 0 : \exists x_0 \geq 0 : \forall x \geq x_0 : & |f(x)| \leq c \cdot |g(x)| \end{aligned}$$

Man kann dies auch als $\frac{f(x)}{g(x)} \rightarrow 0$ für $x \rightarrow \infty$. Nach dieser Definition gilt also auch dass $\delta n = o(n)$, weswegen die beiden Schreibweisen austauschbar verwendet werden. Die Originale Formulierung der ETH verwendet die δn Notation, weswegen sie in unserer Formulierung der Theoreme verwendet wurde.

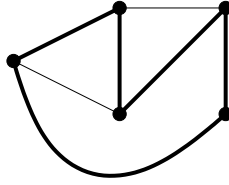
6.14 Hamiltonkreis Problem

Das Hamiltonkreis Problem (HK) ist wie folgt definiert:

Gegeben: Ein ungerichteter Graph $G = (V, E)$ mit $V = \{v_1, \dots, v_n\}$.

Entscheide: Gibt es einen Hamiltonschen Kreis; d.h. eine Permutation π der Knoten $(v_{\pi(1)}, \dots, v_{\pi(n)})$ mit $\{v_{\pi(i)}, v_{\pi(i \bmod n + 1)}\} \in E$ für alle $i = 1, \dots, n$?

Beispiel 6.41. Der Graph in dem folgenden Bild hat einen entsprechenden Hamiltonkreis; dieser ist fettgezeichnet.

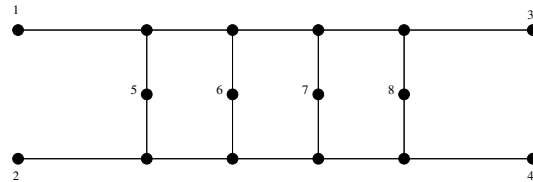


Satz 6.42. *Das Hamiltonkreis Problem ist NP-vollständig.*

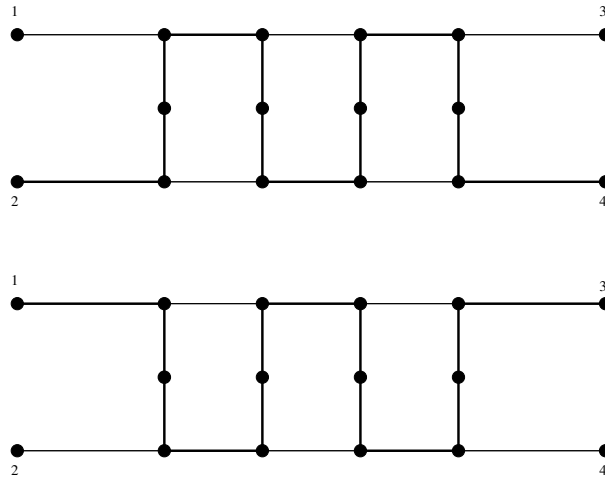
Bemerkung 6.43. *Die gleiche Aussage gilt auch für gerichtete Graphen.*

Beweis. (a) HK is in NP: Wähle als Zertifikat eine Permutation π der Knoten und teste, ob $\{v_{\pi(i)}, v_{\pi(i+1 \bmod n)}\} \in E$ ist für alle $i = 1, \dots, n$.

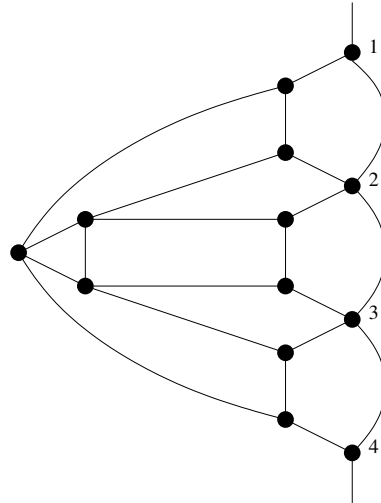
(b) $3-SAT' \leq HK$: Gegeben seine SAT Formel mit genau 3 Literalen pro Klausel (ist ebenfalls NP-vollständig). Zur Beweisidee geben wir die folgende Graph-Konstruktionen an. Als erstes verwende die folgende A -Komponente:



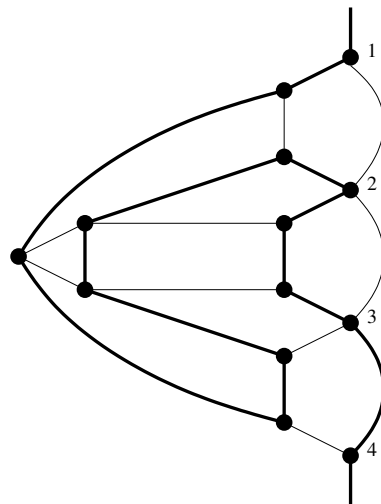
Dann gibt es zwei Möglichkeiten, die Komponente A zu durchlaufen; siehe die folgenden zwei Abbildungen:



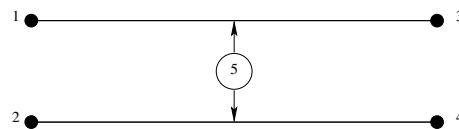
Für jede Klausel verwenden wir die folgende Komponente B :

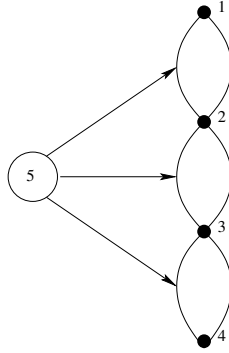


Eine der vielen Möglichkeiten, wie B durchlaufen werden kann, ist im folgenden Bild dargestellt.



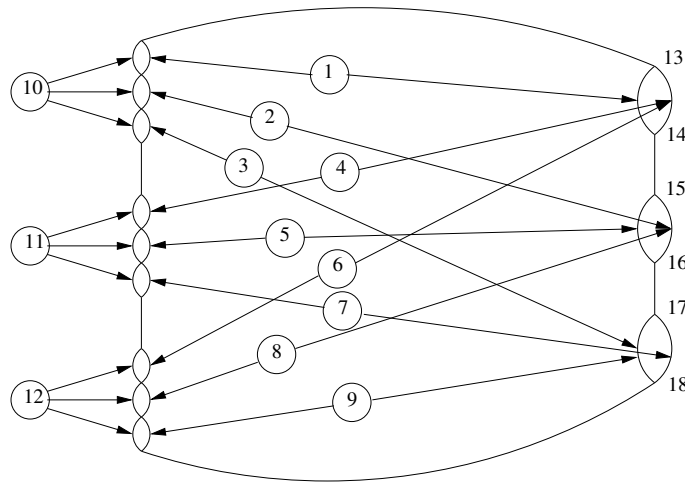
Insgesamt gibt es für jede nicht-leere Teilmenge der rechten Kanten $\{u_1, u_2\}, \{u_2, u_3\}, \{u_3, u_4\}$ eine Möglichkeit die Komponente B zu durchlaufen. Bei einer leeren Teilmenge dieser Kanten funktioniert dies aber nicht; dies entspricht dem Fall, dass alle Literale falsch gewählt worden sind. Als Kurzform für die Spezialkomponenten A und B verwenden wir:





Im folgenden Beispiel zeigen wir die Konstruktion des Graphen G zu einem Booleschen Ausdruck F .

Beispiel 6.44. Ausdruck $F = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$ und der zugehörige konstruierte Graph G :



Für die Variablensetzung von Variable x_i wählt man im rechten Pfad im obigen Bild jeweils zwischen zwei Knoten v_i, w_i für $i = 1, \dots, n$ die rechte oder linke Kopie der Kante $\{v_i, w_i\}$. Die A -Komponenten werden nun genutzt um die Verbindung zwischen den Variablen und den Literalen in den B -Komponenten zu den entsprechenden Klauseln herzustellen. Dazu bezeichnen wir die u Knoten in der B -Komponente zu Klausel F_i mit u_{i1}, u_{i2}, u_{i3} und u_{i4} .

Wenn das j .te Literal in Klausel F_i gleich x_k ist, dann verbinden wir die Kante $\{u_{ij}, u_{ij+1}\}$ via einer A -Komponente mit der linken Kopie von $\{v_k, w_k\}$. Wenn das j .te Literal gleich $\neg x_k$, so verbinden wir die Kante $\{u_{ij}, u_{ij+1}\}$ mit der rechten Kopie.

Formal beweist man Ende die folgende Aussage: F ist erfüllbar genau dann, wenn G einen Hamiltonkreis besitzt. $\square$

Die Details findet man im Buch von Papadimitriou und Steiglitz.

Übungsaufgaben

Übung 6.45. Die Eingabe des Problems SAT^3 ist ein boolescher Ausdruck in KNF, für den entschieden werden soll, ob es mindestens 3 erfüllende Belegungen für diese Formel gibt.

Zeigen Sie, dass SAT^3 NP-vollständig ist.

Übung 6.46. Eine NAE- k -SAT (Not-All-Equal) Formel hat folgende Form $\bigwedge_{i=1}^m (z_1^i, \dots, z_k^i)$, wobei eine Klausel $(z_1^i, \dots, z_k^i)$ mit Literalen $z_1^i, \dots, z_k^i$ genau dann erfüllt ist, wenn mindestens ein Literal zu wahr und mindestens ein Literal zu falsch ausgewertet wird.

Beispiel: Für die NAE-3-SAT Formel $(x, y, z) \wedge (\neg x, y, z)$ ist durch $x = y = z = \text{true}$ keine erfüllende Belegung gegeben, da in diesem Fall die erste Klausel nicht erfüllt ist. Bei der Belegung $x = y = \text{true}$ und $z = \text{false}$ hingegen ist die Formel erfüllt.

