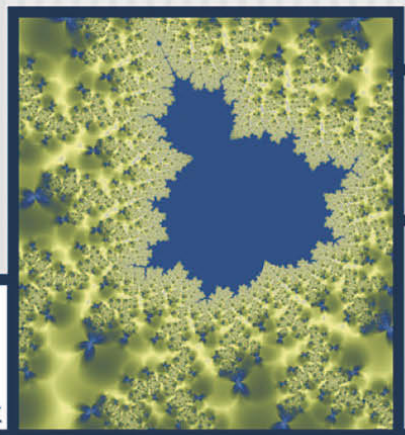
COMPANION WEBSITE CWS www.pearson-studium.de



informatik

Alexander Asteroth
Christel Baier

Theoretische Informatik

Eine Einführung in Berechenbarkeit, Komplexität und formale Sprachen mit 101 Beispielen

Pearson Studium

Pearson Studium

- Matchingproblem (einfache Variante für bipartite Graphen): Gegeben ist ein ungerichteter Graph $G = (V, E)$ und eine Partition $V = V_L \cup V_R$ mit disjunkten nicht leeren Mengen V_L und V_R, sodass jede Kante zwischen V_L und V_R verläuft; d.h. für jede Kante (zweielementige Teilmenge von V) $e \in E$ gilt:

$$e \cap V_L \neq \emptyset \text{ und } e \cap V_R \neq \emptyset.$$

  Gefragt ist, ob es eine Kantenmenge $M \subseteq E$ gibt, sodass jeder Knoten $v \in V$ auf höchstens einer Kante von M liegt und $|M| = |V_L|$.

Zeigen Sie, dass das Job-Rechnerproblem auf das Matchingproblem reduzierbar ist.

(b) Wir betrachten folgende beiden Varianten von linearen Programmierungsproblemen:

- Variante 1: Gegeben ist ein lineares Ungleichungssystem der Form

$$\begin{array}{ccccccccc}
a_{1,1}x_1 & + & a_{1,2}x_2 & + & \ldots & + & a_{1,m}x_m & \geq & b_1 \\
 & & & & \vdots & & & & \\
a_{n,1}x_1 & + & a_{n,2}x_2 & + & \ldots & + & a_{n,m}x_m & \geq & b_n \\
a_{n+1,1}x_1 & + & a_{n+1,2}x_2 & + & \ldots & + & a_{n+1,m}x_m & > & b_{n+1} \\
 & & & & \vdots & & & & \\
a_{n+l,1}x_1 & + & a_{n+l,2}x_2 & + & \ldots & + & a_{n+l,m}x_m & > & b_{n+l}\,,
\end{array}$$

  wobei $a_{i,j}$, $b_i \in \mathbb{Z}$. Dabei sind m, n, $l \geq 1$. Gefragt ist, ob es einen Vektor $(x_1, \ldots, x_m)$ mit ganzzahligen Koeffizienten x_j gibt, der die $n + l$ Ungleichungen löst.

- Variante 2: Gegeben ist ein lineares Ungleichungssystem der Form

$$\begin{array}{ccccccccc}
c_{1,1}x_1 & + & c_{1,2}x_2 & + & \ldots & + & c_{1,m}x_m & \leq & d_1 \\
 & & & & \vdots & & & & \\
c_{k,1}x_1 & + & c_{k,2}x_2 & + & \ldots & + & c_{k,m}x_m & \leq & d_k\,,
\end{array}$$

  wobei $c_{i,j}$, $d_i \in \mathbb{Z}$. Dabei sind m, $k \geq 1$. Gefragt ist, ob es einen Vektor $(x_1, \ldots, x_m)$ mit ganzzahligen Koeffizienten x_j gibt, der die k Ungleichungen löst.

Zeigen Sie, dass sich die Problemstellung von Variante 1 auf die Problemstellung von Variante 2 reduzieren lässt.

Aufgabe 2.3 Satz von Rice

Vervollständigen Sie den vorgestellten Beweis des Satzes von Rice (Satz 2.2.19, Seite 113), indem Sie folgende Aussage beweisen:

$$\text{Ist } f_\perp \notin \mathcal{S}, \text{ so gilt } H_\varepsilon \leq L_{\mathcal{S}}.$$

Aufgabe 2.4 Berechenbarkeit

Zeigen Sie die Existenz einer bijektiven Abbildung $\Phi : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, die samt der beiden Komponenten Ψ_1, Ψ_2 der Umkehrfunktion berechenbar ist.

