8 Isomorphe, dünne und dichte Sprachen

In diesem Abschnitt legen wir die Vermutung nahe, dass alle NP-vollständigen Sprachen zueinander „isomorph“ sind. Dies würde ausschließen, dass es „dünne“ NP-vollständige Sprachen (mit poly(n) Wörtern einer Maximallänge n) gibt.

Die Theorie der isomorphen, dünnen und dichten Sprachen wurde 1977 von Berman und Hartmanis ins Leben gerufen. Da sie die „P versus NP“ Frage aus einer neuen Perspektive beleuchtet, präsentieren wir im Folgenden ihre zentralen Konzepte und Resultate.

Definition 8.1 *Zwei Sprachen $L_1, L_2 \in \Sigma^*$ heißen* polynomiell isomorph*, wenn eine bijektive Abbildung $h : \Sigma^* \to \Sigma^*$ mit Umkehrabbildung $h^{-1} : \Sigma^* \to \Sigma^*$ existiert, die folgende Bedingungen erfüllt:*

1. *Sowohl h als auch h^{-1} sind in Polynomialzeit berechenbar.*
2. *Für alle $x \in \Sigma^*$ gilt $x \in L_1$ gdw $h(x) \in L_2$.*

In Zeichen: $L_1 \overset{pol}{\simeq} L_2$.

Offensichtlich gilt

$$L_1 \overset{pol}{\simeq} L_2 \Rightarrow L_1 \overset{pol}{\sim} L_2 \ ,$$

d.h., zwei polynomiell isomorphe Sprachen sind erst recht polynomiell äquivalent.

Wie wir wissen sind alle NP-vollständigen Sprachen wechselseitig aufeinander polynomiell reduzierbar und somit polynomiell äquivalent. Die polynomiellen Reduktionen, die wir dabei üblicherweise benutzen, sind aber weit davon entfernt, Bijektionen zu sein. Insbesondere die Surjektivität ist selten erfüllt, da wir bei einer Reduktion eines Quellproblems auf ein Zielproblem in der Regel extrem spezielle Eingabeinstanzen des Zielproblems konstruieren. Eine der seltenen Ausnahmen bildet die Reduktion $(G, k) \mapsto (\bar{G}, k)$ von CLIQUE auf IS bzw. die Reduktion $(G, k) \mapsto (G, n-k)$ von IS auf VC, wobei $\bar{G}$ den Komplementärgraphen[1] zu G bezeichnet und n die Anzahl der Knoten in G.

Wir skizzieren im Folgenden eine Technik zur Verwandlung einer polynomiellen Reduktion in eine polynomielle Isomorphie. Diese Technik ist immer dann anwendbar, wenn die Zielsprache der polynomiellen Reduktion über eine sogenannte „Polsterfunktion (padding function“) verfügt:

Definition 8.2 *Eine Funktion $pad : \Sigma^* \times \Sigma^* \to \Sigma^*$ heißt* Polsterfunktion *für $L \subseteq \Sigma^*$, wenn die folgenden Bedingungen erfüllt sind:*

1. *Die Funktion pad ist in Polynomialzeit berechenbar.*
2. *Für alle $x, y \in \Sigma^*$ gilt $x \in L$ gdw $pad(x, y) \in L$.*
3. *Für alle $x, y \in \Sigma^*$ gilt $|pad(x, y)| > |x| + |y|$.*

[1] Zwei Knoten sind in $\bar{G}$ benachbart gdw sie in G nicht benachbart sind.

4. Aus $pad(x, y)$ kann y in Polynomialzeit extrahiert werden.

Eine Polsterfunktion für L ist also im Wesentlichen eine Längen-expandierende polynomielle Reduktion von L auf sich selbst, die (auf leicht dekodierbare Weise) einen zusätzlichen String y in den Eingabestring x hinein kodiert. Wir verwenden im Folgenden oBdA das Binäralphabet $\Sigma = \{0, 1\}$.

Beispiel 8.3 *Wir skizzieren den Entwurf einer Polsterfunktion für SAT. Es bezeichne $x \in \Sigma^*$ einen String zur natürlichen Kodierung[2] einer Eingabeinstanz von SAT, d.h., x repräsentiert eine Boolesche Formel in konjunktiver Normalform mit m Klauseln $C_0, \ldots, C_{m-1}$, in denen n Variable, sagen wir $v_1, \ldots, v_n$, verwendet werden. Für $p = |y|$ bezeichne dann $pad(x, y)$ den Kodierungsstring für die Klauseln $C_0, \ldots, C_{m-1}$ (alte Klauseln) sowie $C_m, \ldots, C_{m+p+1}$ (neue Klauseln). Die neuen Klauseln verwenden neue Variable $v_{n+1}, \ldots, v_{n+1+p}$ und haben die folgende Form:*

- *C_m bestehe nur aus dem Literal v_{n+1} und C_{m+1} sei ein Duplikat von C_m.*
- *Für $i = 1, \ldots, p$ bestehe C_{m+1+i} nur aus dem Literal v_{n+1+i} sofern $y_i = 1$ bzw. nur aus dem Literal $\bar{v}_{n+1+i}$ sofern $y_i = 0$.*

Die hierdurch beschriebene Funktion pad ist in Polynomialzeit berechenbar. Offensichtlich sind die Klauseln $C_0, \ldots, C_{m-1}$ erfüllbar gdw $C_0, \ldots, C_{m-1}, C_m, \ldots, C_{m+1+p}$ erfüllbar sind (da die neu hinzu gefügten Klauseln erstens erfüllbar sind und zweitens nur neue Variable verwenden). Die neue Instanz $pad(x, y)$ hat offensichtlich eine $|x|+|y|$ überschreitende Länge. Weiterhin lässt sich y aus $pad(x, y)$ leicht ablesen: durchmustere $pad(x, y)$ von rechts nach links bis zum ersten Auffinden von zwei identischen Klauseln (C_m und C_{m+1}); lies aus den davon rechts befindlichen Klauseln den Binärstring y (auf die offensichtliche Weise) ab. Somit sind alle an eine Polsterfunktion gerichteten Bedingungen erfüllt.

Polsterfunktionen lassen sich für NP-vollständige Sprachen in der Regel ähnlich leicht angeben wie für SAT. Den Entwurf von Polsterfunktionen für die Sprachen CLIQUE und PARTITION empfehlen wir als **Übung**. **Übg.**

Das folgende Resultat ergibt sich unmittelbar aus den Eigenschaften von Polsterfunktionen und Reduktionsabbildungen.

Lemma 8.4 *Es sei pad eine Polsterfunktion für L_2 und $L_1 \leq_{pol} L_2$ mit Reduktionsabbildung $R : \Sigma^* \to \Sigma^*$. Dann ist*

$$x \mapsto pad(R(x), x)$$

eine Längen-expandierende, injektive und in Polynomialzeit invertierbare Reduktionsabbildung.

[2]Hier und im Folgenden legen wir die Details einer „natürlichen Kodierung“ nicht fest. Es ist aber prinzipiell leicht (wenngleich mühselig), solche Kodierungen über einem Alphabet mit mindestens zwei Zeichen fsetzulegen.