Zeigen Sie in einem ersten Schritt, dass NAE-4-SAT NP-vollständig ist, und in einem zweiten Reduktionsschritt zeigen, dass ebenfalls NAE-3-SAT NP-vollständig ist.

Übung 6.47. Ein Independent Set in einem Graphen $G = (V, E)$ ist eine Menge von Knoten I , so dass für je zwei Knoten $i, j \in I$ gilt, dass $\{i, j\} \notin E$. Für das Problem INDEPENDENT SET ist ein Graph $G = (V, E)$ sowie eine Zahl $k \in \mathbb{N}$ gegeben und es soll entschieden werden, ob es in G ein Independent Set mit k Knoten gibt.

Zeigen Sie, dass das Problem INDEPENDENT SET NP-vollständig ist.

Übung 6.48. Ein Vertex Cover in einem Graphen $G = (V, E)$ ist eine Menge von Knoten $C \subseteq V$ so dass für jede Kante $\{v, w\} \in E$ gilt das $v \in C$ oder $w \in C$. Für das Problem VERTEX COVER ist ein Graph $G = (V, E)$ sowie eine Zahl $k \in \mathbb{N}$ gegeben und es soll entschieden werden, ob es in G ein Vertex Cover mit k Knoten gibt.

Zeigen Sie, dass das Problem VERTEX COVER NP-vollständig ist.

Übung 6.49. In dem Problem CLIQUE-MEMBER ist ein Graph $G = (V, E)$, ein Knoten $v \in V$ sowie eine Zahl $k \in \mathbb{N}$ gegeben und es soll entschieden werden, ob es eine Clique mit k Knoten gibt, die den Knoten v enthält.

Zeigen Sie, dass CLIQUE-MEMBER NP-vollständig ist.

Übung 6.50. In dem Problem CLIQUE-NOMEMBER ist ein Graph $G = (V, E)$, ein Knoten $v \in V$ sowie eine Zahl $k \in \mathbb{N}$ gegeben und es soll entschieden werden, ob es eine Clique mit k Knoten gibt, die den Knoten v nicht enthält.

Zeigen Sie, dass CLIQUE-NOMEMBER NP-vollständig ist, indem Sie eine Reduktion von CLIQUE auf CLIQUE-NOMEMBER angeben.

Übung 6.51. In dem Problem Z-Clique ist ein **zusammenhängender** Graph $G = (V, E)$ und eine Zahl $k \in \mathbb{N}$ gegeben und es soll entschieden werden, ob G eine Clique mit k Knoten enthält.

Zeigen Sie, dass Z-Clique NP-vollständig ist.

Übung 6.52. Zeigen Sie, dass das Partition Problem ebenfalls NP-schwer ist, wenn eine Teilmenge $A'' \subseteq A$ gesucht ist mit

$$\sum_{a \in A''} a = 3 \sum_{a \in A \setminus A''} a.$$

Übung 6.53. Für das Problem SUBSET SUM CARDINALITY seien n ganze Zahlen $c_1, \dots, c_n \in \mathbb{N}_{>0}$ (wobei n gerade sei) sowie eine Zahl $K \in \mathbb{N}$ gegeben. Es soll entschieden werden, ob es eine Teilmenge $S \subset \{1, \dots, n\}$ gibt mit $|S| = n/2$ und $\sum_{i \in S} c_i = K$.

Zeigen Sie, dass das Problem SUBSET SUM CARDINALITY NP-vollständig ist.

Übung 6.54. Eine Eingabe des Problems EVEN-KNAPSACK ist eine Menge von Items I , sowie ein Rucksack mit gerader Größe B (d.h. es gibt $b \in \mathbb{N}$ mit $B = 2b$), sowie ein Profitwert $P \in \mathbb{N}$. Jedes Item hat eine Größe $s_i \in \mathbb{N}$ und einen Profit $p_i \in \mathbb{N}$. Das Entscheidungsproblem ist eine Teilmenge $I' \subseteq I$ zu finden mit $\sum_{i \in I'} s_i \leq B$ und $\sum_{i \in I'} p_i \geq P$.

Zeigen Sie, dass EVEN-KNAPSACK NP-vollständig ist.

Übung 6.55. Die Probleme Multiple-Choice-Knapsack (kurz MC-Knapsack) ist wie folgt definiert: Gegeben sind endliche Mengen $C_1, \dots, C_k \subseteq \mathbb{N} \times \mathbb{N}$ von Items, eine Kapazität $B \in \mathbb{N}$ und ein Zielprofit $P \in \mathbb{N}$. Wir bezeichnen C_i als Klasse von Items und für ein Item $(w, p) \in C_i$ ist w das Gewicht und p der Profit des Items. Es soll entschieden werden, ob aus jeder Klasse C_i genau ein Item (w_i, p_i) gewählt werden kann, sodass die gewählten Items gemeinsam den Zielprofit erfüllen ohne die Kapazität zu überschreiten, d.h. $\sum_{i \in [k]} p_i \geq P$ und $\sum_{i \in [k]} w_i \leq B$.

Das Problem ist in NP. Zeigen Sie: MC-Knapsack ist NP-vollständig

7 Approximative Algorithmen

Das letzte Kapitel hat uns Einblicke in die Schwierigkeit der polynomiellen Berechenbarkeit gegeben. Wir wissen nun insbesondere, dass es viele Probleme gibt, deren exakte Lösung eine große, nicht-polynomielle Laufzeit benötigt - es sei denn, es gilt $P = NP$. In Ermangelung einer Eingebung zu der Frage, ob $P = NP$ oder $P \neq NP$ gilt, hat sich die Disziplin der *Approximativen Algorithmen* entwickelt; anstatt eine exakte Lösung zu verlangen, lässt man die Berechnung *geringfügig* schlechterer Lösungen zu, welche sich aber deutlich schneller berechnen lassen. Die exakte Lösung wird in diesem Sinne nur noch *approximiert*.

In diesem Kapitel wollen wir einige solche Approximationsalgorithmen studieren.

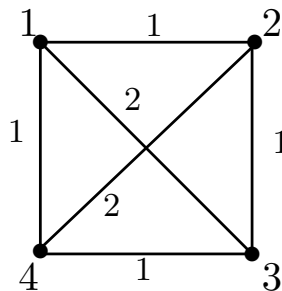
7.1 Traveling Salesman Problem

Gegeben: Eine Menge $V = \{1, \dots, n\}$, Distanzen $d(i, j) \in \mathbb{Z}^+ \cup \{\infty\}$ für alle $i, j \in \{1, \dots, n\}$

Gesucht: Eine minimale Rundreise, die jeden Knoten genau einmal besucht.

Formal: Eine Lösung ist gegeben durch eine Permutation Π von $\{1, \dots, n\}$, wobei die Länge der Tour $\sum_{i=1}^n d(\Pi(i), \Pi(i+1))$ mit $\Pi(n+1) = \Pi(1)$ ist.

Beispiel 7.1. Betrachte den folgenden Graphen mit 4 Knoten:



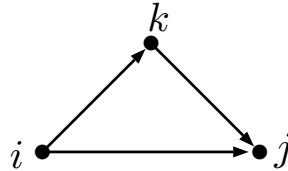
Eine optimale Rundreise ist $[1, 2, 3, 4, 1]$ mit Länge 4.

Satz 7.2. Das Entscheidungsproblem zum Traveling Salesman Problem, ob eine Rundreise mit Länge $\leq L$ existiert, ist NP-vollständig.

Beweis. Dies folgt durch eine Reduktion vom Hamiltonkreis Problem: Wähle für jede Kante $\{u, v\} \in E$ den Wert $d(u, v) = 1$ und für alle anderen Paare $\{u, v\} \notin E$ den Wert $d(u, v) = |V| + 1$ sowie $L = |V|$. $\square$

Durch Modifikation der Distanzen $d(u, v) = \infty$ für alle $\{u, v\} \notin E$, kann man sogar zeigen, dass es keinen approximativen Algorithmus für das allgemeine TSP Problem mit beschränkter Approximationsgüte gibt.

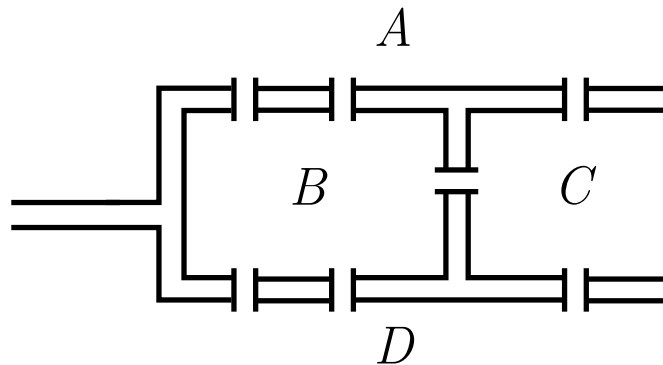
Deswegen betrachten wir den symmetrischen Fall des TSP mit $d(i, j) = d(j, i)$ und mit Erfüllung der Dreiecksungleichung (kurz: Δ -Ungleichung) $d(i, j) \leq d(i, k) + d(k, j)$; siehe auch folgendes Bild:



Bemerkung 7.3. Das TSP bleibt NP-vollständig im symmetrischen Fall mit Dreiecksungleichung.

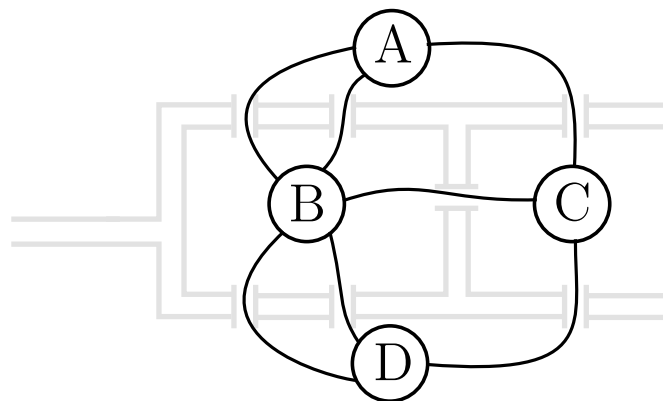
Definition 7.4. Es sei G ein Multigraph mit parallelen Kanten. Ein Eulerscher Kreis ist ein Kreis, der jede Kante von G genau einmal besucht.

