

3 Komplexitätsklassen und ihre Beziehungen zueinander

3.1 Wichtige Komplexitätsklassen

Wir verwenden hier die gebräuchlichen Schreibweisen und Abkürzungen im Zusammenhang mit Komplexitätsklassen:

$$\mathbf{L} = \mathbf{DSPACE}(\log n), \quad \mathbf{NL} = \mathbf{NSPACE}(\log n),$$

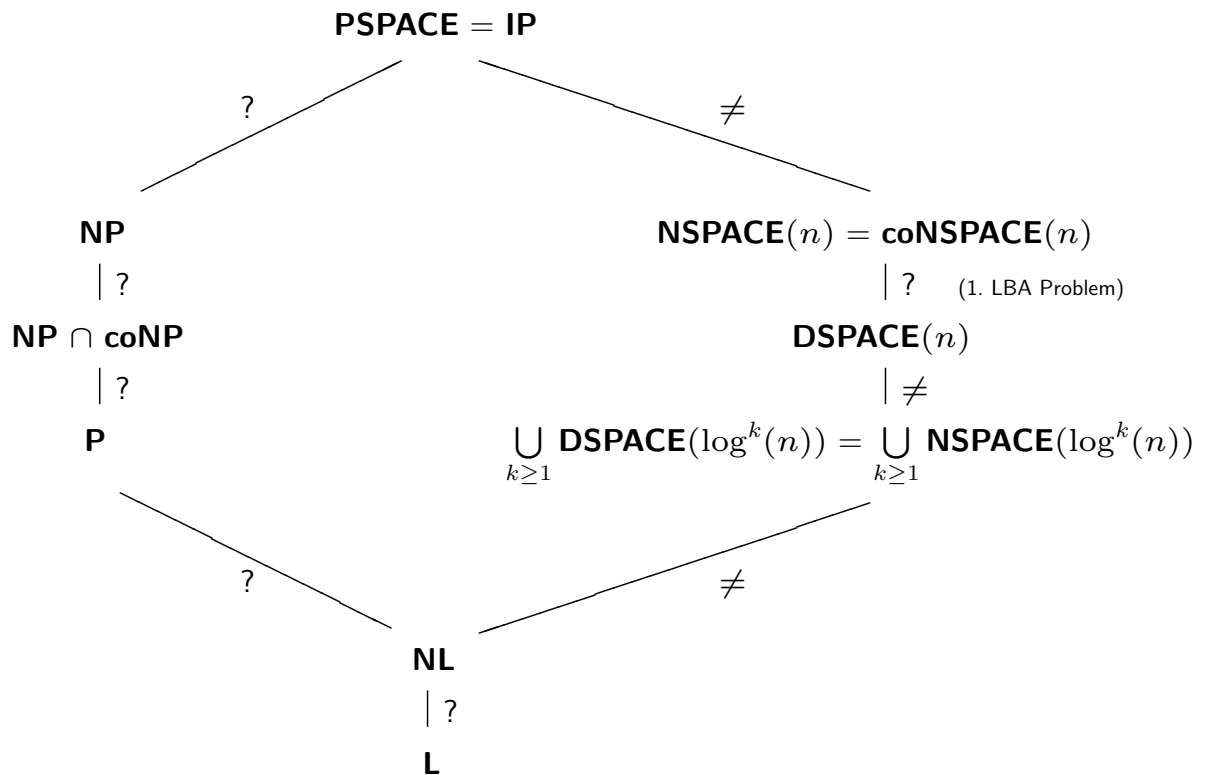
$$\mathbf{P} = \bigcup_{k \geq 1} \mathbf{DTIME}(n^k), \quad \mathbf{NP} = \bigcup_{k \geq 1} \mathbf{NTIME}(n^k),$$

$$\mathbf{PSPACE} = \bigcup_{k \geq 1} \mathbf{DSPACE}(n^k) = \bigcup_{k \geq 1} \mathbf{NSPACE}(n^k).$$

Wir werden bald sehen (Satz von Savitch), dass die angegebene Definition für **PSPACE** sinnvoll ist, d.h. dass beide Vereinigungen die gleiche Klasse ergeben. (Man sagt auch: **PSPACE** ist wohldefiniert.)

Ein Ziel dieses Kapitels wird sein, einige der im folgenden Diagramm angegebenen Inklusionsbeziehungen zu verstehen. Bei den im Diagramm mit einem Fragezeichen versehenen Inklusionen ist die Gleichheit offen.

Mit Ausnahme des Paares $\mathbf{NP} \cap \mathbf{coNP}$ vs. $\bigcup_{k \geq 1} \mathbf{DSPACE}(\log^k(n))$ weiß man, dass die Sprachklassen im linken Zweig von denen im rechten Zweig des Diagramms paarweise verschieden sind. Nach heutigem Kenntnisstand kann man weder $\mathbf{L} = \mathbf{NP}$ noch $\mathbf{P} = \mathbf{PSPACE}$ ausschließen; beides zugleich ist allerdings nicht möglich, da dann $\mathbf{L} = \mathbf{PSPACE}$ gelten würde, im Widerspruch zum Platzhierarchiesatz, den wir beweisen werden.



3.2 Varianten algorithmischer Probleme

Viele algorithmische Probleme treten in Varianten auf. Als typisches Beispiel sei zuerst das Problem des Handlungsreisenden erwähnt:

Beispiel: Ein Reisender will eine gegebene Anzahl von Städten besuchen, ohne dabei an einem Ort zweimal vorbeizukommen, und er will dabei den kürzesten Weg nehmen. Das Wegenetz kann als gerichteter Graph und die Wegstrecken als Gewichte auf den Kanten des Graphen aufgefasst werden. Die Knoten stellen die Städte dar.

Sei $G = (V, E, \gamma : E \rightarrow \mathbb{N})$ ein gerichteter Graph mit Knotenmenge $V = \{1, \dots, n\}$, Kantenmenge $E \subseteq V \times V$ und Kantengewichten $\gamma(e) > 0$ für alle $e \in E$.

Ein Rundweg W ist gegeben durch eine Folge $W = (x_0, \dots, x_n)$, $x_0 = x_n$, $x_i \neq x_j$ für $1 \leq i < j \leq n$ und $(x_{i-1}, x_i) \in E$ für $1 \leq i \leq n$.

