Vorlesungsskriptum

Theoretische Informatik 1

Sommersemester 2012

Institut für Grundlagen der Informationsverarbeitung
Technische Universität Graz

Autoren:
Florian Burgstaller, Andreas Derler, Armin Eibl, Michaela Heschl, Dominik Hirner, Jakob Hohenberger, Sebastian Hrauda, Philipp Kober, Anton Lipp, Lukas Prokop, Bernd Prünster, Mattias Rauter, Gabriel Schanner, Matthias Vierthaler, Dominik Ziegler

Vortragender:
David Kappel

4. August 2012

Vorwort

Dieses Skriptum wurde von Hörern der Vorlesung *Theoretische Informatik 1* des Sommersemesters 2012 an der Technischen Universität Graz erstellt. Es soll als Unterstützung dienen und ersetzt weder den Besuch der Vorlesung, noch das Lesen der empfohlenen Fachliteratur. Es gibt einen Überblick über grundlegende Themen der theoretischen Informatik, Komplexitätstheorie, Analyse von Algorithmen und Randomisierung. Im *Grundlagen* Kapitel werden Konzepte aus der Mengenlehre und der Graphentheorie, die für das Verständnis der Vorlesung notwendig sind zusammengefasst. Das Skriptum wurde nach bestem Wissen und Gewissen erstellt, trotzdem sind Fehler nicht ausgeschlossen und es sollte nicht direkt als Referenz verwendet werden, sondern immer die entsprechende Literatur auf die verwiesen wird. Wenn Sie Fragen oder Anregungen haben, oder sich an der Verbesserung dieses Skriptum beteiligen wollen, wenden Sie sich bitte an *ti1@igi.tugraz.at*.

Inhaltsverzeichnis

0 Grundlagen

Armin Eibl, Anton Lipp, Bernd Prünster

0.1 Mengen

Bei einer *Menge* handelt es sich um eine Zusammenfassung von *Elementen*. Ein Element kann eine Zahl, ein Buchstabe, eine Zeichenkette, eine andere Menge, usw. sein. Es gibt auch eine Menge die keine Elemente enthält, diese Menge wird als *leere Menge* bezeichnet. Es gibt verschiedene Arten eine Menge zu formulieren. Die am Meisten verbreitete Art ist die Menge in geschwungen Klammern zu schreiben und die Elemente mit einem Beistrich zu trennen. Die Symbole $\in$ und $\notin$ beschreiben ob ein Element teil einer Menge ist oder nicht. Bei der Betrachtung einer Menge geht es ausschließlich um die Frage, welche Elemente enthalten sind. Die Reihenfolge spielt keine Rolle.

Beispiel 0.1

Die leere Menge: $\emptyset$ oder auch $\{\}$
Für die Menge $M = \{5, 7, 33\}$ gilt $5 \in M$ und $8 \notin M$.

Da es jedoch oft mühsam ist alle Elemente einer Menge aufzuschreiben bzw. da es Mengen mit unendlich vielen Elementen gibt, kann man eine Menge auch durch Regeln definieren, die für alle Elemente dieser Menge gültig sind.

Beispiel 0.2

Die Menge aller positiven ganzen Zahlen, die durch 3 teilbar sind:
$M = \{x \in \mathbb{N} | \exists y \in \mathbb{N} : y = \frac{x}{3}\}$

0.1.1 Teilmenge

Definition 0.1 (Teilmenge)**.** *Eine Menge A ist eine Teilmenge einer Menge B, wenn jedes Element von A auch ein Element von B ist. B ist dann die Obermenge von A.*

$$A \subseteq B \Leftrightarrow \forall x \in A : x \in B$$

Es gibt außerdem gibt es den Begriff der echten Teilmenge.

Definition 0.2 (Echte Teilmenge)**.** *A ist eine echte Teilmenge von B, wenn alle Elemente von A in B enthalten sind, B jedoch mindestens ein Element enthält, dass nicht in A enthalten ist.*

$$A \subset B \Leftrightarrow \forall x \in A \land x \in B \land \exists y \in B : y \notin A$$