Beispiel 7.5. (Königsberger Brückenproblem)



Frage: Gibt es einen Rundweg, bei dem man über jede der 7 Brücken genau einmal läuft?

Wir können das Brückenproblem in ein Graphproblem wie folgt transformieren:



Frage: Gibt es einen Eulerschen Kreis in dem Graphen?

Die Frage nach einem Eulerschen Kreis lässt sich durch das folgende Kriterium beantworten.

Satz 7.6. *Es sei G ein zusammenhängender Multigraph. Dann sind äquivalent:*

- (a) G ist Eulersch (d.h. besitzt einen Eulerschen Kreis).
- (b) Jeder Knoten in G hat geraden Grad.
- (c) Die Kantenmenge von G kann in disjunkte Kreise zerlegt werden.

Beweis. zur Übung. □

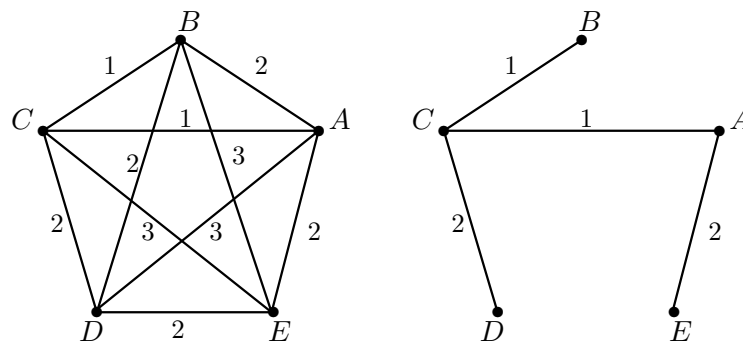
Mit Hilfe von dem obigen Kriterium sieht man nun, dass der obige Graph zum Königsberger Brückenproblem Knoten mit ungeradem Grad besitzt und es daher keinen entsprechenden Rundweg gibt.

Wir betrachten nun den folgenden Algorithmus:

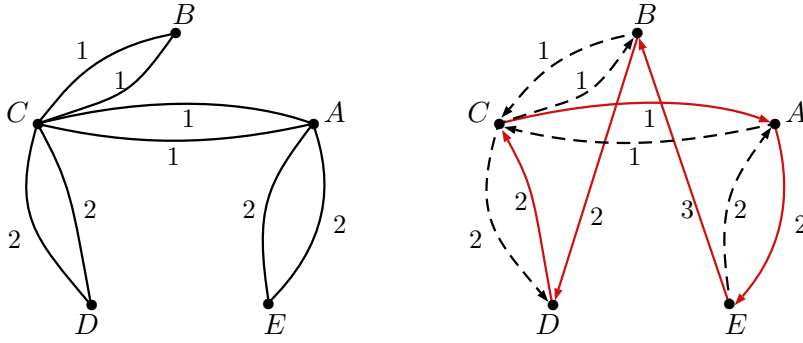
Algorithmus $\Delta TSP_1(V = \{1, \dots, n\}, D = (d(i, j)))$

- 1 Berechne einen minimalen spannenden Baum T des Graphen
- 2 K_n mit Distanzen $d(i, j)$;
- 3 Konstruiere einen Multigraphen G aus T durch Verdoppeln
- 4 aller Kanten;
- 5 Bestimme einen Eulerschen Kreis K in G ;
- 6 Bestimme eine Rundreise R aus K durch Abkürzungen;
- 7 return Rundreise R .

Beispiel 7.7. Betrachte den Graphen mit $V = \{A, B, C, D, E\}$; siehe folgendes Bild (links).



Eine optimale Tour ist $[C, B, D, E, A, C]$ der Länge 8. Der Algorithmus berechnet einen Baum T mit Gewicht $w(T) = 6$; siehe obiges Bild (rechts). Wir erhalten den Graphen G durch Verdopplung der Kanten mit $w(G) = 12$; siehe folgendes Bild:

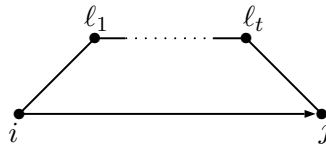


Als Eulerschen Kreis erhalten wir $K = [C, A, E, A, C, B, C, D, C]$ mit Länge $d(K) = 12$; und durch die Abkürzungen (d.h. Überspringen der schon vorher besuchten Knoten) erhalten wir die Rundreise $R = [C, A, E, B, D, C]$ mit Länge $d(R) = 10$.

Satz 7.8. Der Algorithmus ΔTSP_1 hat eine multiplikative Güte 2; d.h. $d(R) \leq 2OPT(I)$, wobei $OPT(I)$ die Länge einer minimalen Rundreise zu der Eingabe I mit $V = \{1, \dots, n\}$ und Distanzen $D = (d(i, j))$ ist.

Beweis. Die Behauptung folgt aus den folgenden Ungleichungen:

- (1) Das Gewicht $w(T) \leq OPT(I)$, da eine optimale Rundreise nach Weglassen einer Kante einen spannenden Baum ergibt und $w(T)$ das kleinste Gewicht eines spannenden Baums ist.
- (2) $d(K) = 2w(T) \leq 2OPT(I)$. Die erste Gleichung folgt wegen der Verdopplung der Kanten und die Ungleichung wegen (1).
- (3) $d(R) \leq d(K)$. Das gilt, da wegen der Δ -Ungleichung die folgende Ungleichung gilt: $d(i, j) \leq d(i, \ell_1) + d(\ell_1, \ell_2) + \dots + d(\ell_t, j)$; siehe auch folgendes Bild:



Formal zeigt man die obige Ungleichung per Induktion nach $t \geq 1$. □

Bemerkung 7.9. Es gibt Beispiele für Eingaben I zu dem Traveling Salesman Problem mit $OPT(I) = n$, bei denen der Algorithmus ΔTSP_1 eine Rundreise der Länge $2n - 2$ berechnet.

Beweis. Zur Beweisidee geben wir hier ein Beispiel mit $n = 8$ Knoten. Betrachte einen Stern mit Knoten 8 in der Mitte und Knoten $1, \dots, 7$ außen, die über einen Kreis miteinander verbunden sind. Formal haben wir $V = \{1, \dots, 8\}$ und $E = \{\{i, 8\} | i = 1, \dots, 7\} \cup \{\{i, i+1\} | i = 1, \dots, 6\} \cup \{\{7, 1\}\}$. Alle Kanten haben die Distanz 1; die anderen Nicht-Kanten haben Distanz 2.

Eine optimale TSP Tour hat die Form $[1, 2, 3, 4, 5, 6, 7, 8, 1]$ der Länge 8. Dagegen kann der MST Algorithmus einen Stern mit 8 als Mittelpunkt und allen anderen Knoten außen generieren; d.h. $T = (V, E')$ hat die Kantenmenge $E' = \{\{i, 8\} | i = 1, \dots, 7\}$. Durch Verdopplung der Kanten erhalten wir einen Multigraphen mit $2n - 2$ Kanten. Eine mögliche Eulertour ist

$$[8, 1, 8, 3, 8, 5, 8, 7, 8, 2, 8, 4, 8, 6, 8].$$

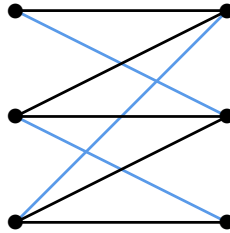
Der Algorithmus erzeugt dann über die Abkürzungen die folgende Tour:

$$[8, 1, 3, 5, 7, 2, 4, 6, 8]$$

mit Länge $14 = 2n - 2$. □

Definition 7.10. Ein Matching in einem ungerichteten Graphen $G = (V, E)$ ist eine Teilmenge $E' \subseteq E$ von Kanten, von denen keine zwei Kanten einen gemeinsamen Endpunkt gemeinsam haben. Bei Gewichten $d(i, j)$ auf den Kanten sei $d(E') = \sum_{\{i, j\} \in E'} d(i, j)$ das Gewicht von E' . Ein Matching ist perfekt, falls jeder Knoten $v \in V$ dabei überdeckt wird; d.h. $2|E'| = |V|$.

Beispiel 7.11. Ein bipartiter Graph G und ein perfektes Matching:



Satz 7.12. (Lawler 1976): Ein perfektes Matching in einem Graphen $G = (V, E)$ mit minimalem Gewicht kann in Zeit $\mathcal{O}(|V|^3)$ berechnet werden.

Beweis. siehe z.B. Vorlesung Discrete Optimization. □

Eine bessere Approximationsgüte erhalten wir, in dem wir die Knoten mit ungeraden Grad optimaler durch Kanten augmentieren. Hier hilft uns ein Matching mit minimalem Gewicht. Man beachte, dass die Anzahl der Knoten mit ungeradem Grad gerade ist; siehe auch Lemma 5.4.

Algorithmus $\Delta TSP_2(V = \{1, \dots, n\}, D = (d(i, j)))$

- 1 Berechne einen minimalen spannenden Baum T des
- 2 Graphen K_n mit Gewichten $d(i, j)$;
- 3 Bestimme die Menge X der Knoten in T ,
- 4 die ungeraden Grad haben;
- 5 Bilde den vollständigen Graphen H auf X

```

6      mit Gewichten  $d(i, j)$  für  $i, j \in X$  mit  $i \neq j$ .
7  Bestimme ein perfektes Matching  $K$  in  $H$  mit minimalem
8      Gewicht.
9  Bilde den Multigraphen  $G$ , der aus  $T$  durch
10     Hinzufügen aller Kanten aus  $K$  entsteht.
11  Bestimme einen Eulerschen Kreis  $C$  in  $G$ ;
12  Bestimme eine Rundreise  $R$  aus  $C$  durch Abkürzungen;
13  return Rundreise  $R$ .