Die Kosten $\gamma(W)$ des Rundwegs W sind durch die Summe der Kantengewichte gegeben:

$$\gamma(W) = \sum_{i=1}^n \gamma((x_{i-1}, x_i)).$$

Wir betrachten drei Varianten: Die Entscheidungs- und die Berechnungsvariante, sowie das Optimierungsproblem:

(A) Entscheidungsvariante: Eingabe: G und ein $k \geq 0$. Frage: Existiert ein Rundweg mit Kosten höchstens k ? Das heißt, existiert ein geschlossener Weg, der alle Knoten genau einmal besucht und dessen Gesamtkosten nicht größer sind als k ?

(B) Berechnungsvariante: Eingabe: G und ein $k \geq 0$. Aufgabe: Falls ein Rundweg W mit $\gamma(W) \leq k$ existiert, berechne einen solchen.

(C) Optimierungsproblem: Eingabe: G . Aufgabe: Berechne einen kostenoptimalen Rundweg (falls überhaupt ein Rundweg existiert).

In der Praxis interessiert man sich häufig nur für eine Lösung der Optimierungsvariante (C). Es lässt sich aber leicht zeigen, dass eine effiziente Lösung des Entscheidungsproblems zu einer Lösung des Optimierungsproblems führt. Ist (A) in Polynomialzeit lösbar, dann ist es auch (C):

Zunächst setzen wir $k_{\max} = \sum_{e \in E} \gamma(e)$. Dieser Wert ist in polynomiell beschränkter Zeit berechenbar, und wenn überhaupt ein Rundweg existiert, dann existiert auch ein solcher mit $\gamma(W) \leq k_{\max}$.

Eine erste Anfrage an die Entscheidungsvariante liefert damit die Existenz eines Rundwegs, die wir hiermit annehmen können. Sei $k_{\text{opt}} = \gamma(W_{\text{opt}})$ die Kosten eines kostenoptimalen Rundwegs. Mit binärer Suche und Anfragen an das Entscheidungsproblem können wir k_{opt} offenbar in polynomialer Zeit bestimmen. (Das binäre Suchverfahren ist notwendig, da der Betrag von k_{opt} exponentiell groß, gemessen in der Eingabelänge, sein kann.)

Im nächsten Schritt berechnen wir nun eine optimale Lösung. (Bisher haben wir ja nur deren Kosten!). Das folgende Programm ermittelt einen optimalen Rundweg.

Algorithmus *Optimaler Rundweg*

```
function rundweg( $G$ : Graph mit Kantenmenge  $E = \{e_1, \dots, e_m\}$ )  
begin  
   $R := E$ ;  
  for  $i := 1$  to  $m$  do  
    if es ex. Rundweg  $W$  mit  $\gamma(W) \leq k_{\text{opt}}$  und  $W$  benutzt nur Kanten aus  $R \setminus e_i$  then  
       $R := R \setminus e_i$ ;  
    endif  
  endfor;  
return  $R$   
endfunction
```

Am Ende der **for**-Schleife besteht R genau aus den Kanten eines kostenoptimalen Rundwegs. Um das zu beweisen, zeigt man induktiv die folgende Schleifeninvariante:

Sei R_i die Kantenmenge R nach dem i -ten Schleifendurchlauf, dann enthält R_i einen kostenoptimalen Rundweg und jeder kostenoptimale Rundweg mit Kanten aus R_i benutzt alle Kanten aus $R_i \cap \{e_1, \dots, e_i\}$.

Mit dieser Invariante folgt, dass am Ende R einen kostenoptimalen Rundweg enthält, der alle Kanten von R benutzt, d.h. R ist der gesuchte optimale Weg. Zur Zeitabschätzung beachten wir, dass m mal die Entscheidungsvariante benutzt wird, d.h. wenn (A) in Polynomialzeit lösbar ist, dann ist es auch (C).

Beispiel: Wir betrachten jetzt das Problem einer Knotenüberdeckung (engl.: *vertex cover*) in einem ungerichteten Graphen:

Definition: Eine Knotenüberdeckung eines Graphen $G = (V, E)$ ist eine Teilmenge $U \subseteq V$ der Knotenmenge mit der Eigenschaft, dass jede Kante $e \in E$ mindestens einen Endpunkt in U hat.

Das Problem **Vertex Cover** (abgekürzt **VC**) besteht in der Entscheidungsvariante (A) darin, auf Eingabe eines Graphen G und einer natürlichen Zahl k zu entscheiden, ob es eine Knotenüberdeckung U für G mit $|U| \leq k$ gibt.

In der Berechnungsvariante (B) soll auf die gleiche Eingabe ein solches U berechnet werden, falls eines existiert. In der Optimierungsvariante (C) soll auf Eingabe des Graphen G eine optimale Knotenüberdeckung ermittelt werden, d.h. eine Knotenüberdeckung U mit der Eigenschaft, dass $|U| \leq |U'|$ gilt für alle Knotenüberdeckungen U' .

Wie beim Problem des Handelsreisenden nimmt auch hier die Schwierigkeit von (A) bis (C) zu, aber auch hier reicht ein effizienter Algorithmus für (A) aus, um auch (C) effizient zu lösen:

Angenommen, das Entscheidungsproblem (A) zur Knotenüberdeckung lässt sich in Polynomialzeit entscheiden. Dann lässt sich eine Knotenüberdeckung U minimaler Größe in Polynomialzeit wie folgt konstruieren:

Sei $G = (V, E)$ der Eingabegraph und $|V| = n$. Indem man für alle i aus $\{1, \dots, n\}$ das Entscheidungsproblem (G, i) löst, kann man das minimale k bestimmen, für das eine Knotenüberdeckung der Größe k existiert. Wir nennen dieses k im weiteren k_{opt} .

Nun müssen wir nur noch die Berechnungsvariante (B) lösen, um eine optimale Knotenüberdeckung zu finden. Hierfür durchlaufen wir die Liste aller Knoten und entfernen jeweils den betrachteten Knoten mit allen ausgehenden Kanten, falls in dem resultierenden Graph eine Knotenüberdeckung mit $k - 1$ Knoten existiert. Dabei setzen wir k auf $k - 1$. (Wenn es keine solche Überdeckung im resultierenden Graph gibt, lassen wir den Graph und auch den Wert k unverändert.) Am Ende dieses Durchlaufs sind alle Kanten aus dem Graph entfernt. Insgesamt wurden genau k_{opt} viele Knoten entfernt, die somit eine optimale Knotenüberdeckung bilden.