0.1.2 Schnittmenge

Definition 0.3 (Schnittmenge)**.** *Die Schnittmenge von zwei Mengen A und B enthält nur Elemente die sowohl in der Menge A als auch in der Menge B enthalten sind.*

$$A \cap B \Leftrightarrow \{x : x \in A \land x \in B\}$$

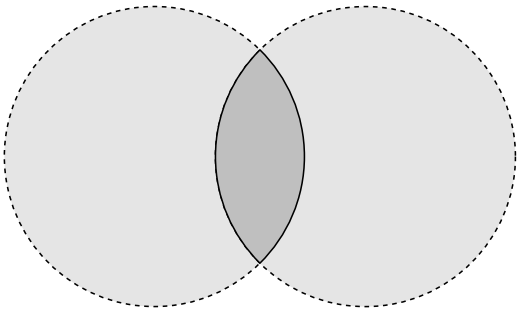

Abbildung 0.1: Schnittmenge

0.1.3 Vereinigungsmenge

Definition 0.4 (Vereinigungsmenge)**.** *Die Vereinigungsmenge von zwei Mengen A und B enthält alle Elemente der Menge A und der Menge B.*

$$A \cup B \Leftrightarrow \{x : x \in A \lor x \in B\}$$

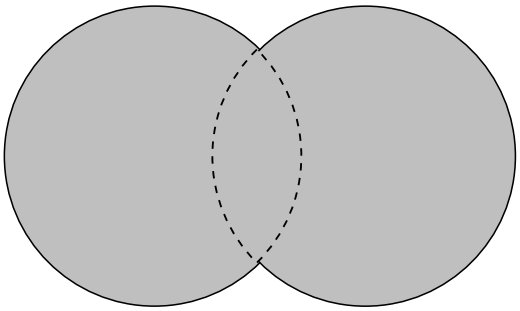

Abbildung 0.2: Vereinigungsmenge

0.1.4 Differenzmenge

Definition 0.5 (Differenzmenge)**.** *Die Differenzmenge von zwei Mengen A zu B enthält alle Elemente die in der Menge A und nicht in der Menge Benthalten sind.*

$$A \backslash B \Leftrightarrow \{x : x \in A \land x \notin B\}$$

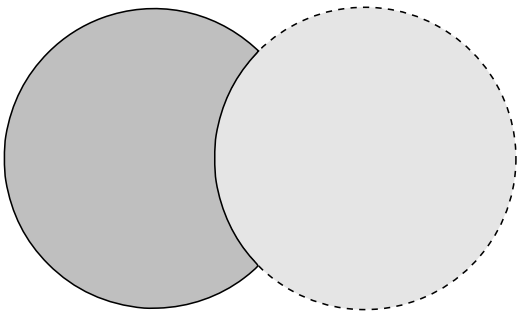

Abbildung 0.3: Differenzmenge

0.1.5 Potenzmenge

Definition 0.6 (Potenzmenge)**.** *Die Potenzmenge einer Menge A ist die Menge aller Teilmengen von A.*

$$\mathcal{P}(A) := \{U : U \subseteq A\}$$

Beispiel 0.3

$A = \{0, 1\}$
Potenzmenge $= \{\emptyset, \{0\}, \{1\}, \{0, 1\}\}$

0.1.6 Abzählbarkeit von Mengen

Eine Menge wird abzählbar unendlich genannt, wenn sie unendlich ist und jedes Element der Menge ein Element der Natürlichen Zahlen zugeordnet werden kann.

Beispiel 0.4

Primzahlen als Teilmenge von $\mathbb{N}$
Paare von Natürlichen Zahlen (Cantorsche Paarungsfunktion)

Als Gegenbeispiel hier ein Beispiel für eine überabzählbar unendliche Menge:

Beispiel 0.5

Potenzmenge einer abzählbar unendlichen Menge.

Die Überabzählbarkeit lässt sich mit dem Hilberts Hotel Paradoxon veranschaulichen.
Man nehme an, dass ein Hotel mit unendlich vielen Zimmern existiert welches voll ist. Um nun Platz für einen Gast zu machen, zieht jeder Gast in das nächste Zimmer. Um Platz für unendlich viele Gäste zu machen, multiplizieren alle Gäste ihre Zimmernummer mit 2 und ziehen in das Zimmer mit dieser Nummer. So werden alle ungeraden Zimmer frei.

0.2 Formale Sprache

Eine *formale Sprache* besteht aus einer *Zeichenkette* die aus einem bestimmten *Alphabet* zusammen gesetzt ist. Ein Alphabet besteht aus einer endliche Menge von Symbolen. Ein Alphabet wird oft mit dem griechischen Buchstaben Σ angegeben. Ein Wort einer Sprache ist also ein Element der Potenzmenge von Σ.