```

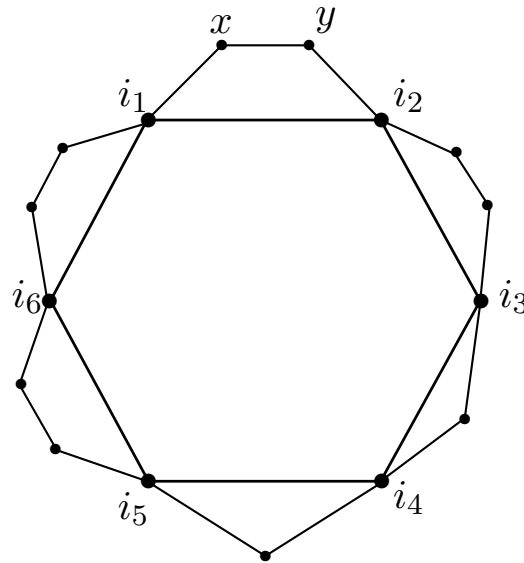
Satz 7.13. (Christofides): Der Algorithmus ΔTSP_2 hat eine multiplikative Güte 1.5; d.h. $d(R) \leq 1.5OPT(I)$.

Beweis. Es gilt $d(C) = w(T) + d(K) \leq OPT(I) + d(K)$. Man zeige nun, dass $d(K) \leq OPT(I)/2$. Daraus folgt dann die Behauptung.

Sei dazu $(i_1, \dots, i_{2m})$ mit $|X| = 2m$ die Reihenfolge, in der die Knoten von X in einer optimalen Tour δ von I durchlaufen werden. Betrachte die Matchings:

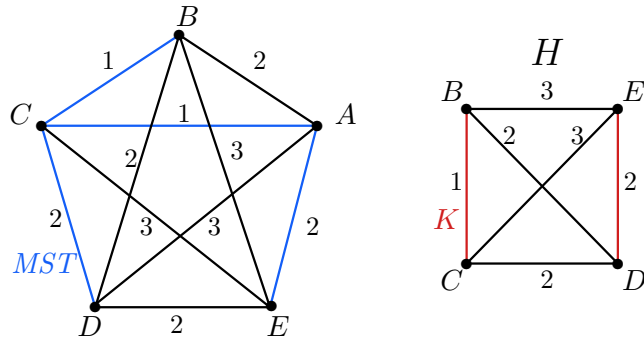
$$\begin{aligned}
 M_1 &= \{\{i_1, i_2\}, \{i_3, i_4\}, \dots, \{i_{2m-1}, i_{2m}\}\} \\
 M_2 &= \{\{i_2, i_3\}, \{i_4, i_5\}, \dots, \{i_{2m}, i_1\}\}
 \end{aligned}$$

Betrachte eine optimale Tour und die Knoten aus X :

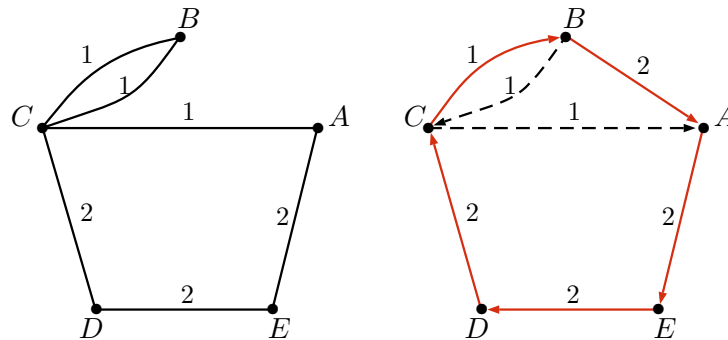


Wegen der Δ -Ungleichung gilt: $OPT(I) \geq d(i_1, i_2) + d(i_2, i_3) + \dots + d(i_{2m}, i_1) = d(M_1) + d(M_2)$. Daher gilt: $OPT(I) \geq d(M_1) + d(M_2) \geq 2d(K)$, da K ein perfektes Matching bezüglich X mit minimalem Gewicht ist. Deswegen gilt: $d(K) \leq OPT(I)/2$ und damit die Behauptung. $\square$

Beispiel 7.14. Zu unserem Graphen im obigen Beispiel erhalten wir den folgenden induzierten Graphen H rechts mit einem perfektem Matching K und Gewicht $w(K) = 3$:



Als Multigraphen G erhalten wir dann



Ein Eulerkreis in G ist $C = [C, B, C, A, E, D, C]$ und nach den Abkürzungen erhalten wir $R = [C, B, A, E, D, C]$ mit Länge $d(R) = 9$.

Lemma 7.15. Es gibt Beispiele für Eingaben I zu dem Traveling Salesman Problem mit $OPT(I) = n$, bei denen der Algorithmus ΔTSP_2 von Christofides eine Rundreise der Länge $(n - 1) + \lfloor n/2 \rfloor$ berechnet.

Beweis. zur Übung. □

7.2 Rucksackproblem

Dass das Rucksackproblem NP -vollständig ist, haben wir schon am Ende des letzten Kapitels gesehen. Hier wollen wir nun einige einfache Approximationsalgorithmen studieren.

Gegeben:

- n Gegenstände mit Gewichten $w_0, \dots, w_{n-1} \in \mathbb{N}$ und Gewinnen $p_0, \dots, p_{n-1} \in \mathbb{N}$.
- Rucksack der Kapazität $B \in \mathbb{N}$.

Gesucht: Teilmenge $I \subseteq \{0, \dots, n - 1\}$ mit $\sum_{i \in I} w_i \leq B$ und

$$\sum_{i \in I} p_i = \max \left\{ \sum_{i \in I'} p_i \mid I' \subseteq \{0, \dots, n - 1\}, \sum_{i \in I'} w_i \leq B \right\}.$$

Satz 7.16. *Das Entscheidungsproblem zum Rucksackproblem ist NP-vollständig; sogar wenn $w_i = p_i$ für alle $i = 0, \dots, n-1$.*

Beweis. siehe Kapitel Einführung in die Komplexitätstheorie. □

Zunächst betrachten wir einen einfachen Greedy Algorithmus (GA):

Algorithmus GA ($I = ((w_0, p_0), \dots, (w_{n-1}, p_{n-1}), B)$)

```

1  sortiere die Gegenstände, so dass  $p_0/w_0 \geq p_1/w_1 \geq \dots \geq p_{n-1}/w_{n-1}$ ;
2  setze  $S = \emptyset$ ;
3  for  $i = 0$  to  $n-1$  do
4    if  $(\sum_{j \in S} w_j) + w_i \leq B$  then
5       $S = S \cup \{i\}$ 
6    fi
7  od
8  return(S).
```

Es sei $GA(I)$ der Gesamtgewinn der durch Algorithmus GA berechneten Lösung $S \subseteq \{0, \dots, n-1\}$ und $OPT(I)$ der Gesamtgewinn einer optimalen Lösung zur Eingabe I .

Satz 7.17. (a) *Die Laufzeit des Algorithmus GA ist $\mathcal{O}(n \log n)$.*

(b) *Der Algorithmus GA hat eine multiplikative Güte größer gleich $B-1$ (d.h. $OPT(I)/GA(I) \geq (B-1)$).*

Der Algorithmus GA hat daher keine konstante Worst-Case Güte.

Beweis. Zu (a): hierzu speichere bei jedem Hinzufügen eines Gegenstands die momentan benutzte Kapazität K . Dann geht Schritt (2–8) in $\mathcal{O}(n)$ Zeit. Die Gesamtlaufzeit ergibt sich durch das Sortieren in $\mathcal{O}(n \log n)$.

Zu (b): Betrachte das folgende Beispiel: $I = ((w_0, p_0) = (1, 1), (w_1, p_1) = (B, B-1), B)$. Hier gilt $p_0/w_0 = 1$ und $p_1/w_1 = (B-1)/B < 1$. Der Algorithmus liefert hier $S = \{0\}$ mit Gewinn 1; optimal ist aber $S_{opt} = \{1\}$ mit Gewinn $B-1$.

In diesem Fall gilt $OPT(I)/GA(I) = B-1$. D.h. im allgemeinen Fall ist die Rate unbeschränkt. □

Hinweis: Man beachte, dass GA in dem obigen Beispiel das Element mit maximalem Profit p_{max} ignoriert hat. Aus diesem Grunde betrachten wir eine leichte Modifikation von GA, die sofort hilft die Approximationsgüte zu beschränken.

Algorithmus MGA ($I = ((w_0, p_0), \dots, (w_{n-1}, p_{n-1}), B)$)

```

1  berechne Lösung  $S_1$  mit Algorithmus GA,
2  berechne Lösung  $S_2 = \{j\}$  mit  $p_j = \max_{i \in \{0, \dots, n-1\}} p_i$ ,
3  wähle Lösung  $S \in \{S_1, S_2\}$  mit größerem Gewinn.
4  return(S).
```

Es sei $MGA(I)$ der Gesamtgewinn der durch Algorithmus MGA berechneten Lösung $S \subseteq \{0, \dots, n-1\}$ und $OPT(I)$ der Gesamtgewinn einer optimalen Lösung zur Eingabe I .

Satz 7.18. Die Worst-Case Güte von Algorithmus MGA ist 2 (d.h. $OPT(I)/MGA(I) \leq 2$ für alle Eingaben I).