Wechselseitige Längen-expandierende, injektive und in Polynomialzeit invertierbare Reduktionsabbildungen können in Isomorphismen transformiert werden:

Lemma 8.5 (Berman und Hartmanis, 1977) *Es seien L_1 und L_2 polynomiell äquivalente Sprachen, R eine Reduktionsabbildung für $L_1 \leq_{pol} L_2$ und S eine Reduktionsabbildung für $L_2 \leq_{pol} L_1$. Sowohl R als auch S seien Längen-expandierend, injektiv und in Polynomialzeit invertierbar. Dann sind L_1 und L_2 polynomiell isomorph.*

Beweis Da $R, S : \Sigma^* \to \Sigma^*$ Längen-expandierend, injektiv und in Polynomialzeit invertierbare Abbildungen sind, sind die Inversen $R^{-1}, S^{-1} : \Sigma^* \to \Sigma^*$ Längen-reduzierende, partiell definierte und in Polynomialzeit berechenbare Abbildungen. Der Beweis beruht auf dem Konzept der R- und S-Ketten. Eine *S-Kette* für $x \in \Sigma^*$ hat die Form

$$\cdots R^{-1}(S^{-1}(R^{-1}(S^{-1}(x)))) \ .$$

Dies ist so zu verstehen, dass wir die Abbildungen S^{-1} und R^{-1} solange alternierend anwenden wie das jeweilige Argument im Definitionsbereich liegt. Da R^{-1} und S^{-1} beide Längen-reduzierend sind, muss die Kette nach maximal $n = |x|$ Schritten abbrechen. Wir unterscheiden die folgenden beiden Fälle:

Fall 1 Die S-Kette für x stoppt mit undefiniertem R^{-1}.
In diesem Fall hat die S-Kette die Form $S^{-1}(R^{-1}(\cdots S^{-1}(x) \cdots))$.

Fall 2 Die S-Kette für x stoppt mit undefiniertem S^{-1}.
In diesem Fall hat die S-Kette die Form $R^{-1}(S^{-1}(\cdots S^{-1}(x) \cdots))$.

Beachte, dass der Grenzfall, dass bereits $S^{-1}(x)$ undefiniert ist, zur S-Kette x führt und einen Unterfall zu Fall 2 darstellt.
Eine *R-Kette* für $x \in \Sigma^*$ ist symmetrisch definiert mit vertauschten Rollen von R und S.
Wir definieren jetzt eine Abbildung $h : \Sigma^* \to \Sigma^*$, von der wir anschließend nachweisen, dass sie einen Isomorphismus von L_1 nach L_2 darstellt:

1. Falls die S-Kette für x mit undefiniertem R^{-1} stoppt, dann setzen wir $h(x) := S^{-1}(x)$.
2. Falls die S-Kette für x mit undefiniertem S^{-1} stoppt, dann setzen wir $h(x) := R(x)$.

Wir weisen zunächst die Injektivität von h nach, indem wir die Annahme es existierten $x, y \in \Sigma^*$ mit $x \neq y$ und $h(x) = h(y)$ zum Widerspruch führen. Wegen der Injektivität von R und S könnte die angenommene Situation höchstens dann eintreten, wenn die S-Ketten für x und y sich auf die beiden Fälle verteilen, sagen wir, die S-Kette für x stoppt mit undefiniertem R^{-1} und die S-Kette für y stoppt mit undefiniertem S^{-1}. Hieraus ergibt sich

$$S^{-1}(x) = h(x) = h(y) = R(y)$$

und somit

$$y = R^{-1}(S^{-1}(x)) \ .$$

Dann wäre aber die S-Kette für y ein Präfix der S-Kette für x und die Ketten müssten entweder beide mit undefiniertem R^{-1} oder mit undefiniertem S^{-1} stoppen (Widerspruch). Demnach muss h eine injektive Abbildung sein.
Als nächstes weisen wir die Surjektivität von h nach, indem wir für jedes $y \in \Sigma^*$ ein Urbild x mit $h(x) = y$ dingfest machen:

Fall 1 Die R-Kette für y stoppt mit undefiniertem S^{-1}.
In diesem Fall hat die R-Kette die Form

$$R^{-1}(S^{-1}(\cdots R^{-1}(y)\cdots)) \ .$$

Wir definieren dann $x = R^{-1}(y)$. Somit stoppt die S-Kette für x (welche ein Präfix der R-Kette für y ist) mit undefiniertem S^{-1} und es gilt (gemäß der Definition von h)

$$h(x) = R(x) = R(R^{-1}(y)) = y \ .$$

Fall 2 Die R-Kette für y stoppt mit undefiniertem R^{-1}.
In diesem Fall hat die R-Kette die Form

$$S^{-1}(R^{-1}(\cdots R^{-1}(y)\cdots)) \ .$$

Wir definieren dann $x = S(y)$, was $y = S^{-1}(x)$ zur Folge hat. Somit stoppt die S-Kette für x (welche die R-Kette für y zum Präfix hat) mit undefiniertem R^{-1} und es gilt (gemäß der Definition von h)

$$h(x) = S^{-1}(x) = S^{-1}(S(y)) = y \ .$$

Es hat sich somit die Surjektivität von h ergeben.
Wir diskutieren nun die Berechnungskomplexität der Funktionen h und h^{-1}. Zur Berechnung von $h(x)$ müssen wir zunächst die vollständige S-Kette für x berechnen, um festzustellen, ob sie mit undefiniertem R^{-1} oder mit undefiniertem S^{-1} stoppt. Dies erfordert lediglich n Auswertungen der Funktionen R^{-1} bzw. S^{-1}, was in Polynomialzeit geleistet werden kann. Nachdem klar ist, welcher Fall vorliegt, muss dann entweder $h(x) = S^{-1}(x)$ oder $h(x) = R(x)$ berechnet werden. Auch dies ist in Polynomialzeit möglich. Die Berechnung von h^{-1} ist aus Symmetriegründen genau so aufwendig wie die Berechnung von h. Wenn wir nämlich die obige Diskussion zur Surjektivität von h noch einmal aufmerksam durchlesen, stellen wir fest, dass die Definition von h^{-1} symmetrisch zur Definition von h (mit vertauschten Rollen von R und S) ist. Also sind sowohl h als auch h^{-1} in Polynomialzeit berechenbar.
Um den Nachweis der Isomorphie-Bedingungen für h zu vervollständigen, müssen wir lediglich noch zeigen, dass jeder String $x \in \Sigma^*$ zu L_1 gehört gdw $h(x)$ zu L_2 gehört. Dies ist aber klar, da entweder $h(x) = S^{-1}(x)$ oder $h(x) = R(x)$ und sowohl S^{-1} als auch R (in ihrer Funktion als Reduktionsabbildungen) die geforderte Eigenschaft besitzen. **qed.**

Folgerung 8.6 (Berman und Hartmanis, 1977) *Alle NP-vollständigen Sprachen, die mit einer Polsterfunktion versehen werden können, sind zueinander polynomiell isomorph.*

Die Tatsache, dass alle bekannten NP-vollständigen Probleme eine Polsterfunktion besitzen, führte zu der

Isomorphie-Vermutung (Berman und Hartmanis, 1977) Alle NP-vollständigen Sprachen sind zueinander isomorph.

Es ist bis heute nicht geklärt, ob diese Vermutung zutrifft.

Definition 8.7 *Die* Dichte *bzw.* Dichtefunktion *einer Sprache $L \subseteq \Sigma^*$ ist definiert als die Folgende Funktion in der Eingabelänge n:*

$$dens_L(n) := |\{x \in L : \ |x| \leq n\}| \ .$$

L heißt dünn *(oder auch* spärlich*), wenn ihre Dichte polynomiell in n beschränkt ist; L heißt* dicht*, wenn ihre Dichte superpolynomiell mit n wächst.*

Alle uns bisher bekannten NP-vollständigen Sprachen sind dicht!

Beobachtung 8.8 *Wenn L_1 und L_2 polynomiell isomorph sind, dann gibt es Polynome q_1, q_2, so dass*

$$dens_{L_1}(n) \leq dens_{L_2}(q_1(n)) \text{ und } dens_{L_2}(n) \leq dens_{L_1}(q_2(n)) \ .$$

Hieraus folgt, dass eine dünne und eine dichte Sprache nicht zueinander polynomiell isomorph sein können.