Beispiel 0.6

$\Sigma_1 = \{0, 1\}$
$\Sigma_2 = \{0, 1, a, b, c\}$

Eine Zeichenkette aus einem Alphabet ist eine endliche Sequenz von Symbolen aus dem Alphabet.

Beispiel 0.7

Alphabet $\Sigma_1 = \{0, 1\}$
Dann ist die Zeichenkette 101101000100 ein Wort, dass aus dem Alphabet Σ_1 gebildet werden kann.

Die Länge der Zeichenkette ist die Anzahl der Symbole. Die Zeichenkette mit der Länge 0 ist die leere Zeichenkette. Die Wörter der Sprache werden oft noch durch Zusatzbedingungen eingeschränkt.

Beispiel 0.8

Die Sprache L sei die Menge aller Wörter über dem Alphabet $\Sigma = \{a, b\}$ die nur Wörter der Länge 2 enthält.

$L = \{w \in \Sigma \quad | \quad |w| = 2\}$

$|w|$ bezeichnet die Länge des Wortes. L ist also die endliche Menge $L = \{aa, ab, ba, bb\}$.

0.2.1 Formale Sprachen in der theoretischen Informatik

Die Sprachen die in der theoretischen Informatik auftreten werden meistens durch Ersetzungsverfahren definiert. Von diesen Verfahren gibt es verschiedene Typen.

0.2.1.1 Semi-Thue Systeme

Ein Semi-Thue System besteht aus dem Alphabet und einer Menge von Substitutionen, die man als $a \to b$ schreibt, a und b sind Wörter des Alphabets. Das System wird als Paar (A, S) definiert, wobei A das Alphabet und S die Substitutionen sind.

0.2.1.2 Chomsky-Grammatiken

Die Chomsky-Grammatik ist eine Hierachie von Klassen formaler Grammatiken, die formale Sprachen erzeugt. Die Grammatik wird als ein 4-Tubel (V_n, V_t, S, ϕ) definiert.
V_n ... endliche Menge von Nonterminalen
T_t ... endliche Menge von Terminalen
$S \in V_n$ ... Startsymbol
ϕ endliche Menge von Produktionsregeln

Die Chromsky Grammatik wird in vier Typen eingeteilt.

Definition 0.7 (Allgemeine Grammatiken G).
Keine Restriktionen bezüglich $a \to b$

Definition 0.8 (Kontextsensitive Grammatiken G_{sens}).
$a \to b$
$|a| \leq |b|$

Definition 0.9 (Kontextfreie Grammatiken G_{free}).
$a \to b$
$|a| \leq |b|, a \in Vn$

Definition 0.10 (Reguläre Grammatik G_{reg}).
$a \to b$
$|a| \leq |b|, a \in Vn, b$ *hat Form* aA *oder* a, *mit* $a \in V_t, A \in V_n$

Die Programmiersprache C ist eine formale Sprache. Die Programme in C setzen sich aus den Wörtern zusammen. Die Schlüsselwörter und die Zeichen sind das Alphabet von C, diese in der Definition von C festgelegt sind.

0.3 Analyse von Algorithmen

Die Landau Notation gibt ein asymptotisches Verhalten von Funktionen und Folgen an. In der Informatik wird sie für die Analyse von Algorithmen verwendet. Die Komplexitätstheorie verwendet sie, um Probleme miteinander zu vergleichen und gibt an wie schwierig ein Problem zu lösen ist.

0.3.1 $\mathcal{O}$-Notation

Die $\mathcal{O}$-Notation gibt die asymptotisch obere Schranke an.

Definition 0.11 ($\mathcal{O}$-Notation). *Alle Funktionen* f, *deren asymptotisches Wachstum in der Größenordnung durch* $g(n)$ *beschränkt ist:*

$$\mathcal{O}(g) = \{f : \mathbb{N} \mapsto \mathbb{R} : \exists c, n_0 : \forall n > n_0 : \ f(n) \leq c \cdot g(n)\}$$

0.3.2 Ω-Notation

Die Ω-Notation gibt die asymptotisch untere Schranke an.