Beweis. Betrachte relaxierte Version des Rucksackproblems, bei dem Gegenstände fraktional genommen werden dürfen. Es sei $OPT_f(I)$ der maximale Wert einer fraktionalen Lösung x^* . Es gilt dann $OPT(I) \leq OPT_f(I)$, da der Lösungsraum sich vergrößert und ein Max-Problem vorliegt.

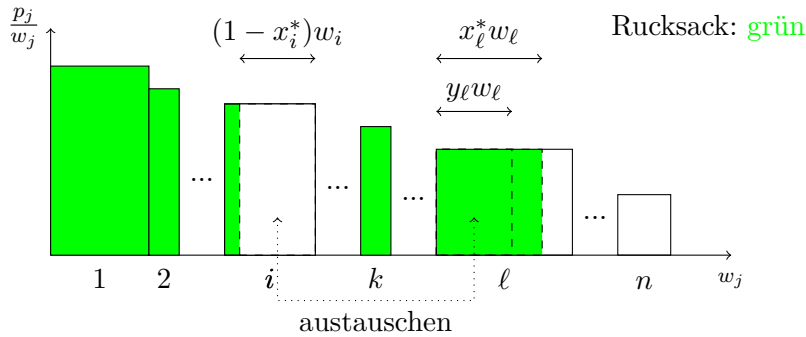
Eine fraktionale Lösung x^* von $\max \sum_{j=1}^n x_j p_j$ mit $\sum_j x_j w_j \leq B$ und $x_j \in [0, 1]$ für alle $j = 1, \dots, n$ nimmt Gegenstände in der sortierten Reihenfolge

$$p_1/w_1 \geq p_2/w_2 \geq \dots \geq p_n/w_n,$$

wobei die ersten k Gegenstände ganzzahlig (d.h. $x_j^* = 1$) und der letzte Gegenstand fraktional mit $x_{k+1}^* \in [0, 1)$ gewählt wird.

Beachte: Es gilt $\sum_{j=1}^k w_j \leq B$ und $\sum_{j=1}^{k+1} w_j > B$. Insgesamt haben wir $\sum_{j=1}^{k+1} x_j^* w_j = B$. Falls wir einen fraktionalen Gegenstand haben mit $x_{k+1}^* > 0$ ist, so gilt $\sum_{j=1}^k w_j < B$.

Im Folgenden zeigen wir, dass eine optimale fraktionalen Lösung die obige Form hat. Angenommen, wir haben ein Teilstück eines Gegenstands $\ell > k+1$ der Länge $x_\ell^* w_\ell$ im Rucksack mit $p_\ell/w_\ell < p_{k+1}/w_{k+1}$ und $x_\ell^* > 0$. Dann muss ein anderes Stück $(1 - x_i^*)w_i$ mit $i \leq k+1$ und $x_i^* < 1$ im Rucksack fehlen (wegen den obigen Bedingungen). Betrachte die minimale Länge $length = \min\{(1 - x_i^*)w_i, x_\ell^* w_\ell\}$. Das folgende Bild veranschaulicht die genaue Situation:



Dann gilt $length = y_i w_i = y_\ell w_\ell$ mit $y_i \in (0, 1 - x_i^*]$ und $y_\ell \in (0, x_\ell^*]$ und $p_i/w_i \geq p_{k+1}/w_{k+1} > p_\ell/w_\ell$.

Frage: Wie bestimmt man y_i, y_ℓ ?

Man beachte, dass $length = \min\{(1 - x_i^*)w_i, x_\ell^* w_\ell\} > 0$.

Fall 1: $(1 - x_i^*)w_i = length$. Hier setze $y_i = (1 - x_i^*)$ und $y_\ell = length/w_\ell$.

Fall 2: $x_\ell^* w_\ell = length$. Hier setze $y_\ell = x_\ell^*$ und $y_i = length/w_i$.

Ersetze nun die Länge $y_\ell w_\ell$ in der Lösung durch $y_i w_i$. Der Profitgewinn durch diesen Austausch ist dann $y_i p_i - y_\ell p_\ell$ und es gilt:

$$\begin{aligned} y_i p_i - y_\ell p_\ell &> y_i (p_\ell / w_\ell) w_i - y_\ell p_\ell \\ &= y_i w_i p_\ell / w_\ell - y_\ell p_\ell \\ &= y_\ell w_\ell p_\ell / w_\ell - y_\ell p_\ell \\ &= y_\ell p_\ell - y_\ell p_\ell = 0. \end{aligned}$$

D.h. der Profitgewinn ist echt positiv und damit kann ein solcher obiger Fall nicht auftreten.

Nun kommen wir zur Abschätzung der Approximationsgüte: Da $k + 1$ das erste Item ist, dass der Algorithmus GA nicht nimmt, gilt

$$OPT_f(I) \leq GA(I) + p_{k+1}.$$

Der Algorithmus MGA berechnet Lösung mit Wert $GA(I)$ oder $p_{max} \geq p_{k+1}$ (bzw. sogar genauer mit Wert $\max\{GA(I), p_{max}\}$). Daher gilt

$$\begin{aligned} OPT(I) &\leq OPT_f(I) \leq GA(I) + p_{k+1} \\ &\leq GA(I) + p_{max} \leq 2 \max\{GA(I), p_{max}\} \\ &= 2 \cdot MGA(I) \end{aligned}$$

□

Bemerkung 7.19. (a) Der Algorithmus MGA läuft in Zeit $\mathcal{O}(n \log n)$.

(b) Durch eine Modifikation von MGA können wir auch einen approximativen Algorithmus angeben mit Worst Case Güte 2, der in $\mathcal{O}(n)$ Zeit läuft.

Beweis. Wir verwenden als Idee zur Laufzeitverbesserung (b) hierzu den Median-Algorithmus um das sogenannte Split-Item $k + 1$ in $\mathcal{O}(n)$ Zeit zu finden. □

Der folgende Algorithmus verallgemeinert die Idee vom MGA Algorithmus, in dem er alle k -elementigen Teilmengen durchprobiert, in den Rucksack vorplaziert, wenn das möglich ist, und mit Hilfe vom GA Algorithmus die Restkapazität auffüllt.

Algorithmus von Sahni

Gegeben: Parameter $k \in \{0, \dots, n\}$.

Idee: Für jedes k definiere Algorithmus A_k wie folgt:

- (1) wähle eine Teilmenge S mit $|S| \leq k$ Gegenständen, die am Anfang im Rucksack liegen,
- (2) wende Algorithmus GA auf die übrigen Elemente an und füge die ausgewählten Gegenstände zusätzlich in den Rucksack ein.

Wende (1) + (2) auf alle möglichen Teilmengen S mit $|S| \leq k$ an und wähle Lösung mit maximalem Gewinn.

Die Technik nennt man *k-Enumeration*. Es sei $A_k(I)$ der Gewinn der Lösung, den der Algorithmus A_k von Sahni bezüglich einer Eingabe I und Parameter $k \in \{0, \dots, n\}$ produziert.

Satz 7.20. Für alle $k \geq 1$ gilt: A_k hat Güte $\leq 1 + 1/k$ und Laufzeit $T_{A_k}(n) = \mathcal{O}(n^{k+1})$.

Beweis. siehe Vorlesung Effiziente Algorithmen. $\square$

Idee für ein schnelleres Verfahren: Konstruiere aus dem exakten Algorithmus *Knapsack* aus Kapitel 2 zum Rucksackproblem einen polynomiellen approximativen Algorithmus.

Erinnerung: Das dynamische Programm zum Rucksackproblem hat eine nicht-polynomielle (bzw. eine sogenannte pseudo-polynomielle) Laufzeit $\mathcal{O}(n^2 \cdot p_{\max})$, wobei $p_{\max} = \max\{p_i | 0 \leq i \leq n-1\}$. Hierbei könnte $p_{\max}$ exponentiell in der Eingabegröße sein; siehe auch Bemerkung 2.20.

Idee zu Algorithmus A_K

- (1) konstruiere eine Eingabe I_K mit Gewinnen $p'_i = \lfloor p_i/K \rfloor$ (alle anderen Größen w_i und B bleiben gleich).
- (2) wende Algorithmus *Knapsack* auf Eingabe I_K an.

Der Algorithmus A_K berechnet eine Lösung $S \subseteq \{0, \dots, n-1\}$ mit Größe $\sum_{i \in S} w_i \leq B$ und maximalem Gewinn $\sum_{i \in S} p'_i$.

Wir entwickeln aus A_K ein Verfahren A_ϵ durch die Wahl

$$K = \frac{p_{\max}}{(1/\epsilon + 1)n}.$$

Satz 7.21. Der Algorithmus A_ϵ läuft in Zeit $\mathcal{O}(n^3/\epsilon)$ und hat eine Worst-Case Güte $OPT(I)/A_\epsilon(I) \leq 1 + \epsilon$.

Man nennt eine solche Familie von Algorithmen (A_ϵ) ein *vollständiges polynomielles Approximationsschema (FPTAS)*.

Beweis. (a) zur Laufzeit: Für die skalierten Profite gilt: $p'_{\max} = \max_i p'_i = \max_i \lfloor p_i/K \rfloor = \max_i \lfloor p_i/p_{\max}(1/\epsilon + 1)n \rfloor = \lfloor (1/\epsilon + 1)n \rfloor = \mathcal{O}(n/\epsilon)$.