Wenn die Isomorphie-Vermutung zutrifft, dann würde aus dieser Beobachtung folgen, dass eine dünne Sprache nicht NP-vollständig sein kann. Diese Folgerung kann jedoch bereits unter der schwächeren $P \neq NP$-Voraussetzung bewiesen werden:

Satz 8.9 (Mahaney, 1982) *Falls $P \neq NP$, dann gibt es keine dünne und zugleich NP-vollständige Sprache.*

Ein Spezialfall von dünnen Sprachen stellen die *unären Sprachen* $L \subseteq \{0\}^*$ dar. Anstelle des Satzes von Mahaney beweisen wir den folgenden (etwas leichteren)

Satz 8.10 (Berman, 1978) *Falls $P \neq NP$, dann gibt es keine unäre und zugleich NP-vollständige Sprache.*

Beweis Wir führen den Beweis indirekt und zeigen, dass die Existenz einer NP-vollständigen unären Sprache $L_0 \subseteq \{0\}^*$ die Gleichheit von P und NP zur Folge hat. Dazu genügt es, mit Hilfe einer Reduktionsabbildung $R : \Sigma^* \to \Sigma^*$ für SAT $\leq_{pol} L_0$ einen deterministischen polynomiell zeitbeschränkten Algorithmus für SAT zu entwerfen. Sei also $x \in \Sigma^N$ der Kodierungsstring für eine CNF-Formel F über den Booleschen Variablen $v_1, \ldots, v_n$. Wir

skizzieren im Folgenden ein effizientes Verfahren, um die Erfüllbarkeit von F zu testen. Dabei identifizieren wir einen String $a \in \{0,1\}^j$ mit der partiellen Belegung

$$v_1 = a_1, \ldots, v_j = a_j \ .$$

Es bezeichne $F[a]$ die vereinfachte CNF-Formel, die aus F durch die von a repräsentierte partielle Belegung hervor geht.[3] Der Kodierungsstring für $F[a]$ sei mit $x[a]$ bezeichnet. Für $a = \epsilon$ (leeres Wort bzw. leere partielle Belegung) identifizieren wir $F[a]$ mit F und $x[a]$ mit x. Offensichtlich gilt für $j = 0, \ldots, n-1$ und alle $a \in \{0,1\}^j$ die Beziehung

$$F[a] \text{ ist erfüllbar} \Leftrightarrow F[a0] \text{ oder } F[a1] \text{ ist erfüllbar.}$$

Für alle $a \in \{0,1\}^n$ ist $F[a]$ eine Boolesche Konstante und somit erfüllbar gdw es sich um die Konstante 1 handelt. Ein exponentiell zeitbeschränkter Algorithmus könnte vorgehen wie folgt:

Top-down Pass Bilde einen vollständig binären Baum T der Tiefe n. Assoziiere mit einer Kante zum linken Kind das Bit 0 und mit einer Kante zum rechten Kind das Bit 1. Auf diese Weise ist zu jedem Knoten z der Tiefe j des Baumes ein eindeutiger Binärstring $a(z) \in \{0,1\}^j$ assoziiert. Insbesondere entsprechen die 2^n Blätter (Knoten der Tiefe n) 1-zu-1 den Strings aus $\{0,1\}^n$.

Bottom-up Pass Markiere jedes Blatt z mit der Booleschen Konstante $F[a(z)]$. Markiere anschließend jeden inneren Knoten z mit bereits markierten Kindern z_0, z_1 mit der Disjunktion (logisches „Oder“) der Booleschen Markierungen seiner Kinder.

Nach unseren obigen Ausführungen sollte klar sei, dass eine Formel $F[a(z)]$ mit 1 markiert wird gdw $F[a(z)]$ erfüllbar ist. Die Markierung an der Wurzel verrät uns, ob F erfüllbar ist. Der „Schönheitsfehler“ dieses Verfahrens ist die exponentielle Größe der verwendeten Datenstruktur T. Jetzt kommen zwei Ideen ins Spiel, die darauf hinaus laufen, dass nur ein polynomiell kleines Anfangsstück T' von T wirklich gebraucht wird:

Lazy Evaluation Wenn $F[a0]$ erfüllbar ist, dann ist (unabhängig von der Erfüllbarkeit von $F[a1]$) auch die Formel $F[a]$ erfüllbar. Im Baum T bedeutet dies für einen Knoten z mit linkem Kind z_0 und rechtem Kind z_1: wenn z_0 beim „bottom-up pass“ die Markierung 1 erhalten hat, brauchen wir uns für den Unterbaum mit Wurzel z_1 nicht mehr zu interessieren und können z direkt mit 1 markieren. Nach welcher Strategie muss der Baum T gebildet werden, um diesen Effekt voll auszunutzen? Die Antwort lautet: „Durchlauf in Präordnung (preorder traversal)“.[4] Der Durchlauf in Präordnung stellt sicher, dass wir den „bottom-up pass“ des Unterbaumes von z_0 bereits abgeschlossen haben, bevor wir in den „top-down pass“ des Unterbaumes von z_1 einsteigen. Abbildung 1 illustriert diesen Gedankengang.

[3] Elimination aller durch die partielle Belegung bereits erfüllten Klauseln und, in den verbleibenden Klauseln, Elimination der durch die partielle Belegung bereits falsifizierten Literale

[4] Durchlauf eines Baumes mit Wurzel r, Wurzelkindern r_0, r_1 und den Unterbäumen T_0, T_1 in Präordnung ist rekursiv definiert wie folgt: durchlaufe zuerst r dann (rekursiv) alle Knoten von T_0 und schließlich (rekursiv) alle Knoten von T_1.

Hashing Wir verwenden die Reduktionsabbildung R als eine Art „Hashfunktion“. Dabei nutzen wir aus, dass zwei CNF-Formeln F, F' mit Kodierungsstrings x, x' und $R(x) = R(x')$ entweder beide erfüllbar sind (nämlich, wenn $R(x) = R(x') \in L_0$) oder beide nicht erfüllbar sind (nämlich, wenn $R(x) = R(x') \notin L_0$). Im Baum T bedeutet dies für einen Knoten z': falls zum Zeitpunkt des Kreierens von z' bereits ein Knoten z in T existiert, welcher mit einem Bit markiert ist und die Bedingung $R(x[a(z)]) = R(x[a(z')]$ erfüllt, dann können wir z' direkt mit dem gleichen Bit markieren. (Da wir für z' keine Kinder kreieren müssen, wird z' dann zu einem Blatt des Baumes T'.)

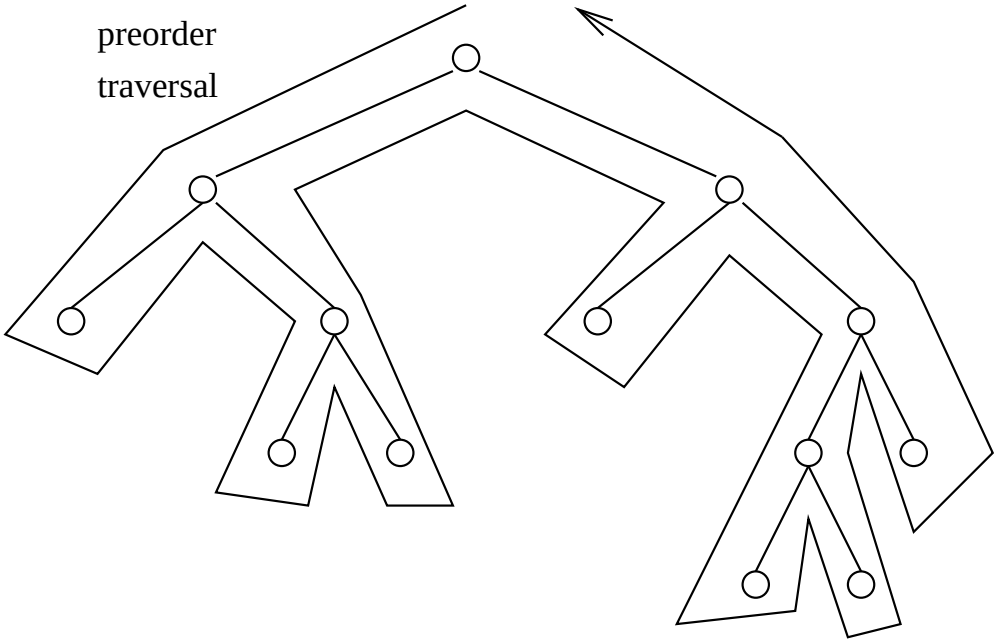

Abbildung 1: Ein Durchlauf der Knoten eines Baumes in Präordnung: bei zwei unabhängigen Knoten z, z' mit z links von z' ist der „bottom-up pass“ von z abgeschlossen, bevor der „top-down pass“ von z' beginnt.