Definition 0.12 (Ω-Notation). *Alle Funktionen* f, *deren asymptotisches Wachstum in der Größenordnung durch* $g(n)$ *beschränkt ist:*

$$\Omega(g) = \{f : \mathbb{N} \mapsto \mathbb{R} : \exists c > 0, n_0 : \forall n > n_0 : f(n) \geq c \cdot g(n)\}$$

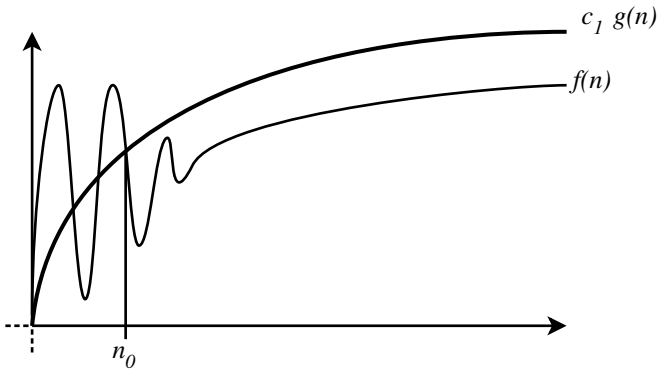

Abbildung 0.4: Asymptotisch obere Schranke

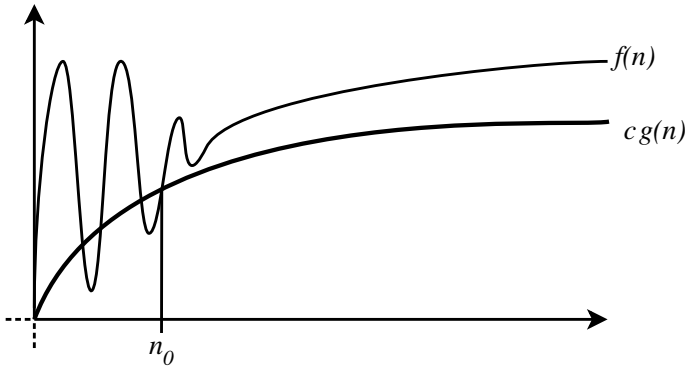

Abbildung 0.5: Asymptotisch untere Schranke

0.3.3 Θ-Notation

Die Θ-Notation gibt die asymptotisch exakte Schranke an.

Definition 0.13 (Θ-Notation). *Alle Funktionen f, deren asymptotisches Wachstum genau in der Größenordnung von $g(n)$ liegt:*

$$\Theta(g) = \{f : \mathbb{N} \mapsto \mathbb{R} : \exists c_1, c_2 c, n_0 : \forall n > n_0 :\ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$

0.3.4 o-Notation

Die o-Notation gibt an, das f asymptotisch vernachlässigbar gegenüber g ist. Man sagt auch g wächst »wirklich« schneller als f.

Definition 0.14 (o-Notation). *Alle Funktionen f, deren asymptotisches Wachstum genau in der Größenordnung von $g(n)$ liegt:*

$$o(g) = \{f : \mathbb{N} \mapsto \mathbb{R} : \forall c > 0, \exists n_0, \forall n > n_0 : f(n) \leq c \cdot g(n)\}$$

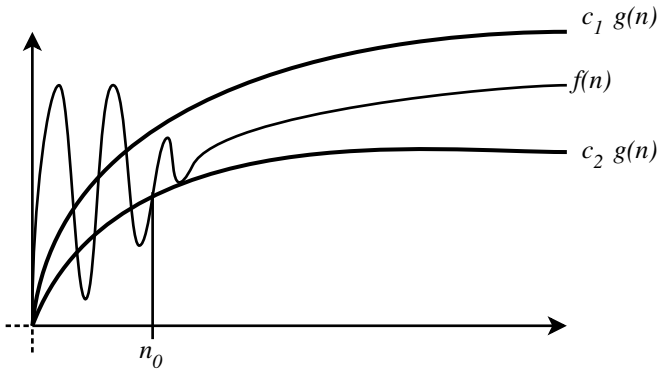

Abbildung 0.6: Asymptotisch exakte Schranke

0.3.5 Landau $\mathcal{O}$ Beweise

Beispiel 0.9

Beweise: $2^n \in \mathcal{O}(n^n)$
Man nimmt die Definition von $\mathcal{O}$ und wählt $n_0 = 2$, $c = 1$.