Tatsächlich ist das Vertex Cover Problem **NP**-vollständig; es ist daher nicht sehr wahrscheinlich, dass es einen effizienten Algorithmus für dieses Problem (egal in welcher Variante) gibt.

3.3 Grapherreichbarkeit

Ein wichtiges algorithmisches Problem ist die Frage nach der Existenz von Wegen in gerichteten Graphen. Sei $G = (V, E)$ mit $E \subseteq V \times V$ ein gerichteter Graph und seien s, t zwei Knoten aus V . Die Frage, ob es einen gerichteten Pfad von s nach t gibt, ist leicht in Polynomialzeit zu lösen. Die entsprechenden Algorithmen erfordern jedoch linearen Platz. Nichtdeterministisch lässt sich dieses Problem aber in logarithmischem Platz lösen. Also gehört das Problem zur Komplexitätsklasse $\mathbf{NL} = \mathbf{NSPACE}(O(\log n))$. Das wollen wir nun nachweisen.

Sei $|V| = n$ und ohne Einschränkung seien die Knoten durch die Zahlen $1, \dots, n$ gegeben. Wir können darüberhinaus $s = 1$ und $t = n$ annehmen. (Der Fall $s = t$ ist trivial.)

Die Lösung geben wir durch das folgende nichtdeterministische Programm an, wobei die Knotenvariablen u und v als Binärzahlen auf dem Arbeitsband gehalten werden.

Das Programm kann nur dann terminieren, wenn es einen Pfad von s nach t gibt. Andererseits: Wenn es einen Pfad von s nach t gibt, so gibt es auch einen terminierenden Programmablauf. Damit ist gezeigt, dass das Programm das Problem korrekt entscheidet. Da der einzige benutzte Speicherplatz für die Variablen u und v gebraucht wird, ist der Platzbedarf in $O(\log n)$.

Algorithmus Grapherreichbarkeit

```
function wegsuche( $G$ : Graph mit Knotenmenge  $V$ );  
var  $u, v$ : Knoten;  
begin  
  while  $u \neq n$  do  
    wähle nichtdeterministisch einen Knoten  $v \in V$  mit  $(u, v) \in E$ ;  
     $u := v$ ;  
  endwhile ;  
return Weg von  $s$  nach  $t$  existiert!  
endfunction
```

Mit Hilfe des Ansatzes von Savitch (vgl. Satz von Savitch, Abschnitt 3.5) lässt sich dieses nichtdeterministische Programm deterministisch so simulieren, dass der benötigte Platz nur auf $O(\log^2(n))$ ansteigt, d.h. Grapherreichbarkeit liegt in $\mathbf{DSpace}(O(\log^2 n))$. Die Grundidee hierbei ist die folgende: Wenn es einen Weg von s nach t gibt, dann gibt es auch einen schleifenfreien solchen Weg. Dieser hat Länge kleiner als n (Anzahl durchlaufener Kanten). Wir definieren daher ein Prädikat $\text{Reach}(u, v, i)$ für $u, v \in V$ und $i \in \mathbb{N}$ mit $0 \leq i \leq \lceil \log_2 n \rceil$ wie folgt:

$$\text{Reach}(u, v, i) \iff \exists \text{ Pfad von } u \text{ nach } v \text{ der Länge } \leq 2^i.$$

Für $i = 0$ lässt sich $\text{Reach}(u, v, 0)$ durch einfache Inspektion der Kantenmenge in logarithmischem Platz entscheiden. ($\text{Reach}(u, v, 0)$ ist genau dann wahr, wenn es eine Kante von u nach v gibt.)

Für größere i gilt die folgende Rekursion:

$$\text{Reach}(u, v, i) \iff \exists w \in V : \text{Reach}(u, w, i - 1) \text{ und } \text{Reach}(w, v, i - 1)$$

Denn für $i \geq 1$ gibt es genau dann einen Pfad von u nach v der Länge $\leq 2^i$, wenn es einen Zwischenknoten $w \in V$ und Pfade von u nach w und von w nach v der Länge $\leq 2^{i-1}$ gibt.

Wir können $\text{Reach}(u, v, i)$ also durch das nachfolgende deterministische Programm berechnen:

Algorithmus *Reach* - deterministisch

```
function Reach( $u, v, i$ : integer);  
var  $b$ : boolean;  
begin  
   $b := \text{false}$ ;  
  if  $i = 0$  then  $b := (u = v)$  or  $(u, v) \in E$   
  else for  $w := 1$  to  $n$  do  
    if not  $b$  then  
      if Reach( $u, w, i - 1$ ) then  
         $b := \text{Reach}(w, v, i - 1)$ ;  
      endif  
    endif  
  endfor  
endif  
  return  $b$   
endfunction
```

Man kann leicht per Induktion nachweisen, dass ein Aufruf $\text{Reach}(u, v, i)$ mit Platzaufwand $O((i + 1) \log n)$ auskommt. Also kann man $\text{Reach}(u, v, \lceil \log_2(n) \rceil)$ in **DSPACE** $(O(\log^2(n)))$ berechnen.

Als Übungsbeispiel ermittle man den Zeitbedarf der rekursiven Berechnung von $\text{Reach}(u, v, \lceil \log_2(n) \rceil)$ für den Fall, dass der Eingabegraph ein kantenloser Graph ist, d.h. dass der Rückgabewert von $\text{Reach}(u, v, i)$ für $i > 0$ immer **false** ist. (Zeigen Sie dazu, dass die Berechnung von $\text{Reach}(u, v, i)$ Zeit $\Theta(n^i)$ benötigt.)

Bemerkung: Der oben angegebene nichtdeterministische Algorithmus für das Grapherreichbarkeitsproblem zeigt, dass dieses zur Klasse **NL** gehört. Tatsächlich ist es **NL**-vollständig (unter der in diesem Bereich üblichen logspace-Reduktion).

3.4 Grundlegende Beziehungen

Die folgenden Beziehungen geben wir ohne genaue Beweise an. Die Beweise kann man in Standardlehrbüchern über Theoretische Informatik bzw. über Komplexitätstheorie finden (z.B. Reischuk: Einf. in die Komplexitätstheorie oder Wagner/Wechsung: Complexity Theory).