Aufgabe 2.5 NTM

Geben Sie (präzise) eine Ein- oder Mehrband-NTM an, die die Sprache

$$L = \{xx^R : x \in \{0,1\}^*\}$$

in linearer Zeit entscheidet. Skizzieren Sie den Berechnungsbaum für die Wörter

$$w = 0110 \text{ und } w' = 0111.$$

Aufgabe 2.6 NTM

Zeigen Sie, dass sich jede NTM $\mathcal{T}$ durch eine NTM $\mathcal{T}'$ simulieren lässt, die in jedem Schritt höchstens zwei Folgekonfigurationen hat und sodass

$$T_{\mathcal{T}'}(n) = \mathcal{O}(T_{\mathcal{T}}(n)).$$

Es genügt eine informelle Beschreibung der simulierenden NTM $\mathcal{T}'$.

Zur Erinnerung: $T_{\mathcal{T}}$ bezeichnet die Kostenfunktion für $\mathcal{T}$ in Abhängigkeit der Eingabegröße; d.h. die Funktion $T_{\mathcal{T}} : \mathbb{N} \to \mathbb{N}$,

$$T_{\mathcal{T}}(n) = \max\{t_{\mathcal{T}}(w) : w \in \Sigma^*, |w| \leq n\}$$

(wobei Σ das Eingabealphabet von $\mathcal{T}$ bezeichnet).

Aufgabe 2.7 Nichtdet. Algorithmen

Geben Sie möglichst einfache nichtdeterministische Algorithmen (umgangssprachliche Formulierung) für folgende Problemstellungen an.

(a) Graphfärben: Gegeben ist ein ungerichteter endlicher Graph G und eine natürliche Zahl k. Gefragt ist, ob es eine Färbung der Knoten von G mit höchstens k Farben gibt, so dass benachbarte Knoten unterschiedlich gefärbt sind.

(b) Zusammengesetztheitsproblem: Gegeben ist eine natürliche Zahl $n \geq 3$. Gefragt ist, ob n zusammengesetzt (keine Primzahl) ist.

(c) Hamiltonwegproblem: Gegeben ist ein endlicher Digraph G. Gefragt ist, ob es einen Hamiltonweg in G gibt.

Zur Erinnerung: Ein Hamiltonweg ist ein einfacher Pfad, der jeden Knoten von G genau einmal besucht.

(d) Gegeben ist eine aussagenlogische Formel α. Gefragt ist, ob α nicht gültig ist, d.h. ob es eine Belegung μ gibt, unter der α falsch ist.

Aufgabe 2.8 Halteproblemvarianten

Zeigen Sie, dass folgende Sprachen unentscheidbar sind:

(a) $H_\forall = \{w \in \{0,1\}^* : \mathcal{T}_w \text{ hält für alle Eingaben an}\}$

(b) $H_\exists = \{w \in \{0,1\}^* : \text{es gibt eine Eingabe } x, \text{ für die } \mathcal{T}_w \text{ anhält}\}$

(c) $\overline{H_\forall}$

(d) $\overline{H_\exists}$

Hinweis: Mit einer Reduktion des Halteproblems auf die in (a) und (b) genannten Probleme ist man gut beraten.

Aufgabe 2.9 Äquivalenzproblem

Zeigen Sie, dass das Äquivalenzproblem für DTMs

$$\ddot{A}quiv = \{w\#v : w, v \in \{0,1\}^*,\ f_{\mathcal{T}_w} = f_{\mathcal{T}_v}\}$$

nicht semientscheidbar ist.

Hinweis: Benutzen Sie das Reduktionsprinzip für semientscheidbare Sprachen (Teil (b) von Satz 2.2.5) in der Form

$$L \leq K \text{ und L } \textit{nicht} \text{ semientscheidbar} \Longrightarrow K \textit{ nicht} \text{ semientscheidbar.}$$

Aufgabe 2.10 Semientscheidbarkeit

Zeigen Sie die Äquivalenz der folgenden Aussagen.