Beide Ideen verfolgen im Kern eine „pruning“-Technik: wir können von dem exponentiell großen Baum T Teilbäume „abschneiden“, wenn sie für unsere Markierungsstrategie nur redundante Information liefern. Es sollte klar sein, dass der komplette Algorithmus für SAT (unter Einsatz von „pruning“) in Polynomialzeit implementiert werden kann, wenn das Anfangsstück T' von T (T ohne die abgeschnittenen bzw. gar nicht erst kreierten Teilbäume) eine polynomiell beschränkte Größe hat. Wir argumentieren nun, dass dies der Fall ist. Zunächst setzen wir oBdA die Bedingung

$$\forall x \in \Sigma^* : R(x) \in \{0\}^*$$

voraus. Wir können nämlich jeden String $R(x)$, der neben 0 noch weitere Buchstaben verwendet, durch einen beliebigen (fest ausgewählten) String aus $\{0\}^* \setminus L_0$ ersetzen.[5] Da R in

[5]Wegen der (angenommenen) *NP*-Vollständigkeit von L_0 muss L_0 eine echte Teilmenge von $\{0\}^*$ sein!

Polynomialzeit berechenbar ist, muss es ein Polynom q geben, so dass für alle x gilt

$$|x| \leq N \Rightarrow |R(x)| < q(N) \ .$$

Da von R nur unäre Bilder produziert werden, induzieren die Strings der Maximallänge N maximal $q(N)$ paarweise verschiedene Bilder unter R. Das bedeutet, dass die Anzahl der bei unserem SAT-Algorithmus auftretenden Hashwerte durch $q(N)$ beschränkt ist. Wie ist nun die Beziehung zwischen der Anzahl m der Knoten in T' und der maximalen Anzahl $q(N)$ von Hashwerten? Die entscheidende Beobachtung ist die folgende: zwei unabhängige[6] innere Knoten z, z' in T' haben verschiedene Hashwerte. Der Grund hierfür ist einfach: einer von beiden, sagen wir z, wurde zuerst durchlaufen und mit einem Bit markiert, bevor der andere, also z', besucht wird; wäre $R(z) = R(z')$ dann hätten wir zu z' keine Kinder kreiert und z' müsste ein Blatt sein. Nun bestimmen wir die Knotenanzahl m wie folgt:

- Es bezeichne m' die Anzahl der Knoten der Höhe 1 in T' (der Level direkt oberhalb der Blätter).
- Da die Knoten der Höhe 1 paarweise unabhängige, innere Knoten von T' sind, müssen sie paarweise verschiedene Hashwerte haben. Somit gilt $m' \leq q(N)$.
- Da die Höhe von T' durch n beschränkt ist, ist die Anzahl der inneren Knoten von T' beschränkt durch n-mal die Anzahl der Knoten wie auf Höhenlevel 1, also durch $nq(N)$.
- Da jeder Knoten in T' maximal zwei Kinder hat, gibt es in T' maximal ein Blatt mehr als innere Knoten, also maximal $nq(N) + 1$ Blätter.

Insgesamt besitzt T' also maximal $2nq(N) + 1$ Knoten. Da $n \leq N$ und $N = |x|$ die Eingabelänge bezeichnet, ist der Beweis abgeschlossen. **qed.**

Die Sätze 8.9 und 8.10 bedeuten im Umkehrschluss: wenn es gelingt eine NP-vollständige unäre bzw. dünne Sprache ausfindig zu machen, dann ist $P = NP$.

9 Relativierung der P versus NP Frage

Baker, Gill und Solovay sind in einer Aufsehen erregenden Publikation aus dem Jahre 1975 der Frage nachgegangen, ob die Frage der (Un-)gleichheit von P und NP in einer durch die Anwesenheit von Orakeln „relativierten Welt“ entschieden werden kann. In einer solchen Welt würde die Rechenkraft von DTMs oder NTMs künstlich erhöht, indem bestimmte (einer Relation oder formalen Sprache entsprechende) Anfragen an ein Orakel weiter geleitet werden können und von diesem dann jeweils in einem Schritt beantwortet werden (Konzept der Orakel-Turing-Maschinen oder kurz OTMs, DOTMs, NOTMs). Warum sollte die Anwesenheit eines Orakel die Beantwortung der P versus NP Frage erleichtern? Hier sind zwei Gründe:

[6] „unabhängig“ bedeutet „kein Knoten ist Vorfahr des anderen“.

- Um den Unterschied zwischen P und NP zu nivellieren, könnte man ein Orakel benutzen, welches deterministische Maschinen so mächtig macht, dass durch Nichtdeterminismus keine zusätzlichen Fähigkeiten erwachsen.

- Um den Unterschied zwischen P und NP zu verstärken (und dadurch mathematisch beweisbar zu machen), könnte man Orakel benutzen, die die Fähigkeiten nichtdeterministischer Maschinen signifikant stärker potenzieren als die von deterministischen.

Es wird sich zeigen, dass in der Tat beide Grundideen durchführbar sind. In diesem Sinne ergibt sich durch Relativierung der P versus NP Frage gewissermaßen ein „unentschieden“: für manche Relativierungen gilt $P = NP$, für andere gilt $P \neq NP$.

Also was (so what)?

Diese Untersuchungen zeigen in erster Linie auf, dass klassische Techniken zum Nachweis der (Un-)gleichheit zweier Komplexitätsklassen in Bezug auf die P versus NP Frage zum Scheitern verdammt sind:

- Eine klassische Technik zum Nachweis von $\mathcal{C} = \mathcal{C}'$ (Gleichheit zweier Komplexitätsklassen) ist *(wechselseitige) Schritt-für-Schritt Simulation*: zeige, dass jeder Rechenschritt einer zu $\mathcal{C}$ passenden Maschine M durch (evtl. mehrere) Rechenschritte einer zu $\mathcal{C}'$ passenden Maschine M' simuliert werden kann (und umgekehrt). Diese Beweistechnik lässt sich relativieren: wenn beide Maschinen zusätzlich mit einem Orakel versehen sind, dann kann die Simulation aufrecht erhalten werden, indem herkömmliche Rechenschritte simuliert werden wie zuvor und eine Orakelanfrage von M „simuliert wird“ durch dieselbe Orakelanfrage von M'. Wenn sich also $\mathcal{C} = \mathcal{C}'$ mit Hilfe von Schritt-für-Schritt Simulation nachweisen lässt, dann sollte die Gleichheit in **jeder** relativierten Welt gelten.

- Eine klassische Technik zum Nachweis von $\mathcal{C} \subset \mathcal{C}'$ (echte Inklusion) ist *(einseitige) Schritt-für-Schritt Simulation* verbunden mit *Diagonalisierung*: wenn $M_1, M_2, \ldots$ eine Aufzählung aller zu $\mathcal{C}$ passenden Maschinen ist und $z_1, z_2, \ldots$ ist eine Aufzählung aller Eingabestrings, dann versuche eine zu $\mathcal{C}'$ passende Maschine M' anzugeben, die auf Eingabe z_i die Maschine M_i Schritt-für-Schritt simuliert, aber am Ende der Rechnung die Antwort JA/NEIN zu NEIN/JA rumdreht. M' ist dann Akzeptor einer Sprache aus $\mathcal{C}' \setminus \mathcal{C}$. Auch diese Beweistechnik lässt sich (in der offensichtlichen Weise) relativieren. Wenn sich also $\mathcal{C} \subset \mathcal{C}'$ mit Hilfe von (auf Schritt-für-Schritt Simulation basierender) Diagonalisierung nachweisen lässt, dann sollte die echte Inklusionsbeziehung in **jeder** relativierten Welt gelten.