Satz: (Bandreduktion/Bandkompression/Zeitreduktion)

1. Für $X = D$ oder $X = N$ gilt $\mathbf{XSPACE}(O(f)) = \mathbf{XSPACE}_{1\text{-Band}}(f)$.
2. $\mathbf{NTIME}(O(f)) = \mathbf{NTIME}(f)$.
3. Sei $\varepsilon > 0$ und $\forall n : f(n) \geq (1 + \varepsilon)n$, dann gilt $\mathbf{DTIME}(O(f)) = \mathbf{DTIME}(f)$.
4. $\mathbf{DTIME}(n) \neq \mathbf{DTIME}(O(n))$.

Die erste Aussage des Satzes ist eine Kombination aus Bandreduktionssatz und Bandkompressionssatz, die nächsten beiden Aussagen nennt man Zeitreduktion. Der letzte Punkt sagt aus, dass im deterministischen Fall Linearzeit nicht auf Realzeit komprimierbar ist.

Auch für Zeitklassen gibt es einen Bandreduktionssatz. Mit einem naiven Ansatz kann man leicht die Inklusion $\mathbf{DTIME}(f) \subseteq \mathbf{DTIME}_{1\text{-Band}}(f^2)$ zeigen. Der Satz von Hennie und Stearns benötigt zwar zwei Bänder, ist dabei aber wesentlich effizienter im Zeitverbrauch:

Satz: (Hennie-Stearns 1966) Sei $\varepsilon > 0$, $k \geq 1$ und $\forall n : f(n) \geq (1 + \varepsilon)n$, dann gilt:

$$\mathbf{DTIME}_{k\text{-Band}}(f) \subseteq \mathbf{DTIME}_{2\text{-Band}}(f \cdot \log(f)).$$

Wir wollen nun die Klassen von der Form **DTIME**, **NTIME**, **DSPACE** und **NSPACE** miteinander in Beziehung setzen. Zuerst zeigen wir:

Satz: Es gelte $f(n) \geq n$ für alle n . Dann folgt:

$$\mathbf{DTIME}(f) \subseteq \mathbf{NTIME}(f) \subseteq \mathbf{DSPACE}(f).$$

Beweis: Zu zeigen ist nur $\mathbf{NTIME}(f) \subseteq \mathbf{DSPACE}(f)$.

Sei also M eine nichtdeterministische $f(n)$ -zeitbeschränkte Turingmaschine. Auf Eingabe $x \in \Sigma^*$ mit $|x| = n$ können wir uns die Rechnung von M als einen Berechnungsbaum vorstellen. Dabei besteht die Wurzel aus der Startkonfiguration, und die Kinder jedes inneren Knotens, der die Konfiguration c darstellt, sind gerade die möglichen Nachfolgekonfigurationen zu c .

Dieser Baum wird in einem Breitendurchlauf auf eine akzeptierende Konfiguration hin durchsucht. Dabei merkt man sich jeweils nur eine Konfiguration und das *Protokoll*, mit dem diese Konfiguration erreicht wird (d.h. den von der Wurzel dorthin führenden Pfad im Baum). Die Tiefe dieses Baums ist beschränkt durch $f(n)$, d.h. der benötigte Speicherplatz ist maximal $f(n)$ für das Protokoll plus $O(f(n))$ für die gespeicherte Konfiguration; insgesamt also $O(f(n))$. Wegen $\mathbf{DSPACE}(O(f)) = \mathbf{DSPACE}(f)$ erhalten wir das gewünschte Ergebnis. \square

Satz: Es gelte $f(n) \geq \log(n)$ für alle n . Dann folgt:

$$\mathbf{DSPACE}(f) \subseteq \mathbf{NSPACE}(f) \subseteq \mathbf{DTIME}(2^{O(f)}).$$

Beweis: Zu zeigen ist nur $\mathbf{NSPACE}(f) \subseteq \mathbf{DTIME}(2^{O(f)})$.

Sei M eine $f(n)$ -platzbeschränkte nichtdeterministische Turingmaschine und $x \in \Sigma^*$ mit $|x| = n$ eine Eingabe. Es existiert eine nur von M abhängende Konstante c , für die die Anzahl der von der Startkonfiguration $\text{Start}(x)$ aus erreichbaren Konfigurationen durch $c^{f(n)}$ beschränkt ist. (Man beachte, dass $c^f \in 2^{O(f)}$ gilt.)

Nun erstellt man eine Liste \mathcal{L} aller von $\text{Start}(x)$ aus erreichbaren Konfigurationen nach folgendem Schema:

Anfangs enthält \mathcal{L} nur die Startkonfiguration $\text{Start}(x)$. Solange \mathcal{L} noch eine Konfiguration α enthält, von der noch nicht alle direkten Nachfolgekonfigurationen in der Liste enthalten sind, wird \mathcal{L} um diese direkten Nachfolger erweitert. Die Anzahl der Konfigurationen in \mathcal{L} ist durch $c^{f(n)}$ beschränkt. In jedem Durchlauf werden also maximal $c^{f(n)}$ Konfigurationen α mit maximaler Länge $f(n)$ betrachtet. Das Durchsuchen, ob alle direkten Nachfolger bereits vorhanden sind, kostet pro Konfiguration höchstens die Zeit $O(f(n) \cdot c^{f(n)})$. In jedem Durchlauf wird die Liste entweder vergrößert oder das Verfahren terminiert. Die Gesamtzeit ist also durch $O(f(n) \cdot c^{2f(n)}) \subseteq O(2^{O(f)})$ beschränkt.

Stößt man während dieser Listenerstellung auf eine erreichbare akzeptierende Konfiguration, bricht man ab und akzeptiert. Stößt man dagegen bis zum Abbruch nie auf eine akzeptierende Konfiguration, wird die Eingabe abgelehnt. Damit akzeptiert die deterministische Simulation genau dieselben Eingaben wie die nichtdeterministische Maschine M . \square

Bemerkung: *Es gelten die folgenden Beziehungen:*

- $L \subseteq NL \subseteq \text{DSPACE}(\log^2(n)) \subseteq \text{DTIME}(n^{O(\log(n))})$

- $NL \subseteq P$

- $CS = LBA = \text{NSPACE}(n) \subseteq \text{DTIME}(2^{O(n)})$

(Hierbei bezeichnet CS die Klasse der kontextsensitiven Sprachen, und LBA die Klasse der durch linear beschränkte Automaten akzeptierten Sprachen.)