(a) L ist semientscheidbar,

(b) es gibt eine partielle berechenbare Funktion $f : \Sigma^* \to \Sigma^*$, sodass für alle $x \in \Sigma^*$ gilt:

$$f(x) \neq \perp \quad \text{gdw} \quad x \in L,$$

(c) es gibt eine totale berechenbare Funktion $g : \mathbb{N} \to \Sigma^*$ mit $g(\mathbb{N}) = L$.

Dabei ist Σ ein Alphabet und $\emptyset \neq L \subseteq \Sigma^*$.

Aufgabe 2.11 NTM

Gegeben ist folgende NTM

$$\mathcal{T} = (\{q_0, q_1, q_2, q_F\}, \{a, b, c\}, \{a, b, c, \square\}, \delta, q_0, \square, \{q_F\}),$$

wobei

$$\begin{aligned}
\delta(q_0, a) &= \{(q_0, c, N)\} \\
\delta(q_0, b) &= \{(q_2, b, R), (q_0, b, N)\} \\
\delta(q_0, c) &= \{(q_0, c, R), (q_1, \square, N)\} \\
\delta(q_0, \square) &= \{(q_0, \square, N), (q_F, \square, L)\} \\
\delta(q_1, \square) &= \{(q_1, \square, N)\} \ .
\end{aligned}$$

In allen verbleibenden Fällen ist $\delta(\cdot) = \emptyset$.

(a) Skizzieren Sie die Berechnungsbäume von $\mathcal{T}$ für die Eingabewörter

$$w_1 = \varepsilon, \quad w_2 = aca, \quad w_3 = bbac \ .$$

(b) Welche Sprache akzeptiert $\mathcal{T}$? (Geben Sie eine umgangssprachliche Charakterisierung von $\mathcal{L}(\mathcal{T})$ an.)

Aufgabe 2.12 Semientscheidungsverfahren

Geben Sie ein deterministisches Semientscheidungsverfahren und ein (rekursives) Aufzählungsverfahren für die Sprache

$$H_\exists = \{w \in \{0,1\}^* : \text{es existiert ein } x \in \{0,1\}^*, \text{sodass } \mathcal{T}_w \text{ bei Eingabe } x \text{ anhält}\}$$

an. Es genügt eine verbale Beschreibung der Verfahren, in der die wesentlichen Ideen skizziert sind.

Teil II

Komplexität

Die Entscheidbarkeit nützt für praktische Anwendungen recht wenig, wenn keine effizienten Entscheidungsverfahren verfügbar sind. In den folgenden beiden Kapiteln untersuchen wir, was es bedeutet, dass ein Problem *effizient lösbar* ist.

Üblicherweise orientiert man sich bei der Kostenanalyse eines Algorithmus entweder an dem schlimmsten oder an dem durchschnittlichen Fall. In diesem Buch beschränken wir unsere Effizienzbetrachtungen auf den *schlimmsten Fall*.

Bereits in Abschnitt 1.3 haben wir gesehen, dass die unterschiedlichen Rechnermodelle zwar zu unterschiedlichen Laufzeiten und unterschiedlichem Platzbedarf führen, jedoch entsteht durch die gegenseitigen Simulationen jeweils nur ein polynomieller Mehraufwand.[1] Aus diesem Grund betrachtet man *Komplexitätsklassen*, die unter polynomiellen Transformationen abgeschlossen sind. Als effizient lösbar werden genau solche Probleme angesehen, die sich mit einem Verfahren polynomieller Laufzeit lösen lassen. Selbstverständlich können auch Polynomialzeitalgorithmen für praktische Anwendungen ineffizient sein. Beispielsweise wird man einen Algorithmus mit der Laufzeit $\Theta(n^{1000})$ als indiskutabel schlecht ansehen. Jedoch ist es meistens so, dass die Laufzeit von Polynomialzeitalgorithmen für praxisrelevante Probleme durch ein Polynom geringen Grads beschränkt ist.