Diese Überlegungen zeigen im Umkehrschluss: da die Antwort auf die P versus NP Frage in den relativierten Welten zweideutig ausfällt, kann weder die Gleicheit noch die Ungleichheit von P und NP mit einer relativierenden Technik (wie Schritt-für-Schritt Simulation, Diagonalisierung,...) geführt werden.[7]

[7] Viele „Beweise“ für $P = NP$ oder $P \neq NP$, die (mehr oder weniger qualifizierte) Forscher der staunenden Fachwelt präsentiert haben, scheitern bereits an dieser Hürde: es wurde eine relativierende Beweistechnik verwendet. Ein solcher Beweis kann von vorne herein nicht korrekt sein.

Nach diesen philosophischen Anmerkungen kommen wir nun zur eigentlichen Arbeit. Es sei daran erinnert, dass M^R bzw. M^L eine Orakel-Turing-Maschine (OTM) mit einem Orakel für die Relation R bzw. für die Sprache L bezeichnet. Wenn M einen Fragestring x auf das spezielle Orakelband schreibt und in den Fragezustand $q_?$ übergeht, dann ersetzt das R-Orakel (in **einem** Schritt) den String x durch einen String y mit $(x, y) \in R$ (sofern möglich) und „beamt" M in den Zustand q_+. Wenn es keinen passenden String y gibt, dann wird x nicht ersetzt und in den Zustand q_- „gebeamt". Im Falle eines L-Orakels wird im Falle $x \in L$ der String x gelöscht und M in Zustand q_+ „gebeamt"; falls $x \notin L$, dann wird x nicht gelöscht und M in Zustand q_- „gebeamt". Wenn M deterministisch ist, sprechen wir von einer DOTM M^R bzw. M^L; ist M nichtdeterministisch sprechen wir von einer NOTM. Wenn $\mathcal{C}$ eine (deterministische oder nicht-deterministische) Zeitkomplexitätsklasse bezeichnet, dann sei $\mathcal{C}^R$ bzw. $\mathcal{C}^L$ die Klasse aller Sprachen, die sich mit einer OTM M^R bzw. M^L erkennen lassen, wobei M eine zu $\mathcal{C}$ passende Maschine sein muss.

Welches Orakel könnte geeignet sein, den Unterschied zwischen P und NP (beweisbar) zu nivellieren? Zur Beantwortung dieser Frage erinnern wir an die Definition von PSpace:

$$PSpace = \bigcup_{k \geq 1} DSpace(n^k) \ .$$

Wir werden in Bälde den Satz von Savitch beweisen, der im Wesentlichen besagt, dass eine NTM mit Platzschranke $S(n) \geq \log n$ sich durch eine DTM mit Platzschranke $S^2(n)$ simulieren lässt. Das impliziert

$$PSpace = \bigcup_{k \geq 1} NSpace(n^k) \ .$$

Vereinfacht gesagt: PSpace nivelliert den Unterschied zwischen Determinismus und Nichtdeterminismus. Diese Tatsache kommt im Beweis des folgenden Resultates zum Tragen:

Satz 9.1 (Baker, Gill, and Solovay, 1975) *Es gibt eine Sprache A mit $P^A = NP^A$.*

Beweis Es sei A eine (unter Karp-Reduktionen)*PSpace*-vollständige Sprache.[8] Dann folgt $P^A = NP^A$ aus

$$PSpace \subseteq P^A \subseteq NP^A \subseteq \bigcup_{k \geq 1} NSpace(n^k) = \bigcup_{k \geq 1} DSpace(n^k) = PSpace \ . \qquad (1)$$

Die erste Inklusion gilt wegen der *PSpace*-Vollständigkeit von A. Sei nämlich $L \in PSpace$, R eine Reduktionsabbildung für $L \leq_{pol} A$ und $x \in \Sigma^n$ ein Eingabestring. Um die Frage der Mitgliedschaft von x in L zu entscheiden, können wir zunächst in poly(n) Schritten $R(x)$ berechnen und dann das A-Orakel nach der Mitgliedschaft von $R(x)$ in A befragen. Die zweite Inklusion in (1) ist trivial. Die dritte Inklusion in (1) ergibt sich daraus, dass eine NOTM M^A für $L \in NP^A$ durch eine NTM M' mit polynomieller Platzschranke simuliert

[8] Beispiele solcher Sprachen werden wir in einem späteren Abschnitt kennen lernen.

werden kann: wann immer M^A das A-Orakel nach der Mitgliedschaft eines Strings z zur Sprache A befragt, kann sich M' (wegen $A \in PSpace$) die Antwort selber herleiten. Die vorletzte Gleichung ergibt sich aus dem Satz von Savitch, und die letzte Gleichung ist die Definition von PSpace. **qed.**

Ein Orakel, dass P von NP trennt, ist etwas schwieriger zu konstruieren, existiert aber ebenfalls:

Satz 9.2 (Baker, Gill, and Solovay, 1975) *Es gibt eine Sprache B mit $P^B \neq NP^B$.*

Beweis Wir konstruieren eine Sprache B und eine (von B abhängige) Sprache $L_* \in NP^B \setminus P^B$ und beginnen mit der Definition von L_*:

$$L_* := \{0^n | \ B \cap \Sigma^n \neq \emptyset\} \ .$$

Bereits ohne Kenntnis von B ist sonnenklar, dass L_* zur Klasse NP^B gehört, da die Wörter $0^n \in L_*$ mit Hilfe des B-Orakels nichtdeterministisch in Polynomialzeit erkannt werden können: rate $x \in \Sigma^n$ und verifiziere $x \in B$ durch Anfrage an das B-Orakel. Die Kampfaufgabe im restlichen Beweis wird darin bestehen, B so zu definieren, dass $L_* \notin P^B$. Zu diesem Zweck greifen wir auf die Technik der Diagonalisierung zurück. Es bezeichne

$$M_1^B, M_2^B, \ldots$$

eine Aufzählung[9] aller polynomiell zeitbeschränkter DOTMs mit Orakel B, so dass

$$P^B = \{L_{M_i^B} | \ i \geq 1\} \ .$$

Wir setzen zusätzlich voraus, dass jede Sprache in P^B, die in Zeit n^c erkannt werden kann in der Aufzählung mit unendlich vielen Akzeptoren vertreten ist (was durch „redundante Kodierung“ von Turing-Maschinen-Programmen leicht zu erreichen ist). Wir definieren nun B stufenweise, wobei

$$B_i = B \cap \Sigma^i$$

die *i-the Stufe* von B darstellt. Nebenbei protokollieren wir eine Menge X von „Ausnahmestrings“, die wir auf keinen Fall in B aufzunehmen wünschen. Anfangs gilt

$$B_0 = X = \emptyset \ .$$

Die Menge X ist aber eine dynamische Größe, die sich im Laufe der Konstruktion von B verändert. In Stufe i der Konstruktion wollen wir im Prinzip erreichen, dass der String 0^i die Sprachen L_* und $L_{M_i^B}$ trennt:

[9]Diese Aufzählung hängt **nicht** von B ab, da im Programm einer DOTM M_i lediglich festgelegt ist, wie nach Erreichen der Zustände q_- bzw. q_+ weiter gearbeitet wird. Die Abhängigkeit von B ergibt sich so zu sagen erst zur Laufzeit.