- $\text{DSPACE}(n^2) \subseteq \text{DTIME}(2^{O(n^2)})$

Einschub: Starke Komplexitätsklassen, Zeitkonstruierbarkeit, Platzkonstruierbarkeit

Die drei genannten Begriffe spielen in unseren weiteren Betrachtungen keine besonders wichtige Rolle. Aber man sollte eine Vorstellung davon haben, was es damit auf sich hat:

- Als starke (nichtdeterministische) Komplexitätsklassen bezeichnet man Klassen, bei denen die Einhaltung der entsprechenden Schranken (Zeit- oder Platzschranken) bei jeder Berechnung obligatorisch ist - nicht nur bei akzeptierenden Berechnungen!
- Eine Funktion $f : \mathbb{N} \rightarrow \mathbb{N}$ heißt *zeitkonstruierbar*, falls es eine deterministische Turingmaschine gibt, die auf Eingabe 1^n (d.h. n in unärer Kodierung) nach genau $f(n)$ Schritten anhält.

Beispielsweise sind alle Polynome zeitkonstruierbar. (Beweis: Übung)

- Eine Funktion $f : \mathbb{N} \rightarrow \mathbb{N}$ heißt *platzkonstruierbar*, falls es eine deterministische Turingmaschine gibt, die auf Eingabe 1^n genau $f(n)$ Felder auf den Arbeitsbändern markiert und dann anhält. Darüberhinaus darf sie den markierten Platz bei der Berechnung nicht verlassen.

Beispielsweise sind alle zeitkonstruierbaren Funktionen auch platzkonstruierbar, aber auch solche Funktionen wie $\lceil \log^k(n) \rceil$ oder \sqrt{n} . (Beweis: Übung)

Näheres findet man in der Literatur über Komplexitätstheorie.

3.5 Der Satz von Savitch

Der Satz von Savitch (1970) besagt, dass eine nichtdeterministische platzbeschränkte Turingmaschine unter quadratischem Mehraufwand deterministisch simuliert werden kann. Diese platzeffiziente Simulation wird durch einen extremen Mehraufwand an Rechenzeit realisiert. Der Satz gilt für die hier verwendeten starken Platzschranken, ohne dass s als (platz-) konstruierbar vorausgesetzt werden muss.

Satz: (Savitch, 1970) Es sei $s \in \Omega(\log(n))$. Dann gilt

$$\mathbf{NSPACE}(s) \subseteq \mathbf{DSPACE}(s^2).$$

Beweis: Wir skizzieren den Beweis unter der Annahme, dass s platzkonstruierbar ist. Wie man sich von dieser Annahme befreit, kann der interessierte Leser selbst erarbeiten.

Der Beweis verläuft analog zur Lösung des Grapherreichbarkeitsproblems. Im hier betrachteten Graphen sind die Knoten die Konfigurationen bis zu einer bestimmten Länge m . Die Kanten sind durch die direkte Nachfolgerrelation gegeben. Die Frage ist, ob eine akzeptierende Konfiguration von der Startkonfiguration aus erreichbar ist oder nicht.

Sei M eine s -platzbeschränkte nichtdeterministische Turingmaschine, die die Sprache L akzeptiert.

Mit $\alpha, \beta, \gamma \dots$ bezeichnen wir Konfigurationen, und $\text{Reach}(\alpha, \beta, i)$ soll bedeuten, dass sich (bei Eingabe $w \in \Sigma^*$) die Konfiguration α in maximal 2^i Schritten in die Konfiguration β überführen lässt. Wir können annehmen, dass M nur eine einzige akzeptierende Konfiguration (Accept) hat. $\text{Start}(w)$ sei die Startkonfiguration (bei der gegebenen Eingabe). Aus der Beschreibung von M kann man explizit eine Konstante c berechnen, so dass für alle $w \in \Sigma^*$ gilt:

$$w \in L \iff \text{Reach}(\text{Start}(w), \text{Accept}, c \cdot s(|w|))$$

Das Prädikat $\text{Reach}(\alpha, \beta, i)$ lässt sich für $i = 0$ direkt berechnen, und für $i > 0$ verwenden wir das Rekursionsschema:

$$\text{Reach}(\alpha, \beta, i) \iff \exists \text{ Konfiguration } \gamma \text{ mit } |\gamma| \leq s(|w|): \\ \text{Reach}(\alpha, \gamma, i - 1) \text{ und } \text{Reach}(\gamma, \beta, i - 1).$$

Dieses Schema kann offensichtlich in ein deterministisches Programm übersetzt werden, wenn $s(|w|)$ bekannt ist. Das ist in jedem Fall so, wenn s eine platzkonstruierbare Funktion ist.

Für eine geeignete Konstante c' kann man per Induktion über i zeigen, dass der Platzbedarf zur Berechnung von $\text{Reach}(\alpha, \beta, i)$ durch $c' \cdot s(|w|) \cdot (i + 1)$ abgeschätzt werden kann. Daraus ergibt sich der maximale Platzbedarf von $O(s^2)$. \square

Um sich von der Forderung der Platzkonstruierbarkeit für s zu befreien, kann man mit demselben Ansatz zeigen, dass sich der tatsächliche Platzbedarf einer s -platzbeschränkten Turingmaschine in $\mathbf{DSPACE}(s^2)$ berechnen lässt.

Nun erhalten wir auch die Wohldefiniiertheit von **PSPACE**:

Korollar: *Es gilt:*

$$\bigcup_{k \geq 1} \mathbf{DSPACE}(n^k) = \bigcup_{k \geq 1} \mathbf{NSPACE}(n^k).$$

Als Beweis genügt es zu zeigen, dass für alle j gilt: $\mathbf{NSPACE}(n^j) \subseteq \bigcup_{k \geq 1} \mathbf{DSPACE}(n^k)$. Aber das ist nach dem Satz von Savitch korrekt, da ja $\mathbf{NSPACE}(n^j)$ eine Teilklasse von $\mathbf{DSPACE}(n^{2j})$ ist.