Der Algorithmus *Knapsack* angewandt auf die skalierte Instanz hat dann eine Laufzeit von

$$\mathcal{O}(n^2 p'_{\max}) = \mathcal{O}(n^2 n/\epsilon) = \mathcal{O}(n^3/\epsilon).$$

- (b) zur Güte: Es gilt zunächst

$$OPT(I) \leq K \cdot OPT(I_K) + K \cdot n \quad (1)$$

Da $p'_i = \lfloor p_i/K \rfloor \geq p_i/K - 1$, haben wir $p_i \leq K \cdot (p'_i + 1)$ und deswegen haben wir

$$\begin{aligned}
OPT(I) &= \sum_{i \in I_{opt}} p_i \leq \sum_{i \in I_{opt}} (Kp'_i + K) \\
&\leq \sum_{i \in I_{opt}} Kp'_i + K \cdot n \leq K \cdot OPT(I_K) + K \cdot n
\end{aligned}$$

Beachte, dass I_{opt} eine zulässige Lösung von I_K ist und $(I_K)_{opt}$ i.a. ein höheren Gewinn als I_{opt} bezüglich Instanz I_K hat. Daneben gilt:

$$K \cdot OPT(I_K) \leq A_K(I) \quad (2)$$

Da $p'_i \leq p_i/K$ gilt

$$\begin{aligned}
K \cdot OPT(I_K) &= K \cdot \sum_{i \in (I_K)_{opt}} p'_i \leq K \cdot \sum_{i \in (I_K)_{opt}} p_i/K \\
&= \sum_{i \in (I_K)_{opt}} p_i \leq A_K(I).
\end{aligned}$$

Aus (1) + (2) folgt

$$OPT(I) \leq A_K(I) + K \cdot n.$$

Da alle $w_i \leq B$ (ansonsten kann Gegenstand i aus der Instanz gelöscht werden), gilt $OPT(I) \geq p_{max} = \max_i p_i$.

Wir erhalten

$$\begin{aligned}
\frac{OPT(I)}{A_\epsilon(I)} &\leq \frac{A_K(I) + Kn}{A_K(I)} \\
&= 1 + \frac{Kn}{A_K(I)} \\
&\leq 1 + \frac{Kn}{OPT(I) - Kn} \\
&\leq 1 + \frac{Kn}{p_{max} - Kn} \\
&= 1 + \epsilon.
\end{aligned}$$

Die letzte Gleichung gilt, da

$$\begin{aligned}
\frac{Kn}{p_{max} - Kn} &= \frac{\frac{p_{max}}{(1/\epsilon+1)n} \cdot n}{p_{max} - \frac{p_{max}}{(1/\epsilon+1)n} n} \\
&= \frac{\frac{1}{(1/\epsilon+1)}}{1 - \frac{1}{(1/\epsilon+1)}} = \frac{\frac{1}{1/\epsilon+1}}{\frac{1/\epsilon+1-1}{1/\epsilon+1}} \\
&= \frac{1}{1/\epsilon} = \epsilon.
\end{aligned}$$

□

Weitere Approximationsschema

- $\mathcal{O}(n^2/\epsilon)$ (Ibarra und Kim 1975).
- $\mathcal{O}(n \log(1/\epsilon) + 1/\epsilon^4)$ (Lawler 1979).

- $\mathcal{O}(n \min\{\log n, \log(1/\epsilon)\} + 1/\epsilon^2 \log(1/\epsilon) \min\{n, 1/\epsilon \log(1/\epsilon)\})$ (Kellerer, Pferschy 2004).
- $\tilde{\mathcal{O}}(n + (1/\epsilon)^{5/2})$ (Rhee 2015).
- $\tilde{\mathcal{O}}(n + (1/\epsilon)^{12/5})$ (Chan 2018).
- $\tilde{\mathcal{O}}(n + (1/\epsilon)^{9/4})$ (Jin 2019).

In der $\tilde{\mathcal{O}}$ Notation vernachlässigt man $\log(\cdot)$ -Terme in der Laufzeit.

Bemerkung 7.22. *Es gibt kein FPTAS mit einer Laufzeit $\mathcal{O}((n + 1/\epsilon)^{2-\delta})$, außer $(\min, +)$ Convolution hat einen Algorithmus mit einer subquadratischen Laufzeit (Cygan u.a. 2017 bzw. Künnemann u.a. 2017).*

7.3 Scheduling

Der Begriff *Scheduling* beschreibt die Problematik der Erstellung eines *Ablaufplans* bzw. *Schedules*, der Prozessen zeitlich begrenzt Ressourcen, wie etwa *Maschinen* bzw. *Prozessoren* oder *Speicher*, zuteilt. Dabei untersucht man die Optimierung bzw. Approximation diverser Zielfunktionen. Eine klassische Variante wird formal als $P||C_{\max}$ angegeben und lässt sich wie folgt formulieren.

Gegeben:

n Jobs $J = \{J_1, \dots, J_n\}$ mit Ausführungszeiten $p_1, \dots, p_n \in \mathbb{N}$ und m identische Maschinen.

Gesucht:

Partition von J in m Teilmengen $B_1, \dots, B_m$ mit *minimaler maximaler Last*

$$C_{\max} = \max_{1 \leq i \leq m} \sum_{J_j \in B_i} p_j.$$

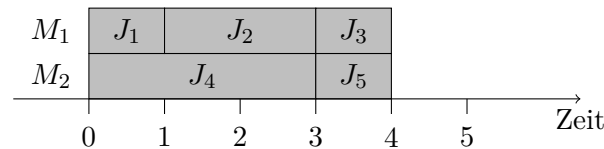
Den Wert $C_{\max}$ nennt man auch den *Makespan des Schedules*. Das Problem wird bezeichnet durch $P||C_{\max}$ bzw. $Pm||C_{\max}$, wenn die Anzahl der Maschinen m konstant ist. Im Folgenden betrachten wir eine Instanz für das Schedulingproblem $P||C_{\max}$:

Tab. 7.1: Schedulinginstanz

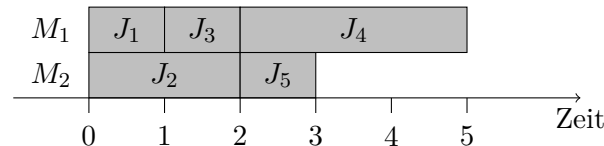
J_j	J_1	J_2	J_3	J_4	J_5	
p_j	1	2	1	3	1	$m = 2$

Für diese Instanz mit zwei Maschinen können wir etwa die beiden folgenden Ablaufpläne/Schedules als zulässig erkennen:

- $B_1 = \{J_1, J_2, J_5\}$, $B_2 = \{J_4, J_3\}$ mit Makespan 4.



- $B_1 = \{J_1, J_3, J_4\}$, $B_2 = \{J_2, J_5\}$ mit Makespan 5.



Man erkennt leicht, dass der erste Ablaufplan ein *optimaler* Ablaufplan ist. Der optimale Makespan ist also 4. Wie schwer ist nun dieses Problem? Der folgende Satz gibt uns Auskunft.

Satz 7.23. Das Schedulingproblem auf identischen Maschinen $Pm||C_{\max}$ ist NP-vollständig sogar für $m = 2$ Maschinen.

Beweis. Dies folgt durch eine Reduktion des Partitionsproblems. □

Man mache sich klar, dass daraus unmittelbar auch die NP-Vollständigkeit von $P||C_{\max}$ folgt. Wir wollen nun einen ersten Approximationsalgorithmus studieren, den wir LIST-SCHEDULING nennen.

Algorithmus ListScheduling($L=(J_1, \dots, J_n), m$)

```

1  for i=1 to m do
2     $E_i = 0$ ;  $B_i = \emptyset$ ;
3  od
4  for j=1 to n do
5    wähle Job  $J_j$  aus Liste L;
6    wähle Maschine  $M_i$  mit minimaler Last  $E_i$ ;
7     $B_i = B_i \cup \{J_j\}$ ;
8     $E_i = E_i + p_j$ ;
9  od
```

Es sei $LS(I)$ die Länge eines List Schedules und $OPT(I)$ die Länge eines optimalen Schedules zur Eingabe I .

Satz 7.24. (a) Für alle Eingaben $I = (L, m)$ gilt $LS(I)/OPT(I) \leq 2 - 1/m$.

(b) Es existiert eine Eingabe I^* mit $LS(I^*) = (2 - 1/m)OPT(I^*)$. Der Algorithmus hat also eine multiplikative Güte bzw. absolute Worst Case Güte von $2 - 1/m$.

Beweis. Zu (a): O.B.d.A. hat Maschine M_1 nach der Zuordnung die höchste Last $L = \sum_{j \in B_1} p_j$ (ansonsten nummeriere die Maschinen um). Es sei J_k der letzte Job, der auf Maschine M_1 fertig wird. Dann haben alle Maschinen eine Last $L_i \geq L - p_k$. Zu dem Zeitpunkt, wo J_k der Maschine M_1 zugeordnet wurde, hatte M_1 die kleinste Last $L - p_k$. Wir illustrieren dies in Abb. 7.1.