Stufe i „Simuliere“ M_i^B auf Eingabestring 0^i für

$$q(i) := i^{\lfloor i \rfloor}$$

viele Schritte.[10] Bei der Simulation kann M_i^B das B-Orakel nach der Mitgliedschaft von Strings z zur Sprache B befragen. Falls $|z| \leq i-1$, antworten wir in der Simulation mit JA, falls $z \in B_{|z|}$, und mit NEIN, falls $z \notin B_{|z|}$. Beachte dabei, dass die $|z|$-te Stufe von B bereits vorher konstruiert wurde. Falls aber $|z| \geq i$, dann antworten wir mit NEIN und nehmen z in X auf (was die Antwort NEIN im Nachhinein rechtfertigen wird, da wir Strings aus X niemals in B aufnehmen werden). Auf diese Weise kann die Simulation am Laufen gehalten werden. Beachte, dass in Stufe i maximal $q(i)$ Strings in X aufgenommen werden. Eine leichte Rechnung ergibt, dass

$$\sum_{j=1}^{i} q(j) < 2^i \ ,$$

d.h., nach Stufe i muss $X \cap \Sigma^i$ eine echte Teilmenge von Σ^i sein. Nun sind drei Fälle denkbar:

Fall 1 M_i^B verwirft 0^i in maximal $q(i)$ Schritten.
Dann setzen wir

$$B_i := \Sigma^i \setminus X \neq \emptyset \ .$$

Nun gilt $0^i \notin L_{M_i^B}$ und (gemäß der Definition von L_*) $0^i \in L_*$.

Fall 2 M_i^B akzeptiert 0^i in maximal $q(i)$ Schritten.
Dann setzen wir

$$B_i := \emptyset$$

mit der Konsequenz, dass $0^i \in L_{M_i^B}$ und (wieder gemäß der Definition von L_*) $0^i \notin L_*$.

Fall 3 M_i^B kommt im Laufe von $q(i)$ Schritten zu keiner Entscheidung.
In diesem Fall setzen wir auch $B_i := \emptyset$, gelangen aber (im Unterschied zu den ersten beiden Fällen) zu keiner Separation von L_* und $L_{M_i^B}$.

Ohne den Fall 3 wäre der Diagonalisierungsbeweis nun abgeschlossen. Erinnern wir uns daran, dass wir listigerweise verlangt hatten, dass jede Sprache L, die mit Zeitschranke n^c erkennbar ist, in der Aufzählung der M_i unendlich oft vertreten ist. Da $q(i)$ schneller wächst als jedes Polynom, wird für Akzeptoren von L mit hinreichend großem Index i der Fall 3 nicht eintreten. Somit kann jede Sprache aus P^B von L_* getrennt werden. **qed.**

[10] Die Funktion $q(i)$ ist so gewählt, dass sie asymptotisch schneller wächst als jedes Polynom, aber langsamer als 2^i.

10 Beziehungen zwischen den Komplexitätsklassen

10.1 Kontrolle der Ressourcenschranken

Eine Schlüsseltechnik zur Analyse der Beziehungen zwischen verschiedenen Komplexitätsklassen ist die Simulation einer TM M durch eine andere TM M'. Dabei wird es von Bedeutung sein, dass M' ihre Ressourcenschranken nicht überschreitet. Unter Umständen muss eine Simulation von M durch M' erfolglos abgebrochen werden, wenn sie zur Überschreitung der Ressourcenschranke führen würde. Wie aber kann M' die Kontrolle darüber bewahren? Die Beantwortung dieser Frage beruht auf den Konzepten der Platz- und Zeitkonstruierbarkeit.

Definition 10.1 (Platzkonstruierbarkeit) *Eine Funktion $S : \mathbb{N} \to \mathbb{N}$ heißt* platzkonstruierbar*, wenn die Funktion $1^n \mapsto 1^{S(n)}$ von einer $S(n)$-platzbeschränkten DTM berechnet werden kann.*

Die Hauptanwendung der Platzkonstruierbarkeit besteht darin, ein Bandsegment der Größe $S(n)$ abzustecken (zum Beispiel mit Endmarkierungen auf der Zelle am weitesten links bzw. rechts). Dies geschieht oft in Verbindung mit einer Simulation, welche abgebrochen wird, wenn sie zum Verlasssen des abgesteckten Bandsegmentes führen würde.

Definition 10.2 (Zeitkonstruierbarkeit) *Eine Funktion $T : \mathbb{N} \to \mathbb{N}$ heißt* zeitkonstruierbar*, wenn die Funktion $1^n \mapsto bin(T(n))$ von einer $T(n)$-zeitbeschränkten 2-Band DTM M berechnet werden kann.*

Die Hauptanwendung der Zeitkonstruierbarkeit besteht darin, einen Binärzähler mit $\mathrm{bin}(T(n))$ zu initialisieren und dann in Verbindung mit einer Simulation als „Uhr“ zu verwenden: nach jedem Schritt in der Simulation wird der Zähler um 1 dekrementiert; erreicht er den Wert 0 (Zeit abgelaufen), dann wird die Simulation abgebrochen.[11]

Wir merken kurz an: offensichtlich sind zeitkonstruierbare Funktionen erst recht platzkonstruierbar. Die folgenden Beobachtungen zeigen, dass die Klasse der konstruierbaren Funktionen sehr reichhaltig ist: **Übg.**

1. Jede konstante Funktion $n \mapsto c$ mit $c \in \mathbb{N}$ ist platz- und zeitkonstruierbar.
2. Die Funktionen $n \mapsto |\mathrm{bin}(n)| = 1 + \lfloor \log n \rfloor$, $n \mapsto \lfloor \log n \rfloor$ und $n \mapsto \lceil \log n \rceil$ sind platzkonstruierbar.
3. Die Funktionen $n \mapsto n$ und $n \mapsto 2^n$ sind platz- und zeitkonstruierbar.
4. Platz- und zeitkonstruierbare Funktionen sind abgeschlossen unter Addition und Multiplikation.
5. Alle Polynome mit Koeffizienten aus $\mathbb{N}$ (aufgefasst als Funktionen von $\mathbb{N}$ nach $\mathbb{N}$) sind platz- und zeitkonstruierbar.

[11] Mit einer amortisierten Analyse lässt sich zeigen, dass der „Overhead“ zum Zurückzählen des Binärzählers von $\mathrm{bin}(T(n))$ auf 0 durch $O(T(n))$ Rechenschritte beschränkt ist.

Konvention Mit einem „Polynom“ meinen wir im Folgenden stets ein Polynom mit Koeffizienten aus $\mathbb{N}$, das als Funktion von $\mathbb{N}$ nach $\mathbb{N}$ (oder manchmal auch als Funktion von $\mathbb{N}_0$ nach $\mathbb{N}_0$) aufgefasst wird.

10.2 Verhältnis von Determinismus und Nicht-Determinismus

R, S, T seien Ressourcenschranken. Trivialerweise gilt

$$DTime(T) \subseteq NTime(T) \text{ und } DSpace(S) \subseteq NSpace(S) \ ,$$

da sich eine DTM als Spezialfall einer NTM auffassen lässt. Die uns bereits bekannte deterministische Simulation von NTMs impliziert folgenden

Satz 10.3 *Für jede platzkonstruierbare Funktion T gilt:*

$$NTime(T) \subseteq DTime\left(2^{O(T)}\right) \quad \textit{und} \quad NTime(T) \subseteq DSpace(T) \ .$$