Polynomiell zeitbeschränkte Algorithmen: In Kapitel 2 haben wir konsequent zwischen umgangssprachlich formulierten JA/NEIN-Problemen und der präzisen Formulierung als Wortproblem unterschieden. Wir werden im Folgenden zu den lässigen Formulierungen von JA/NEIN-Problemen zurückkehren und nur vereinzelt darauf hinweisen, welche Kodierung in dem betreffenden Fall geeignet ist. Dabei werden wir jedoch im

[1] Wir erinnern an die Ergebnisse aus Abschnitt 1.3. Dort haben wir gesehen, dass polynomiell zeitbeschränkte Turingmaschinen durch polynomiell zeitbeschränkte Registermaschinen simuliert werden können und umgekehrt, vorausgesetzt, für Registermaschinen wird das logarithmische Kostenmaß zu Grunde gelegt.

Auge behalten, was es bedeutet, dass ein Algorithmus (etwa in Pseudo-Code) polynomielle Laufzeit hat.[2]

Die Eingabegröße eines Algorithmus wird gemessen an der Darstellungsgröße N der *Eingabe einer entsprechenden Turingmaschine*, die Zahlen, Nummern etc. binär kodiert. Wir sprechen von einem *polynomiell zeitbeschränkten Algorithmus*, wenn seine Kostenfunktion $T(N)$ unter dem *logarithmischen Kostenmaß* durch ein Polynom beschränkt ist. Das Argument N steht für die Eingabegröße. Auf das angenehmere uniforme Kostenmaß können wir für den Nachweis polynomieller Laufzeit nur unter gewissen Bedingungen zurückgreifen. Am Ende von Abschnitt 1.1.20 (Seite 35) haben wir ein Kriterium für Registermaschinen angegeben, das sich auf Algorithmen in Pseudo-Code übertragen lässt. Im Wesentlichen war die Forderung, dass — neben polylogarithmischer Laufzeit unter dem uniformen Kostenmaß — in jedem Rechenschritt nur polynomiell viele Bits verarbeitet werden. Diese Bedingungung ist für viele Algorithmen erfüllt.

Beispielsweise ist für das Mustererkennungsproblem (vgl. Beispiel 2.3.9, Seite 121) die Darstellungsgröße der Eingabe die Länge n des Texts plus die Länge k des Musters. Die Laufzeit $\Theta(nk)$ des naiven Verfahrens unter dem uniformen Kostenmaß induziert polynomielle Laufzeit, gemessen mit dem logarithmischen Kostenmaß.

Entsprechendes gilt für die einfachen Graphalgorithmen wie z. B. Tiefensuche, Breitensuche oder Zyklentest. In diesen Fällen müssen wir zur Ermittlung der Eingabegröße eine Kodierung der Eingabe vornehmen. Beispielsweise kann für Graphalgorithmen eine wie in Kapitel 2 (Seite 86) vorgestellte Kodierung verwendet werden. Die Darstellungsgröße des Graphen ist dann $\Theta(m \log n)$, wobei m die Kantenanzahl und n die Knotenanzahl ist. Die Laufzeit $\Theta(n+m)$ unter dem uniformen Kostenmaß der Tiefen- oder Breitensuche induziert polynomielle Laufzeit unter dem logarithmischen Kostenmaß, wenn man $m \geq n$ voraussetzt.

Entscheidungsprobleme versus Optimierungsprobleme

In Kapitel 2 haben wir lediglich Entscheidungsprobleme betrachtet. Tatsächlich liegen aber oftmals Probleme vor, in denen nicht nach den Antworten »JA« oder »NEIN« gefragt ist, sondern eine komplexere Lösung gesucht ist. In vielen Fällen liegt ein Optimierungsproblem vor, in dem eine Lösung gesucht ist, die eine gewisse Zielfunktion minimiert oder maximiert. Beispiele sind das Handlungsreisendenproblem (TSP), bei dem eine kürzeste Tour in einem vorgegebenen Wegenetz (Graph) zu bestimmen ist, oder das Graphfärbeproblem (GFP), bei dem ein gegebener Graph mit einer möglichst geringen Zahl an Farben so eingefärbt werden soll, dass benachbarte Knoten mit unterschiedlichen Farben gefärbt sind. Für die optimalen Lösungen sind nur Algorithmen mit exponentieller Laufzeit bekannt. Bevor wir uns der Frage widmen, ob bessere Algorithmen für diese Probleme entworfen werden können, machen wir uns klar, dass die

[2] Um Missverständnisse zu vermeiden, erwähnen wir, dass der Begriff *Algorithmus* stets für einen deterministischen Algorithmus gebraucht wird. Für Verfahren, die das Konzept von Nichtdeterminismus verwenden, werden wir stets den Zusatz »nichtdeterministisch« machen.

komplexitätstheoretische Schwierigkeit dieser und vieler anderer Probleme unverändert bleibt, wenn man die Fragestellung zu einem JA/NEIN-Problem vereinfacht.