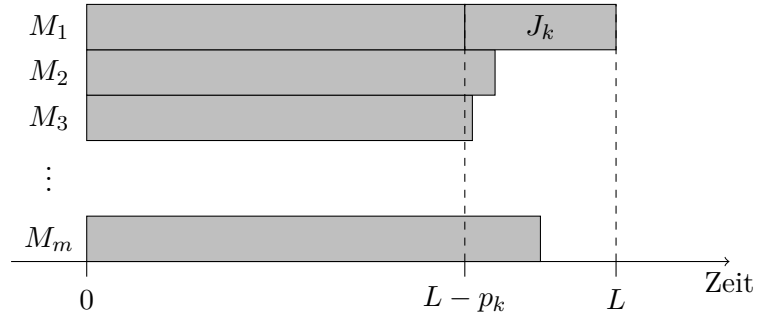


Abb. 7.1: Maschine M_1 hat mit J_k bzw. ohne J_k die größte Last bzw. kleinste Last

Daraus folgt sofort die folgende Abschätzung:

$$\sum_{i=1}^n p_i \geq m(L - p_k) + p_k.$$

Daneben gilt (da die gesamte Last $\sum p_i$ auf m Maschinen verteilt werden muss und mindestens eine Maschine eine Last $\geq \sum p_i / m$ hat):

$$\sum_{i=1}^n p_i / m \leq OPT(I).$$

Da nun $LS(I) = L$ ist, gilt

$$\begin{aligned} OPT(I) &\geq \sum_{i=1}^n p_i / m \geq m(L - p_k) / m + p_k / m \\ &= L - (1 - 1/m)p_k = LS(I) - (1 - 1/m)p_k \end{aligned}$$

Da $OPT(I) \geq p_k$ folgt nun $OPT(I) \geq LS(I) - (1 - 1/m)OPT(I)$ bzw.

$$LS(I) \leq (2 - 1/m)OPT(I).$$

Zu (b): zur Übung. □

Wir haben gesehen, dass ein erstaunlich einfacher Algorithmus schon eine Güte von 2 garantieren kann. Allerdings fällt auf, dass die beliebige Wahl des nächsten Jobs (in Zeile 5) mitunter schlecht sein kann. Es scheint besser zu sein, die Jobs absteigend ihrer Ausführungszeit nach zu platzieren, was uns zum folgenden Algorithmus LPT SCHEDULING führt.

Algorithmus LPT Scheduling($I=(J, m)$)

- 1 sortiere die Jobs in J so, dass $p_1 \geq p_2 \geq \dots \geq p_n$;
- 2 wende den List Scheduling Algorithmus an auf $I=(L, m)$ mit $L=(J_1, \dots, J_n)$

Es sei $LPT(I)$ die Länge eines LPT Schedules bezüglich einer Eingabe $I = (J, m)$.

Satz 7.25. *Der LPT Algorithmus hat eine absolute Worst Case Rate von $4/3 - 1/(3m)$; d.h.*

$$LPT(I) \leq (4/3 - 1/(3m))OPT(I). \quad (*)$$

Beweis. Annahme: Es existiert eine Jobmenge J und Maschinenzahl m , die die Behauptung $(*)$ nicht erfüllt. Der Satz gilt sofort für $m = 1$ (in diesem Fall gilt nämlich $LPT(I) = OPT(I)$). Daher nehmen wir an: $m \geq 2$ und n ist minimal (d.h. wir haben ein minimales Gegenbeispiel).

Annahme: Es existiert ein Job J_r mit $r < n$, für den die Fertigstellungszeit $f_r = LPT(I = (J, m))$ gilt. Betrachte $J' = \{J_1, \dots, J_r\}$ mit Liste $L' = (J_1, \dots, J_r)$. Dann gilt $LPT(I = (J, m)) = LPT(I' = (J', m))$ und der optimale Wert $OPT(I') \leq OPT(I)$. Damit haben wir

$$\frac{LPT(I')}{OPT(I')} \geq \frac{LPT(I)}{OPT(I)} > 4/3 - 1/(3m)$$

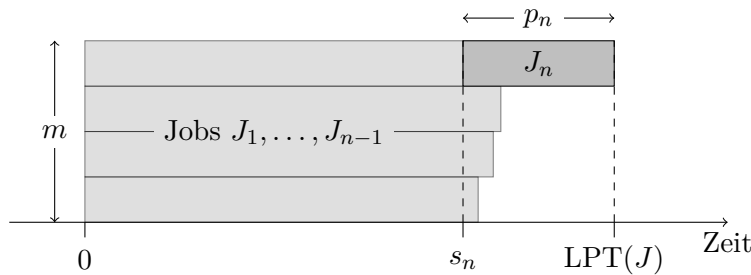
ein kleineres Gegenbeispiel gefunden. Widerspruch zur Minimalität von n .

Es gilt also $f_k < LPT(I = (J, m))$ für alle $k < n$. Desweiteren haben wir die Ungleichungen:

$$OPT(I) \geq \frac{1}{m} \sum_{i=1}^n p_i \quad (1)$$

$$\sum_{i=1}^{n-1} p_i \geq m s_n = m(LPT(I) - p_n), \quad (2)$$

wobei s_n der Startzeitpunkt von Job J_n ist.



Beachte: Alle Maschinen sind bis zum Zeitpunkt s_n voll beschäftigt (d.h. keine Ma-

schine ist idle vor dem Zeitpunkt s_n).

$$\begin{aligned}
 \frac{LPT(I)}{OPT(I)} &= \frac{s_n + p_n}{OPT(I)} \\
 &\leq_{(2)} \frac{p_n}{OPT(I)} + \frac{1}{mOPT(I)} \sum_{i=1}^{n-1} p_i \\
 &= \frac{(m-1)p_n}{mOPT(I)} + \frac{1}{mOPT(I)} \sum_{i=1}^n p_i \\
 &\leq_{(1)} \frac{(m-1)p_n}{mOPT(I)} + 1.
 \end{aligned}$$

Da (*) für I nicht gilt, folgt

$$1 + \frac{(m-1)p_n}{mOPT(I)} \geq \frac{LPT(I)}{OPT(I)} > 4/3 - 1/(3m).$$

Daraus folgt

$$\frac{(m-1)p_n}{mOPT(I)} > 1/3 - 1/(3m) = \frac{m-1}{3m}$$

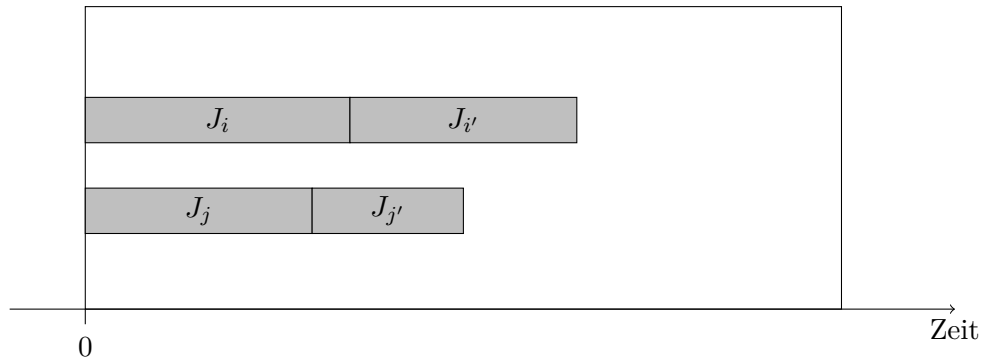
bzw.

$$p_n > \frac{OPT(I)}{3}.$$

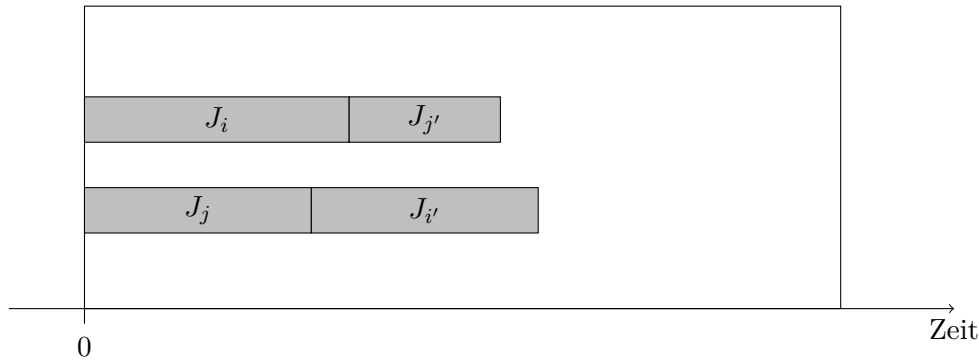
(d.h. der kleinste Job J_n ist größer als $OPT(I)/3$). Daraus folgt, dass keine Maschine in einem optimalen Schedule mehr als 2 Jobs ausführen kann.

Als nächstes transformieren wir einen optimalen Schedule D_{opt} für solch eine Jobmenge mit $p_n > OPT(I)/3$ in einen Schedule $\bar{D}_{opt}$, der zusätzliche Eigenschaften erfüllt.

Transformation I.1 für D_{opt} . Gegeben sei ein Schedule D_{opt} mit Makespan $C_{\max}$ und zwei Jobs J_i und $J_{i'}$ auf einer Maschine und zwei Jobs J_j und $J_{j'}$ auf einer anderen Maschine, wobei $p_i > p_j$ und $p_{i'} > p_{j'}$ ist.

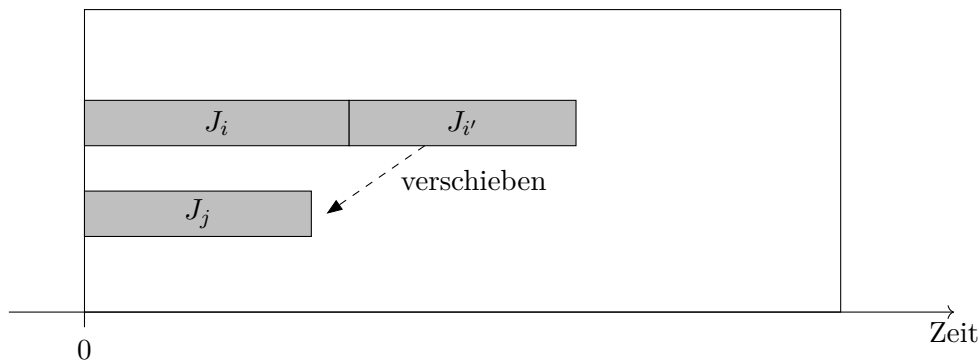


Wenn wir $J_{i'}$ und $J_{j'}$ vertauschen, so erhalten wir den folgenden Schedule D'_{opt} mit Makespan $C'_{\max} \leq C_{\max}$:

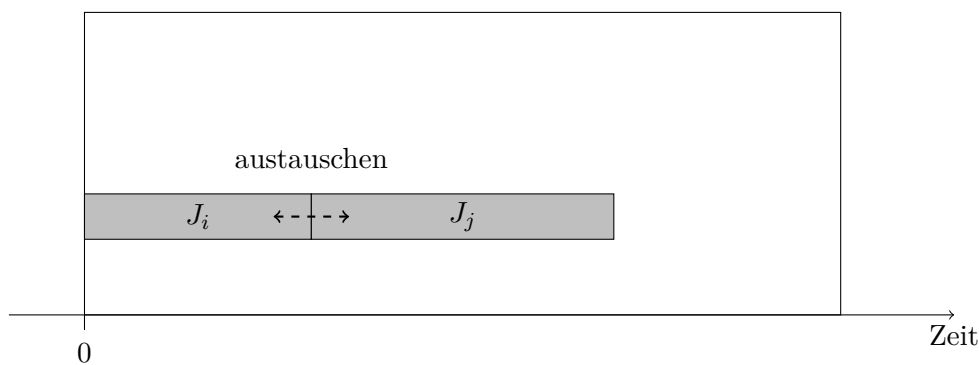


Transformation I.2 für D_{opt} . Gegeben sei ein Schedule D_{opt} mit Makespan $C_{\max}$ und zwei Jobs J_i und $J_{i'}$ auf einer Maschine und ein Job J_j auf einer anderen Maschine, wobei $p_i > p_j$ ist.