Bevor wir zur nächsten Simulation über gehen, schieben wir eine kleine Zwischenüberlegung ein. Bei einer platzbeschränkten Turing-Maschine ist auf Anhieb nicht klar, ob sie überhaupt auf jeder Eingabe nach endlich vielen Schritten anhält. Allerdings gibt es nur endlich viele Konfigurationen, in die sie auf Eingaben der Länge n geraten kann. Für eine $S(n)$-platzbeschränkte k-Band Turing-Maschine mit einem zusätzlichen Read-only Eingabeband[12] zum Beispiel erhalten wir für die Menge $\mathcal{K}_n$ der möglichen Konfigurationen für Eingaben der Länge n die folgende Ungleichung:

$$|\mathcal{K}_n| \leq |Z| \cdot n \cdot S^k(n) \cdot |\Gamma|^{kS(n)} \ .$$

Hierbei ist $|Z|$ ist die Anzahl der Zustände, n die Anzahl der Positionen des Lesekopfes für die Eingabe, $S(n)$ die Anzahl der Positionen des Kopfes für ein Arbeitsband und $|\Gamma|^{S(n)}$ die Anzahl der Beschriftungen eines Arbeitsbandes. Da es insgesamt k Arbeitsbänder gibt gehen die Faktoren $S(n)$ und $|\Gamma|^{S(n)}$ in die angegebene Schranke in der k-ten Potenz ein. Im Falle $S(n) \geq \log n$ gilt offensichtlich:

$$|\mathcal{K}_n| = 2^{O(S(n))} \ .$$

Der folgende Satz besagt, dass es relativ platzeffiziente deterministische Simulationen nichtdeterministischer Maschinen gibt (lediglich quadratischer „blow-up“ der Platzschranke):

Satz 10.4 (von Savitch) *Es sei $S(n) \geq \log n$ eine platzkonstruierbare Funktion. Dann gilt*

$$NSpace(S) \subseteq DSpace^2(S) \ .$$

[12] benötigt für sublineare Platzschranken: nur die auf den k Arbeitsbändern verwendeten Zellen werden beim Platzverbrauch gezählt.

Beweis Es sei $L \in NSpace(S)$ und M eine NTM mit Platzschranke S und $L = L_M$. Wegen des 1. Bandreduktionstheorems dürfen wir voraus setzen, dass M lediglich ein Arbeitsband (und evtl. ein weiteres Read-only Eingabeband) besitzt. Wegen $S(n) \geq \log n$ existiert eine Konstante c, so dass M auf Eingaben der Länge n maximal $2^{cS(n)}$ Konfigurationen annehmen kann. Es bezeichne $K_0(w)$ die Startkonfiguration von M (bei Eingabe $w \in \Sigma^n$), K_+ eine oBdA eindeutige akzeptierende Endkonfiguration und $\mathcal{K}_n$ die Menge aller auf Eingaben der Länge n denkbaren Konfigurationen (mit $|\mathcal{K}_n| \leq 2^{cS(n)}$). Die Platzkonstruierbarkeit von $S(n)$ wird im Folgenden benötigt, um alle Konfigurationen aus $\mathcal{K}_n$ systematisch (zum Beispiel in lexicographischer Ordnung) durchlaufen zu können.
Betrachte folgende Relation auf $\mathcal{K}_n$:

$$K \vdash_M^t K' :\Leftrightarrow M \text{ kann } K \text{ in maximal } t \text{ Rechenschritten nach } K' \text{ überführen.}$$

Es sei $T := 2^{cS(n)}$. Beachte, dass jede Rechnung von M mit mindestens T Schritten eine Konfiguration mehrfach durchläuft. „Zykelfreie“ Rechnungen von M auf Eingabe w dauern daher weniger als T Schritte. Somit gilt:

$$w \in L_M \Leftrightarrow K_0(w) \vdash_M^T K_+ \ .$$

Wir wollen aufzeigen, wie die Beziehung $K_0(w) \vdash_M^T K_+$ *deterministisch* mit Platzschranke S^2 getestet werden kann. Der Schlüssel hierzu liegt in folgendem rekursiven Ansatz:

$$K \vdash_M^{2^s} K' \Leftrightarrow \exists K'' \in \mathcal{K}_n : K \vdash_M^{2^{s-1}} K'' \wedge K'' \vdash_M^{2^{s-1}} K' \ .$$

Es bezeichne TEST eine rekursive Boolesche Prozedur, die auf Parametern K, K', s den Wert TRUE ausgibt gdw $K \vdash_M^{2^s} K'$. Für $s = 0$ kann diese Prozedur leicht ausgewertet werden, da $K \vdash_M^{2^0} K'$ bedeutet, dass K' eine direkte Folgekonfiguration von K bezüglich der NTM M ist. Für $s \geq 1$ kann TEST rekursiv ausgewertet werden:

$$\text{TEST}(K, K', s) := \bigvee_{K'' \in \mathcal{K}_n} \text{TEST}(K, K'', s-1) \wedge \text{TEST}(K'', K', s-1) \ .$$

Um $w \in L$ zu testen, starten wir den Aufruf $\text{TEST}(K_0(w), K_+, cS(n))$. Es resultiert ein Rekursionsbaum der Tiefe $cS(n)$.
Wir haben abschließend die Frage zu klären, wie eine S^2-platzbeschränkte DTM M' den Prozeduraufruf $\text{TEST}(K_0(w), K_+, cS(n))$ (und die dadurch ausgelösten rekursiven Aufrufe) implementieren kann. Die wesentliche Idee ist, den Rekursionsbaum niemals vollständig aufs Band zu schreiben, sondern immer nur den aktuellen „Ariadne-Faden“. Dies sind die Informationen, die einem Pfad im Rekursionsbaum von der Wurzel (erster Aufruf von TEST) bis zum aktuellen Knoten (aktueller Aufruf von TEST) entsprechen. Zur Speicherung des Ariadne-Fadens verwendet M' eines ihrer Bänder als Kellerspeicher. Zu jedem neuen Knoten auf dem Ariadne-Faden (neuer rekursiver Aufruf $\text{TEST}(K, K', s)$) wird dabei ein neuer Informationsblock (genannt $\text{FRAME}(K, K', s)$) in den Keller gepusht. Wir werden weiter unten sehen, dass ein FRAME die Länge $O(S(n))$ besitzt. Da der Ariadne-Faden maximal aus $cS(n)$ Knoten besteht (dies ist die Tiefe des Rekursionsbaumes) ergibt sich somit die

Platzschranke S^2.
Bleibt zu zeigen, dass eine FRAME-Größe von $O(S(n))$ Zellen ausreicht, um die Rekursion am Laufen zu halten. Beim Aufruf TEST(K, K', s) werden die folgenden Informationen in den Keller gepusht:

- die aktuellen Eingabeparameter $K, K' \in \mathcal{K}_n$ und $s \in \{1, \ldots, cS(n)\}$
- den aktuellen Wert der lokalen „Laufvariable“ $K'' \in \mathcal{K}_n$ (von der wir annehmen, dass sie die Konfigurationen aus $\mathcal{K}_n$ in lexicographischer Ordnung durchläuft)
- die Boolesche Variable SUCCESS (initialisiert auf FALSE) zur Speicherung des Ergebniswertes des ersten rekursiven Aufrufs TEST$(K, K'', s-1)$

Wir bezeichnen das Bandsegment mit diesen Informationen mit FRAME(K, K', s). Offensichtlich reichen $O(S(n))$ Zellen für einen FRAME aus.
Zu jeder festen Zwischenkonfiguration $K'' \in \mathcal{K}_n$ verfährt M' wie folgt:

1. Der erste rekursive Aufruf TEST$(K, K'', s-1)$ wird getätigt (d.h., FRAME$(K, K'', s-1)$ wird angelegt).
2. Bei erfolgloser Rückkehr aus Aufruf 1 wird K'' lexicographisch inkrementiert (falls möglich) und zu Schritt 1 zurück gegangen. Falls jedoch K'' bereits die lexicographisch letzte Konfiguration in $\mathcal{K}_n$ war (keine weitere Inkrementierung möglich) wird der Aufruf TEST(K, K', s) erfolglos terminiert, d.h., M' löscht FRAME(K, K', s) und kehrt erfolglos in den darunter liegenden FRAME (sofern vorhanden) zurück.
3. Bei erfolgreicher Rückkehr aus Aufruf 1 wird SUCCESS auf TRUE gesetzt und der zweite rekursive Aufruf TEST$(K'', K', s-1)$ getätigt.
4. Bei erfolgloser Rückkehr aus Aufruf 2 wird SUCCESS auf FALSE zurück gesetzt und ansonsten so vorgegangen wie bei der erfolglosen Rückkehr aus Aufruf 1.[13]
5. Bei erfolgreicher Rückkehr aus Aufruf 2 wird der Aufruf TEST(K, K', s) erfolgreich terminiert, d.h., M' löscht FRAME(K, K', s) und kehrt erfolreich in den darunter liegenden FRAME zurück.