Drei Varianten für »schwierige« Probleme: Probleme wie TSP, RP oder GFP können in drei Varianten betrachtet werden.

1. Entscheidungsproblem: Gibt es eine Lösung mit einem konkreten Wert?
2. Optimierungsproblem (Typ 1): Bestimme den Wert einer optimalen Lösung.
3. Optimierungsproblem (Typ 2): Bestimme eine optimale Lösung.

Beispielsweise kann man das Graphfärbeproblem so formulieren, dass für einen gegebenen Graphen G und eine gegebene Konstante k geprüft werden soll, ob es eine Knotenfärbung für G mit k Farben gibt. Typ 1 der Optimierungsvariante fragt nach der minimalen Anzahl an Farben, die für eine Färbung von G notwendig ist. In Typ 2 der Optimierungsvariante ist eine optimale Knotenfärbung gesucht.

Zunächst scheint es so, als ob die Probleme zunehmend schwieriger sind. Jede Lösung der Optimierungsvariante des zweiten Typs induziert offenbar eine Lösung für die beiden anderen Varianten. Entsprechend lässt sich die Entscheidungsvariante mithilfe der Lösung für die Optimierungsvariante des ersten Typs oftmals sehr leicht beantworten. Dennoch sind (in vielen Fällen) alle drei Varianten aus komplexitätstheoretischer Sicht gleich schwierig. Wir machen uns diesen Sachverhalt an einem Beispiel für folgendes »schwierige« Problem klar.

Das Cliquenproblem: Gegeben ist ein ungerichteter Graph $G = (V, E)$. Eine *Clique* in G ist eine Knotenmenge $V' \subseteq V$, sodass je zwei Knoten in V' über eine Kante miteinander verbunden sind. Die Größe einer Clique V' ist $|V'|$ (die Anzahl an Knoten in V').

Abbildung 3.0.1 Cliquenproblem

Die drei Varianten des Cliquenproblems sind in 3.0.2 angegeben.

Wie für das Graphfärbeproblem induziert eine polynomiell zeitbeschränkte Lösung für die Optimierungsvariante (Typ 2) eine polynomiell zeitbeschränkte Lösung für die Optimierungsvariante (Typ 1), und diese wiederum eine polynomiell zeitbeschränkte Lösung für die Entscheidungsvariante. Folgendes Lemma zeigt, dass auch die Umkehrungen gelten.

Tabelle 3.0.2 [Die drei Varianten des Cliquenproblems]

1. Entscheidungsvariante:
   - Gegeben: Ungerichteter Graph G und eine ganze Zahl $k \geq 1$
   - Gefragt: Gibt es in G eine Clique der Größe k?
2. Optimierungsvariante (Typ 1):
   - Gegeben: Ungerichteter Graph G
   - Gesucht: Maximale Cliquengröße in G
3. Optimierungsvariante (Typ 2):
   - Gegeben: Ungerichteter Graph G
   - Gesucht: Clique in G mit maximaler Größe

Lemma 3.0.3

Die drei Varianten des Cliquenproblems haben aus komplexitätstheoretischer Sicht denselben Schwierigkeitsgrad.

(a) Wenn die Optimierungsvariante (Typ 1) des Cliquenproblems in polynomieller Zeit lösbar ist, dann auch die Optimierungsvariante (Typ 2).

(b) Wenn die Entscheidungsvariante des Cliquenproblems in polynomieller Zeit lösbar ist, dann auch die beiden Optimierungsvarianten.