Wenn wir den Job $J_{i'}$ auf die andere Maschine verschieben, so erhalten wir einen Schedule D'_{opt} ebenso mit Makespan $C'_{\max} \leq C_{\max}$:



Transformation II für D_{opt} . Gegeben sei ein Schedule D_{opt} mit Makespan $C_{\max}$ und zwei Jobs J_i und $J_{i'}$ auf einer Maschine mit $p_i < p_j$. In diesem Fall vertauschen wir die Reihenfolge beider Jobs auf der entsprechenden Maschine:



Beachte: Keine der Transformationstypen I.1, I.2 oder II erhöht den Makespan.

Definition 7.26 (Lastfunktion). *Bei gegebenen Lastwerten L_i auf den Maschinen M_i in einem Schedule D ist die Lastfunktion $\mathcal{L}(D) = \sum_{1 \leq i < j \leq m} |L_i - L_j|$.*

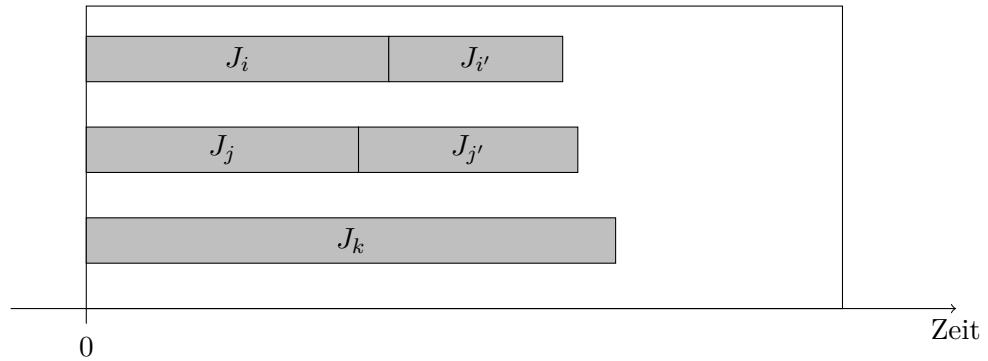
Bemerkung 7.27. (1) *Erhalten wir einen Schedule D' aus D durch eine Type I Transformation, so gilt $\mathcal{L}(D') < \mathcal{L}(D)$.*

(2) *Erhalten wir einen Schedule D' aus D durch eine Type II Transformation, so gilt $\mathcal{L}(D') = \mathcal{L}(D)$.*

Wir wenden nun auf D_{opt} alle möglichen Type I und Type II Transformationen an, bis keine Transformation mehr anwendbar ist. Es sei D^* der erzeugte Schedule. Ein solcher Schedule existiert, da

- (1) es nur endlich viele Anordnungen von n Jobs auf m Maschinen gibt,
- (2) wir zwischen zwei Type I Transformationen nur endlich viele Type II Transformationen einfügen können,
- (3) wir wegen Eigenschaft (1) in der obigen Bemerkung nur endlich viele Type I Transformationen ausführen können.

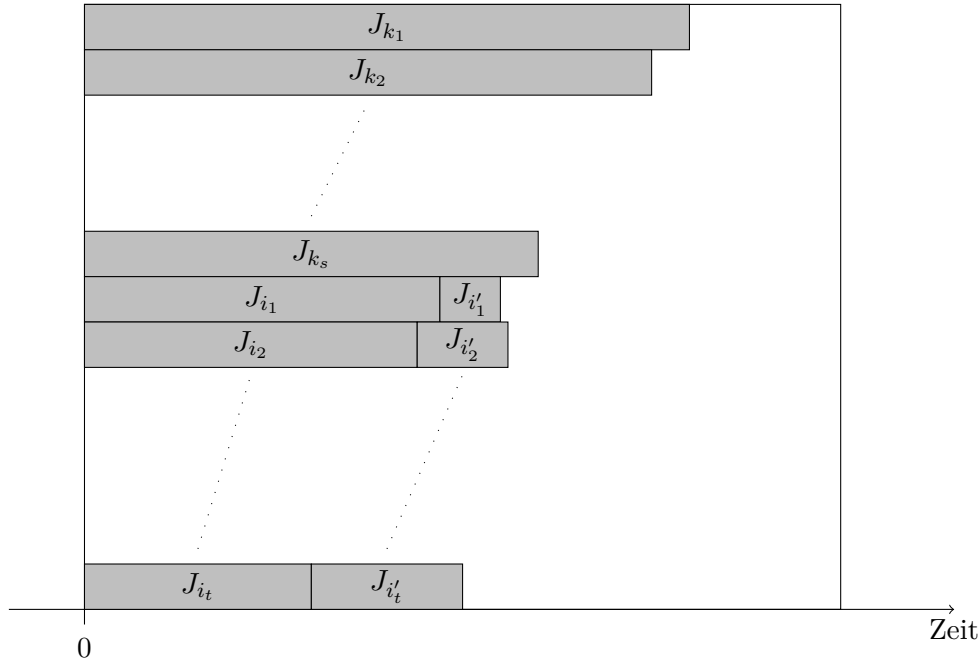
D^* erfüllt die folgende Bedingung:



Wenn wir zwei Jobs $J_i, J_{i'}$ auf einer Maschine, zwei Jobs $J_j, J_{j'}$ auf einer zweiten Maschine und einen einzelnen Job J_k auf einer dritten Maschine haben mit $p_i > p_j$, so folgt

$$\begin{aligned} p_{j'} &\geq p_{i'}, & p_i &\leq p_k, & p_i &\geq p_{i'}, & & (**) \\ p_j &\leq p_k, & p_j &\geq p_{j'}. \end{aligned}$$

Durch Umordnen der Maschinen in D^* erhalten wir einen Schedule $\bar{D}_{opt}$:



Hierbei gelten die Ungleichungen $p_{k_1} \geq \dots \geq p_{k_s}$ sowie $p_{i_1} \geq \dots \geq p_{i_t}$ und wegen (**) haben wir $p'_{i_1} \leq \dots \leq p'_{i_t}$ und $p_{k_s} \geq p_{i_1}$ sowie $p_{i_t} \geq p'_{i_t}$. Damit haben wir die folgende Ordnung der Jobs:

$$p_{k_1} \geq \dots \geq p_{k_s} \geq p_{i_1} \geq \dots \geq p_{i_t} \geq p'_{i_t} \geq \dots p'_{i_1}.$$

Der Schedule $\bar{D}_{opt}$ ist nun äquivalent zu einem List Schedule D_L bei gegebener Liste $L = (J_1, \dots, J_n)$ mit $p_1 \geq \dots \geq p_n$ (bis auf Vertauschen von Jobs mit gleicher Ausführungszeit). Daraus folgt aber

$$LPT(I = (J, m)) = OPT(I = (J, m)).$$

Wegen $LPT(I)/OPT(I) > 4/3 - 1/(3m)$ erhalten wir nun aber einen Widerspruch (d.h. es kann kein Gegenbeispiel für die Ungleichung (*) existieren). $\square$

Bemerkung 7.28. Für jede Anzahl m von Maschinen existiert eine Eingabe $I_m^* = (J^*, m)$ mit

$$\frac{LPT(I_m^*)}{OPT(I_m^*)} = \frac{4}{3} - \frac{1}{3m}.$$

Beweis. zur Übung. $\square$

Satz 7.29. Für jede Genauigkeit $\epsilon > 0$ gibt es einen approximativen Algorithmus A_ϵ für $P||C_{\max}$ mit $A_\epsilon(I) \leq (1 + \epsilon)OPT(I)$ und Laufzeit $2^{\mathcal{O}(1/\epsilon \log^2(1/\epsilon))} + \mathcal{O}(n)$.

Bemerkung 7.30. Eine Familie (A_ϵ) von solchen Algorithmen nennt man ein (**effizientes**) **polynomielles Approximationsschema (EPTAS)**.

Details: siehe Vorlesung *Effiziente Algorithmen*.

Übungsaufgaben

Übung 7.31. *Zeigen Sie:*

- *Ein zusammenhängender ungerichteter Graph $G = (V, E)$ enthält eine Eulertour genau dann, wenn jeder Knoten in V geraden Knotengrad hat.*
- *Ein stark zusammenhängender gerichteter Graph $G = (V, E)$ enthält eine Eulertour genau dann, wenn für jeden Knoten $v \in V$ gilt, dass $d_{in}(v) = d_{out}(v)$.*