Bei dieser Vorgehensweise merkt sich M' in der endlichen Kontrolle, ob sie sich in einer PUSH- (neuer rekursiver Aufruf, neuer FRAME) oder POP-Phase (Rückkehr aus einem Aufruf, FRAME löschen) befindet. In einer POP-Phase merkt sie sich zudem, ob sie aus dem Aufruf des zuletzt gelöschten FRAME's erfolgreich oder erfolglos zurück kehrt.
Wenn M' irgendwann ihren Keller vollständig geleert hat, wurde gerade der FRAME des initialen Aufrufes TEST$(K_0(w), K_+, cS(n))$ gelöscht. Zu diesem Zeitpunkt weiß M' in der endlichen Kontrolle, ob dieser Aufruf erfolgreich war. Sie akzeptiert die Eingabe w gdw dies der Fall ist.

[13] Wie merkt M', ob sie aus dem 1. oder 2. rekursiven Aufruf für Zwischenkonfiguration K'' zurück kehrt? Antwort: am Booleschen Wert von SUCCESS. TRUE zeigt an, dass die Rückkehr aus die 2. Aufruf erfolgte.

Aus unserer Diskussion geht hervor, dass $w \in L$ von M' korrekt getestet und dass die Platzschranke S^2 respektiert wird. **qed.**

Satz 10.5 *Es sei $S(n) \geq \log n$ eine platzkonstruierbare Funktion. Dann gilt*

$$NSpace(S) \subseteq DTime(2^{O(S)}) \ .$$

Beweis Es sei $L \in NSpace(S)$ und M eine $S(n)$-platzbeschränkte NTM mit $L = L_M$. Konstante $c > 0$, $K_0(w)$, K_+ und $\mathcal{K}_n$ mit $|\mathcal{K}_n| \leq 2^{cS(n)}$ seien gewählt wie im Beweis zum Satz von Savitch. Wir betrachten diesmal $\mathcal{K}_n$ als die Knotenmenge eines Digraphen $G_n(M)$, wobei wir eine gerichtete Kante von K nach K' ziehen, wenn K' eine direkte Folgekonfiguration von K bezüglich der NTM M ist. Offensichtlich gilt

$$w \in L_M \Leftrightarrow \text{Es gibt in } G_n(M) \text{ einen Pfad von } K_0(w) \text{ nach } K_+.$$

Der Digraph $G_n(M)$ hat maximal $2^{cS(n)}$ Knoten und maximal $(2^{cS(n)})^2 = 2^{2cS(n)}$ Kanten. Erreichbarkeitsprobleme, wo nach Pfadverbindungen zwischen zwei Knoten gefragt wird, sind mit einfachen Graphexplorationstechniken zeiteffizient lösbar. Eine DTM benötigt bei Digraphen der Größe N hierfür maximal $O(N^k)$ Schritten (für eine geeignete Konstante k). Wegen $(2^{2cS(n)})^k = 2^{2kcS(n)}$ ergibt sich die Zeitschranke $2^{O(S)}$. **qed.**

Der im Beweis verwendete Digraph $G_n(M)$ heißt der *Konfigurationsgraph* von M.

10.3 Abschluss unter Komplement

Deterministische Zeitkomplexitätsklassen sind trivialerweise abgeschlossen unter Komplement:

Satz 10.6 *$DTime(T(n)) = co\text{-}DTime(T(n))$.*

Beweis Vertauschen von Endzuständen und Nicht-Endzuständen macht aus einem $T(n)$-zeitbeschränkten Akzeptor von L einen $T(n)$-zeitbeschränkten Akzeptor von $\bar{L} = \Sigma^* \setminus L$. **qed.**

Beachte, dass die entsprechende Aussage für *nichtdeterministische* Zeitkomplexitätsklassen vermutlich nicht gilt. Zum Beispiel gilt ja vermutlich $NP \neq$ co-NP.

Wie verhalten sich Platzkomplexitätsklassen bezüglich Komplementbildung. Die folgenden Resultate liefern eine Antwort.

Satz 10.7 *Für jede platzkonstruierbare Funktion $S(n) \geq \log n$ gilt:*
$DSpace(S(n)) = co\text{-}DSpace(S(n))$.

Beweis Wenn ein $S(n)$-platzbeschränkter Akzeptor von L auf jeder Eingabe nach endlich vielen Schritten stoppt, dann können wir erneut mit dem Vertauschen von Endzuständen und Nicht-Endzuständen argumentieren. Durch etwaige Endlosschleifen ensteht aber ein Zusatzproblem. Andererseits wissen wir bereits, dass ein $S(n)$-platzbeschränkter Akzeptor M nur $2^{O(S(n))}$ Konfigurationen hat. Es gibt daher eine Konstante k mit der Eigenschaft: nach $2^{kS(n)}$ Schritten muss M sich in einer Endlosschleife befinden. Wir können die Platzkonstruierbarkeit von $S(n)$ und damit auch $kS(n)$ benutzen, um einen Binärzähler mit $10^{kS(n)}$, also der Binärdarstellung von $2^{kS(n)}$, zu installieren. Dieser Binärzähler kann als eine Art „Uhr" rückwärts gezählt werden. Wenn die Uhr auf 0 steht, brechen wir die Rechnung nichtakzeptierend ab. **qed.**

- Satz 10.7 gilt sogar *ohne* die Voraussetzung der Platzkonstruierbarkeit von $S(n)$. Den Nachweis hierfür empfehlen wir als **Übung**. **Übg.**
- Weiterhin gilt für jede platzkonstruierbare Funktion $S(n) \geq \log n$: $\mathit{NSpace}(S(n)) =$ co-$\mathit{NSpace}(S(n))$. Den Nachweis hierfür empfehlen wir ebenfalls als **Übung**. **Übg.**

10.4 Die Platz-Zeit-Hierarchie

Wir fügen den uns inzwischen bekannten Beziehungen zwischen Komplexitätsklassen noch die folgenden hinzu:

$$\mathit{DTime}(R) \subseteq \mathit{DSpace}(R) \text{ und } \mathit{NTime}(R) \subseteq \mathit{NSpace}(R) \ .$$

Diese ergeben sich einfach daraus, dass jeder Besuch einer noch unbesuchten Zelle mindestens einen Rechenschritt erfordert. Es ergibt sich nun die folgende Situation für eine platzkonstruierbare Ressourcenschranke R:

$$\begin{aligned} \mathit{DSpace}(R) \subseteq \mathit{NSpace}(R) \subseteq \mathit{DTime}\left(2^{O(R)}\right) &\subseteq \mathit{NTime}\left(2^{O(R)}\right) \\ &\subseteq \mathit{NSpace}\left(2^{O(R)}\right) = \mathit{DSpace}\left(2^{O(R)}\right) \ . \end{aligned}$$

Bei der letzten Gleichung wurde der Satz von Savitch benutzt.

Wir erinnern an die Komplexitätsklassen $L = \mathit{DSpace}(\log n)$, $NL = \mathit{NSpace}(\log n)$, P, NP und PSpace. Mit $R(n) = \log n$ ergibt sich (unter Beachtung von $2^{O(\log n)} = n^{O(1)}$) die

Folgerung 10.8 $L \subseteq NL \subseteq P \subseteq NP \subseteq \mathit{PSpace}$.