Beweis: ad (a): Wir nehmen an, dass uns ein polynomiell zeitbeschränkter Algorithmus für die Optimierungsvariante (Typ 1) vorliegt. Diesen verwenden wir in Algorithmus 3.0.4 (Seite 135) und erhalten somit einen Algorithmus für die Optimierungsvariante (Typ 2). (Dieser ist für den Fall $E \neq \emptyset$ formuliert. Für $E = \emptyset$ und $V \neq \emptyset$ ist jede einelementige Knotenmenge eine Clique maximaler Größe.)

Seien n die Knotenanzahl, m die Kantenanzahl in G und $T(n, m)$ die Rechenzeit, die der Algorithmus für die Optimierungsvariante vom Typ 1 benötigt. Die Darstellungsgröße des Graphen ist

$$N = \Theta(m \log n).$$

Die Kosten für Algorithmus 3.0.4 sind $\Theta(m \cdot T(n, m))$. Ist $T(n, m) = \mathcal{O}(poly(N))$, so ist auch

$$m \cdot T(n, m) = \mathcal{O}(poly(N)).$$

ad (b): Wir nehmen an, dass uns ein polynomiell zeitbeschränkter Algorithmus für die Entscheidungsvariante vorliegt. Um die maximale Cliquengröße zu berechnen, wenden wir Algorithmus 3.0.5 an.

Algorithmus 3.0.4 [Algorithmus für das Cliquenproblem (Opt., Typ 2)]

Berechne die maximale Cliquengröße k_{opt} in G;
Markiere alle Kanten in G als unbesichtigt;
$E' := E$; (* E ist die Kantenmenge von G *)
REPEAT
  Wähle eine noch nicht besichtigte Kante $e \in E'$;
  Bestimme die maximale Cliquengröße k in dem Graphen $(V, E' \setminus \{e\})$;
  IF $k = k_{opt}$ **THEN**
    $E' := E' \setminus \{e\}$; (* Kante e wird in maximaler Clique *nicht* benötigt *)
  ELSE
    Markiere e als besichtigt; (* Kante e wird in maximaler Clique benötigt *)
  FI
UNTIL alle Kanten in E' wurden besichtigt;
$V' :=$ Menge aller Endknoten von Kanten in E';
Gib V' aus.

Algorithmus 3.0.5 [Algorithmus für das Cliquenproblem (Opt., Typ 2)]

$k_{opt} := 0$;
FOR $k = 1, \ldots, \min\{|V|, |E| + 1\}$ **DO**
  IF es gibt eine Clique der Größe k **THEN**
    $k_{opt} := k$
  FI
OD
Gib k_{opt} aus.

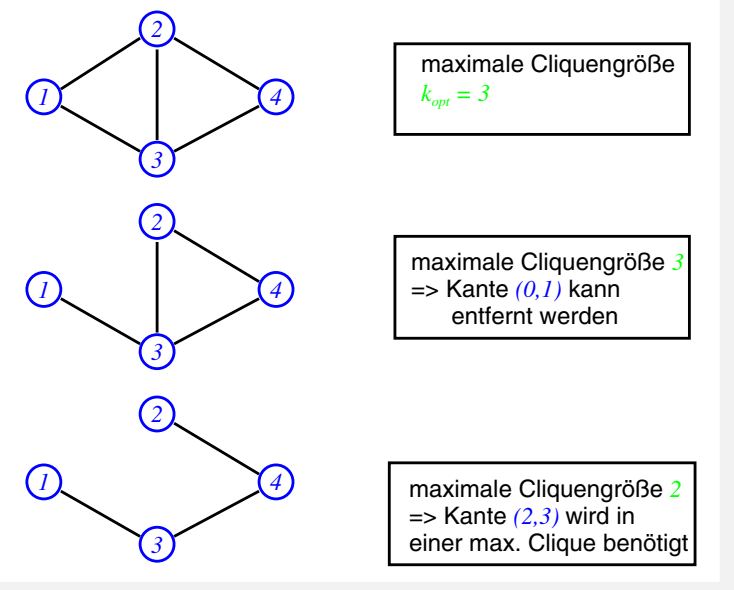
Man überlegt sich leicht, dass die Kosten polynomiell beschränkt sind, falls der Test, ob es eine Clique fester Größe gibt, in polynomieller Zeit durchführbar ist. □

Beispiel 3.0.6. [Cliquenproblem] Wir demonstrieren die Arbeitsweise von Algorithmus 3.0.4 (Seite 135) an einem Beispiel.