3.6 Hierarchiesätze

Der Nachweis sogenannter unterer Schranken gestaltet sich im Allgemeinen sehr schwierig. Die generelle Idee ist, dass die Bereitstellung größerer Ressourcen an Platz und/oder Zeit zu größeren Klassen führen soll. Um jedoch zu beweisbaren Hierarchieresultaten zu gelangen, müssen mehrere Voraussetzungen erfüllt sein. Wir benötigen die Konstruierbarkeit der Platz- bzw. Zeitschranken, sowie die Abgeschlossenheit der betrachteten Klassen unter Komplement. Außerdem muss eine Mehrbandmaschine auf einer universellen Turingmaschine (mit einer festen Zahl von Arbeitsbändern) simulierbar sein.

Für deterministische Zeitklassen ist der Komplementabschluss trivialerweise immer gegeben. Auch für die deterministischen Platzklassen $\mathbf{DSPACE}(f)$ mit $f \in \Omega(\log n)$ wurde die Komplementabgeschlossenheit gezeigt (vgl. z.B. das Lehrbuch von Hopcroft und Ullman). Die Simulation beliebig vieler Bänder mit einem Band ist ohne Erhöhung des Platzbedarfs möglich (Bandreduktionssatz). Also erhalten wir den deterministischen Platzhierarchiesatz:

Satz: (*Platzhierarchiesatz*)

Seien $s_1, s_2 : \mathbb{N} \rightarrow \mathbb{N}$ Funktionen, $s_1 \notin \Omega(s_2)$, $s_2 \in \Omega(\log n)$ und sei s_2 platzkonstruierbar. Dann gilt

$$\mathbf{DSPACE}(s_2) \setminus \mathbf{DSPACE}(s_1) \neq \emptyset.$$

Diesen Satz werden wir gleich beweisen. Zuvor noch:

Bemerkung:

- Zur Erinnerung: Die Bedingung $s_1 \notin \Omega(s_2)$ bedeutet, dass für alle $\varepsilon > 0$ unendlich viele n existieren mit $s_1(n) < \varepsilon \cdot s_2(n)$.

Seien etwa $s_1(n) = n$ und $s_2(2n) = n^2$, $s_2(2n + 1) = \log n$ für alle n . Dann gilt sowohl $s_1 \notin \Omega(s_2)$ als auch umgekehrt $s_2 \notin \Omega(s_1)$. Also sind nach dem Platzhierarchiesatz $\mathbf{DSPACE}(s_1)$ und $\mathbf{DSPACE}(s_2)$ unvergleichbar.

- Aus dem Satz folgt unter anderem auch:

$$\begin{aligned} & \mathbf{DSPACE}(\log n) \subsetneq \mathbf{DSPACE}(\log^2 n) \subsetneq \mathbf{DSPACE}(n) \\ & \subseteq \mathbf{NSPACE}(n) \subsetneq \mathbf{DSPACE}(n^{2.1}) \subsetneq \mathbf{PSPACE} \end{aligned}$$

Beweis: (des Platzhierarchiesatzes:)

Wir wählen zuerst eine geeignete Kodierung deterministischer Turingmaschinen, die eine effiziente Simulation erlaubt. Jedes Wort $x \in \Sigma^*$ soll als Kodierung einer Turingmaschine M_x interpretiert werden können. Dazu kann man für nicht wohlgeformtes x zum Beispiel immer dieselbe Default-Maschine wählen.

Wichtig ist folgende Konvention: Wenn x die Turingmaschine M_x kodiert, dann soll für jedes $n \in \mathbb{N}$ auch $y = x|n$ eine Kodierung derselben Turingmaschine sein, d.h. $M_x = M_y$. Als Folgerung erhält man, dass jede Turingmaschine Kodierungen für alle bis auf endlich viele Längen besitzt.

Nun werden wir eine deterministische s_2 -platzbeschränkte Turingmaschine M so konstruieren, dass $\overline{L(M)} \notin \mathbf{DSPACE}(s_1)$ gilt. Damit folgt (wegen der Abgeschlossenheit von $\mathbf{DSPACE}(s_2)$), dass $L(M)$ in $\mathbf{DSPACE}(s_2) \setminus \mathbf{DSPACE}(s_1)$ liegt, und daraus die Behauptung.

Nun also zur Konstruktion von M : Auf Eingaben der Länge n markiert M zunächst den Platz $s_2(n)$ auf seinem Arbeitsband. Wenn M im folgenden den markierten Platz verlassen möchte, stoppt die Maschine und lehnt ab. Damit ist M in jedem Fall s_2 -platzbeschränkt.

Nach der Markierung des Bandes arbeitet M wie eine universelle Turingmaschine, und zwar so, dass bei Eingabe y die Maschine M_y auf Eingabe y simuliert wird. Wenn M ohne das oben beschriebene Verlassen des markierten Bereichs zur Akzeptierung (von M_y) kommt, dann akzeptiert auch M .

Die Simulation verläuft so: Zunächst kopiert M den wesentlichen Teil von y (d.h. das kürzeste x , für das y in der Form $y = x|n$ geschrieben werden kann) auf eine Spur ihres Arbeitsbandes. Auf einer anderen Spur merkt sich M die aktuelle Konfiguration von M_y mit Ausnahme der Kopfposition, die direkt auf dem Eingabeband simuliert werden kann. Um sich eine Konfiguration von M_y der Länge l zu merken, braucht M also $k \cdot (l + 1)$ Platz, wobei die Zahl $k \in \mathbb{N}$ nur von x bzw. der Maschine $M_x = M_y$ abhängt.

Nun müssen wir noch zeigen, dass $\overline{L(M)} \notin \mathbf{DSPACE}(s_1)$ gilt. Wir tun das mit einem Widerspruchsbeweis: Angenommen, \overline{M} wäre eine s_1 -platzbeschränkte deterministische 1-Band Turingmaschine, die $\overline{L(M)}$ akzeptiert. Sei x die kürzeste Kodierung von \overline{M} . Die Simulation von \overline{M}

auf Eingaben der Länge n benötigt daher maximal $k \cdot (s_1(n) + 1)$ Platz für ein nur von x abhängendes k . Wähle nun $n \geq |x|$ so, dass $k \cdot (s_1(n) + 1) \leq s_2(n)$ gilt. Das ist möglich, da $s_1 \notin \Omega(s_2)$ und $s_2 \notin O(1)$ gelten. Eine Simulation von \overline{M} auf Eingaben der Länge n , z.B. auf $x|^{n-|x|}$, verlässt den markierten Speicherbereich also nicht. Und damit gilt: M akzeptiert die Eingabe y genau dann, wenn $M_y = \overline{M}$ die Eingabe y akzeptiert. Wegen $L(\overline{M}) = \overline{L(M)}$ ist dies ein Widerspruch. \square

Bemerkung: Sipser hat 1980 die Komplementabgeschlossenheit aller deterministischen Platzklassen gezeigt. Das bedeutet, dass man im Platzhierarchiesatz die Bedingung $s_2 \in \Omega(\log n)$ ersetzen kann durch $s_2 \notin O(1)$.