$$c \cdot n^n = n^n \geq 2^n, \quad \forall n > n_0$$

Alternative: $f(n) \in \mathcal{O}(g(n)) \quad \Leftrightarrow \quad 0 \leq \limsup_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| < \infty$

$$\begin{aligned}
\lim_{n\to\infty} \frac{2^n}{n^n} &= \lim_{n\to\infty} \frac{2^n}{2^{n\log n}} \\
&= \lim_{n\to\infty} 2^{(n-n\log n)} \\
&= 2^{\lim_{n\to\infty} n(1-\log n)} \\
&= 2^{-\overbrace{\lim_{n\to\infty} n\,(\log n - 1)}^{\infty}} = 0
\end{aligned}$$

Beispiel 0.10

Beweise: $n^{\log n} \in \mathcal{O}(2^n)$

$$\begin{aligned}
\lim_{n\to\infty} \frac{n^{\log n}}{2^n} &= \lim_{n\to\infty} \frac{2^{(\log n)^2}}{2^n} \\
&= 2^{\lim_{n\to\infty} (\log n)^2 - n}
\end{aligned}$$

Substitution $x = \log n$

$$\lim_{n\to\infty} \frac{n^{\log n}}{2^n} = 2^{\overbrace{\lim_{x\to\infty} x^2 - 2^x}^{-\infty}} = 0$$

0.3.6 Schleifeninvariante

Um die Korrektheit von Algorithmen, die Schleifen verwenden zu beweisen, kann eine Schleifeninvariante aufgestellt werden. Dies ist eine Bedingung, die bei jedem Durchlauf der Schleife an einem gewissen Punkt im Schleifenkörper erfüllt ist. Das ein Algorithmus eine bestimmte Schleifeninvariante besitzt kann meist einfach durch vollständige Induktion gezeigt werden. Durch einsetzten der Abbruchbedingung kann dann die Richtigkeit des Algorithmus gezeigt werden.

Beispiel 0.11

Algorithmus: Multiplikation durch Addition

Es soll gezeigt werden, dass der Algorithmus korrekt multipliziert, daher dass $z = a \cdot b$ beim Schleifenabbruch gilt.

$x_1 := a$	Schleifeninvariante p(i)
$y_1 := b$	
$z_1 := 0$	$(x \cdot y) + z = a \cdot b$
while $x > 0$ *do*	
	Schleifenabbruch bei $x = 0$
$z_{i+1} := z_i + y$	
$x_{i+1} := x_i - 1$	$z = a \cdot b$
`end`	

0.4 Graphen

Ein *Graph* ist definiert als ein Tupel (V, E) wobei V eine Menge von *Knoten* (»vertices«) und E eine Menge von Kanten (»edges«) bezeichnet. Knoten werden mit arabischen Ziffern bezeichnet.
Eine *Kante* E ist ein Tupel (v_1, v_2) von Knoten $v_1, v_2 \in V$, wobei in *gerichteten Graphen* v_1 den ausgehenden und v_2 den eingehenden Knoten repräsentiert. In *ungerichteten Graphen* ist es unerheblich, welcher Knoten einer Kante der Eingangs- und welcher der Ausgangsknoten ist. Formal wird der Relationsoperator verwendet, um anzuzeigen, dass eine Kante von v_1 nach v_2 existiert: $v_1 \underset{V}{\longrightarrow} v_2 \Leftrightarrow (v_1, v_2) \in E$

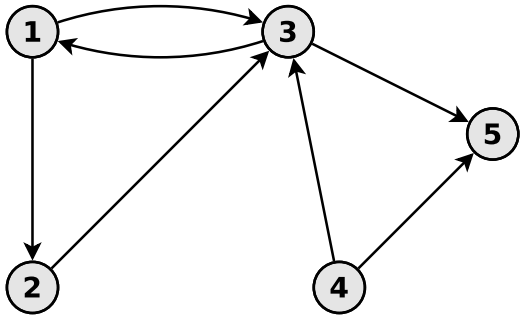

Abbildung 0.7: gerichteter Graph G

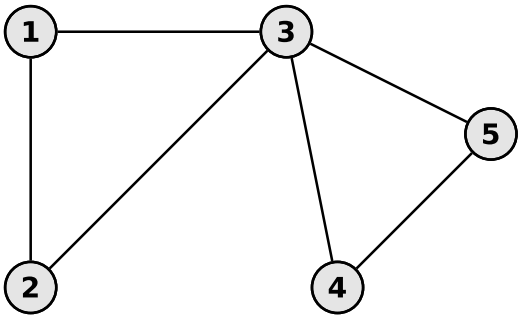

Abbildung 0.8: ungerichteter Graph

Beispiel 0.12

$G = (V, E)$
$V = \{1, 2, 3, 4, 5\}$
$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 5), (4, 3), (4, 5)\}$
Anzahl Knoten $= |V|$
Anzahl Kanten $= |E|$
siehe Abbildung 0.7

0.4.1 Pfad

Eine Folge von aufeinanderfolgenden Knoten $v_1 \dots v_n$, welche durch Kanten miteinander verbunden sind, werden als Pfad bezeichnet. Wenn $v_1 = v_n$ dann bezeichnet man einen solchen Pfad als Zyklus. Im Gegensatz zu ungerichteten Graphen sind in gerichteten Graphen nur Pfade entlang der Kantenrichtung möglich.