- Im ersten Schritt berechnen wir die maximale Cliquengröße 3.
- Die Kante $(1, 2)$ wird nicht in einer maximalen Clique benötigt, da die Knoten 2, 3, 4 eine Clique bzgl. der Kantenmenge $E' = E \setminus \{(1, 2)\}$ bilden. Die Kante (1,2) wird also gestrichen.
- In den folgenden Schritten stellen wir fest, dass die Kanten $(2, 3)$, $(3, 4)$ oder $(2, 4)$ nicht entfernt werden können; wohl aber die Kante $(1, 3)$.

Das skizzierte Verfahren liefert die Knotenmenge $\{2, 3, 4\}$ als Clique maximaler Größe.
■

Abbildung 3.0.7 Polynomielle Reduktion von CP-Typ2 auf CP-Typ1



Für eine Vielzahl von Problemen können derartige Überlegungen belegen, dass die Untersuchung der einfacher erscheinenden Entscheidungsvariante für komplexitätstheoretische Untersuchungen mit der eigentlichen Fragestellung gleichwertig ist. Dies rechtfertigt es, sich im Folgenden auf Entscheidungsprobleme zu beschränken.

KAPITEL 3

Komplexitätsklassen

Im Folgenden definieren wir Zeit- und Platzkomplexitätsklassen, die bzgl. polynomieller Transformationen abgeschlossen und damit unabhängig vom zu Grunde gelegten Rechnermodell sind. Zur Formalisierung werden Turingmaschinen verwendet.

Die Elemente von Komplexitätsklassen sind JA/NEIN-Probleme (und nicht etwa Algorithmen). In der formalisierten Darstellung einer Komplexitätsklasse nehmen wir eine Beschreibung der JA/NEIN-Probleme durch formale Sprachen an (s. Seite 85 ff), auch wenn wir diese nicht explizit angeben.

3.1 Zeitkomplexität

In Abschnitt 1.2 (Def. 1.2.23, Seite 56) haben wir die Rechenzeit $t_{\mathcal{T}}(x)$ einer DTM $\mathcal{T}$ als die Anzahl an Konfigurationswechseln, die $\mathcal{T}$ bei Eingabe x durchführt, definiert. Die Abbildung $t_{\mathcal{T}} : \Sigma^* \to \mathbb{N} \cup \{\infty\}$ induziert eine Abbildung $T_{\mathcal{T}} : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$, welche die Rechenzeit an der Eingabegröße misst.

Rechenzeit von NTMs: Eine entsprechende Definition können wir für NTMs angeben: Sei $\mathcal{T}$ eine Einband-NTM mit dem Eingabealphabet Σ. Wir definieren die Funktion $t_{\mathcal{T}} : \Sigma^* \to \mathbb{N} \cup \{\infty\}$ durch

$$t_{\mathcal{T}}(x) \;=\; \text{maximale Länge einer Berechnung von } \mathcal{T} \text{ für } x.$$

Offenbar ist $t_{\mathcal{T}}(x)$ die Höhe des Berechnungsbaums von $\mathcal{T}$ für x. Wir erweitern $t_{\mathcal{T}}$ zu einer Abbildung

$$T_{\mathcal{T}} : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$$

in Abhängigkeit der Eingabegröße.

$$T_{\mathcal{T}}(n) \;=\; \max\{t_{\mathcal{T}}(x) : x \in \Sigma^*, |x| \leq n\}.$$

In analoger Weise lässt sich die Laufzeit von NTMs mit zwei oder mehr Bändern definieren.

Die so definierte Rechenzeit nichtdeterministischer Turingmaschinen lässt sich auf umgangssprachlich formulierte nichtdeterministische Algorithmen übertragen. Die worstcase Laufzeit eines nichtdeterministischen Entscheidungsverfahrens ergibt sich, indem man die Kosten $T(n)$ für Eingaben der Größe n ermittelt, die im Falle einer längsten möglichen Berechnung für eine Eingabe der Größe $\leq n$ entstehen können. Wie für deterministische Algorithmen müssen wir das logarithmische Kostenmaß zu Grunde legen