Mit der Technik von Hennie und Stearns kann man eine Mehrband-Turingmaschine mit nur einem logarithmischen Faktor Zeitverlust auf einer Maschine mit zwei Bändern simulieren. Da **DTIME**(f) auch unter Komplementbildung abgeschlossen ist, ergibt sich analog zum eben bewiesenen Satz der deterministische Zeithierarchiesatz, den wir hier ohne Beweis angeben:

Satz: (Deterministischer Zeithierarchiesatz)

Es seien $t_1, t_2 : \mathbb{N} \rightarrow \mathbb{N}$ Funktionen, $t_1 \cdot \log(t_1) \notin \Omega(t_2)$, $t_2 \in \Omega(n \log n)$ und t_2 zeitkonstruierbar. Dann gilt

$$\mathbf{DTIME}(t_2) \setminus \mathbf{DTIME}(t_1) \neq \emptyset.$$

Hieraus kann man folgern:

$$\begin{aligned} \mathbf{DTIME}(O(n)) &\subsetneq \mathbf{DTIME}(O(n^2)) \subsetneq \mathbf{P} \\ &\subsetneq \mathbf{DTIME}(O(2^n)) \subsetneq \mathbf{DTIME}(O((2 + \varepsilon)^n)) \end{aligned}$$

Die in den Hierarchiesätzen vorkommende Konstruierbarkeitsvoraussetzung ist essentiell, wie man aus dem nachfolgenden *Lückensatz* von Borodin schließen kann:

Satz: (Lückensatz von Borodin, 1972)

Es sei r eine totale, berechenbare Funktion, $r(n) \geq n$ für alle n . Dann gibt es effektiv eine totale, berechenbare Funktion $s : \mathbb{N} \rightarrow \mathbb{N}$ mit der Eigenschaft:

$$\mathbf{DTIME}(s) = \mathbf{DTIME}(r \circ s).$$

Beweis: Sei M_1, M_2, \dots eine Aufzählung aller Turingmaschinen, und sei $t_i(n)$ der tatsächliche maximale Zeitbedarf einer Rechnung von M_i auf Eingaben der Länge n . Es gilt $t_i(n) \in \mathbb{N} \cup \{\infty\}$.

Nun betrachten wir die Mengen $N_n = \{t_k(n) \mid 1 \leq k \leq n\} \subseteq \mathbb{N} \cup \{\infty\}$. Diese Mengen sind endlich, also existieren Zahlen s_n und Intervalle $[s_n, r(s_n)]$ (das sind die *Lücken*) mit $N_n \cap [s_n, r(s_n)] = \emptyset$. Für jedes n ist etwa $s_n = \max\{t_k(n) \mid 1 \leq k \leq n \wedge t_k(n) < \infty\}$ ein möglicher Wert – dieser ist jedoch im Allgemeinen zu groß und nicht berechenbar. Einen passenden berechenbaren Wert für s_n finden wir mit der folgenden Funktion:

Algorithmus zur Ermittlung von s_n

```
function  $s(n : \text{integer})$ ;  
 $s := 0$ ;  
repeat  
   $s := r(s) + 1$   
until  $\forall k \leq n : (t_k(n) < s) \text{ or } (t_k(n) > r(s))$ ;  
return  $s$   
endfunction
```

(Die Funktion s wächst monoton; es kann aber sein, dass sie nicht platzkonstruierbar ist.)

Wir zeigen jetzt, dass $\mathbf{DTIME}(s) = \mathbf{DTIME}(r \circ s)$ gilt:

Da $r(n) \geq n$ für alle n gilt, ist die Inklusion von links nach rechts klar. Sei nun $L \in \mathbf{DTIME}(r \circ s)$; wir wollen zeigen, dass $L \in \mathbf{DTIME}(s)$ gilt.

Sei M_k eine $(r \circ s)$ -zeitbeschränkte deterministische Turingmaschine mit $L = L(M_k)$. Dann gilt für alle n : $t_k(n) \leq r(s(n))$. Es folgt für alle $n \geq k$ nach der konstruktiven Berechnungsvorschrift für s : $t_k(n) < s(n)$. Das heißt, für fast alle n gilt $t_k(n) \leq s(n)$ und daher ist L in $\mathbf{DTIME}(s)$. \square

Korollar: *Es gibt berechenbare Funktionen s , für die folgendes gilt:*

$$\mathbf{DTIME}(s) = \mathbf{NSPACE}(s) = \mathbf{DTIME}(s^{O(s)}) = \mathbf{NSPACE}(2^{2^{2^{2^s}}}).$$

Der Lückensatz ist auf den ersten Blick erstaunlich. Man könnte naiverweise annehmen, dass für jede Funktion s die Klasse $\mathbf{DTIME}(2^s)$ mächtiger sein müsste als $\mathbf{DTIME}(s)$.

3.7 Die Translationstechnik

Mit Hilfe des Translationsatzes lässt sich eine Enthaltenseinsbeziehung zwischen kleinen Komplexitätsklassen in eine Beziehung zwischen großen Klassen überführen. Die zugrundeliegende Idee besteht im sogenannten *Ausstopfen* (engl.: padding) einer Sprache.

Definition: Sei $L \subseteq \Sigma^*$ eine Sprache, $f : \mathbb{N} \rightarrow \mathbb{N}$ eine Funktion mit $f(n) \geq n$ für alle $n \geq 0$ und $\$ \notin \Sigma$ ein neues Symbol. Definiere eine Sprache $\text{Pad}_f(L)$ über dem erweiterten Alphabet $\Sigma \cup \{\$\}$ durch

$$\text{Pad}_f(L) = \{w\$^{f(|w|)-|w|} \mid w \in L\}.$$

Man beachte, dass auf diese Weise jedem Wort aus L mit Länge n ein Wort aus $w\* der Länge $f(n)$ zugeordnet wird.

Satz: Seien g und f zwei zeitkonstruierbare Funktionen, $f(n) \geq n$ und $g(n) \geq n$ für alle n . Dann gilt für alle Sprachen $L \subseteq \Sigma^*$:

1. $\text{Pad}_f(L) \in \mathbf{DTIME}(O(g)) \iff L \in \mathbf{DTIME}(O(g \circ f))$,
2. $\text{Pad}_f(L) \in \mathbf{NTIME}(O(g)) \iff L \in \mathbf{NTIME}(O(g \circ f))$.

Beweis: Der Beweis wird hier nur für **DTIME** geführt – für **NTIME** geht er ganz analog.

Für die Richtung von links nach rechts sei $\text{Pad}_f(L)$ in $\mathbf{DTIME}(O(g))$. Wir müssen zeigen, dass L in der Klasse $\mathbf{DTIME}(O(g \circ f))$ liegt: Auf Eingabe w aus Σ^* berechnen wir erst das Wort $w\$^{f(|w|)-|w|}$ in Zeit $f(|w|)$. (Das geht, da f zeitkonstruierbar ist.)

Jetzt teste in $\mathbf{DTIME}(g(f(|w|)))$, ob dieses Wort zu $\text{Pad}_f(L)$ gehört. Da das genau dann der Fall ist, wenn $w \in L$ liegt, haben wir gezeigt, dass L zu der gewünschten Klasse gehört.

Nun gehen wir für die andere Richtung davon aus, dass L in $\mathbf{DTIME}(O(g \circ f))$ liegt. Auf eine Eingabe $x \in (\Sigma \cup \{\$\})^*$ testen wir zunächst, ob x von der Form $w\k ist für ein $w \in \Sigma^*$ und ein $k \geq 0$. Wenn nicht, lehnen wir ab.

Nun setzen wir $n = |w|$, $m = |x|$ und berechnen $g(m)$ in Zeit $g(m)$. Das geht wieder wegen der Zeitkonstruierbarkeit. Für die weiteren Berechnungen lassen wir eine „Uhr“ mitlaufen und lehnen ab, falls die Zeit $c \cdot g(m)$ überschritten wird. (c ist dabei eine geeignet gewählte Konstante, die so gewählt ist, dass für Eingaben $x \in \text{Pad}_f(L)$ diese Zeit ausreicht, um zu akzeptieren.)

Jetzt testen wir, ob $m = f(n)$ gilt. Das geht in Zeit $f(n)$, da f zeitkonstruierbar ist. Dann prüfen wir, ob $w \in L$ liegt; hierfür brauchen wir Zeit $c \cdot g(f(n)) = c \cdot g(m)$. Also liegt $\text{Pad}_f(L)$ in $\mathbf{DTIME}(O(g))$. \square

Ein analoges Resultat (mit angepassten Voraussetzungen für f und g) gilt auch für Raumklassen, d.h. für **DSPACE** und **NSPACE**.

Satz: Sei $g \in \Omega(\log n)$ und sei $f(n) \geq n$ für alle n . Auf unäre Eingabe 1^n sei der binär codierte Wert von $f(n)$ deterministisch mit Platzbedarf $g(f(n))$ berechenbar. Dann gilt für alle Sprachen $L \subseteq \Sigma^*$:

1. $\text{Pad}_f(L) \in \mathbf{DSPACE}(g) \iff L \in \mathbf{DSPACE}(g \circ f),$
2. $\text{Pad}_f(L) \in \mathbf{NSPACE}(g) \iff L \in \mathbf{NSPACE}(g \circ f),$

(Ohne Beweis...)

Der Zusammenfall einer Hierarchie von Komplexitätsklassen ist daher am ehesten weit oben zu erwarten. Wollen wir dagegen Separationsresultate erzielen, so haben wir hierfür die besten Aussichten am unteren Ende einer Hierarchie.

Korollar:

$$\mathbf{DSPACE}(n) \neq \mathbf{NSPACE}(n) \implies \mathbf{L} \neq \mathbf{NL}.$$

Beweis: Wir nehmen $\mathbf{L} = \mathbf{NL}$ an. Dann gilt für jede Sprache $L \in \mathbf{NSPACE}(n)$ nach dem Translationssatz $\text{Pad}_{\text{exp}}(L) \in \mathbf{NSPACE}(\log n) = \mathbf{NL} = \mathbf{L} = \mathbf{DSPACE}(\log n)$. Nun wenden wir wieder den Translationssatz an und erhalten $L \in \mathbf{DSPACE}(n)$. \square

Mit der Translationstechnik lässt sich in einigen Fällen direkt die Verschiedenheit von Komplexitätsklassen nachweisen:

Korollar: *Es gilt*

$$\mathbf{P} \neq \mathbf{DSPACE}(n)$$

Beweis: Wir wählen $L \in \mathbf{DSPACE}(n^2) \setminus \mathbf{DSPACE}(n)$ (existiert nach Platzhierarchiesatz) und die Paddingfunktion $f(n) = n^2$. Dann gilt $\text{Pad}_f(L) \in \mathbf{DSPACE}(n)$. Wäre $\mathbf{P} = \mathbf{DSPACE}(n)$, dann wäre $\text{Pad}_f(L) \in \mathbf{P}$ und folglich $\text{Pad}_f(L) \in \mathbf{DTIME}(n^k)$ für ein $k \geq 1$. Also wäre nach dem Translationssatz $L \in \mathbf{DTIME}(n^{2k})$, also $L \in \mathbf{P} = \mathbf{DSPACE}(n)$, im Widerspruch zur Definition von L . \square

Bemerkung:

- Insbesondere gilt, dass \mathbf{P} nicht gleich der Sprachklasse der kontextsensitiven Sprachen ist.
- Sowohl $\mathbf{DSPACE}(\log n) = \mathbf{P}$, wie auch $\mathbf{DSPACE}(n) \subset \mathbf{P}$ oder $\mathbf{P} \subset \mathbf{DSPACE}(n)$ sind nach heutigem Wissensstand möglich.