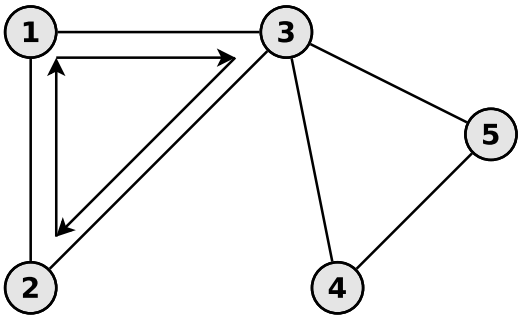

Abbildung 0.9: Zyklus von 1 nach 1

0.4.2 Transitive Hülle

Ein formales Werkzeug um Pfade in Graphen zu beschreiben ist die *transitive Hülle*. Diese umfasst die Menge aller Pfade von v_1 nach v_2. Die *reflexiv transitive Hülle* entspricht der transitiven Hülle unter Einbezug von Reflexivität.
Notation:

transitive Hülle $v_1 \xrightarrow[V]{+} v_2 \Leftrightarrow v_1 \xrightarrow[V]{} v_2 \vee \exists h \in V : v_1 \xrightarrow[V]{} h \xrightarrow[V]{+} v_2$

reflexiv transitive Hülle $v_1 \xrightarrow[V]{*} v_2 \Leftrightarrow v_1 = v_2 \vee v_1 \xrightarrow[V]{+} v_2$

Allgemein ist die transitive Hülle $\mathcal{R}^+$ einer Relation $\mathcal{R}$ definiert als die kleinste transitive Relation, die $\mathcal{R}$ enthält.[1]

0.4.3 Multigraph

Als *Multigraphen* bezeichnet man Graphen mit mindestens einer Schleife oder Mehrfachkante.[2] Sind zwei Knoten mit mehr als einer Kante verbunden, werden diese Kanten zusammengefasst als *Mehrfachkante* bezeichnet. Zu beachten gilt, dass es in gerichteten Graphen mehr als eine Kante zwischen zwei Knoten v_1 und v_2 geben kann: (v_1, v_2) und (v_2, v_1). Derartige Kanten werden nicht als Mehrfachkanten angesehen, da es sich um Kanten unterschiedlicher Richtung handelt.
Eine *Schleife* entsteht, wenn ein Knoten mit sich selbst verbunden wird. Ein *einfacher Graph* ist ein Graph ohne Schleifen und Mehrfachkanten.

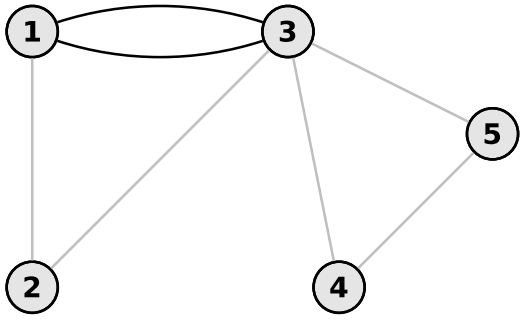

Abbildung 0.10: Mehrfachkante

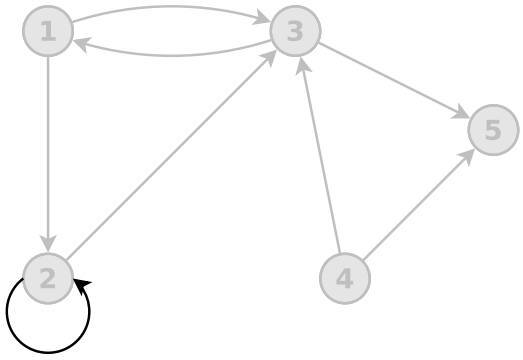

Abbildung 0.11: Schleife

0.4.4 Clique

Eine *Clique* U ist definiert als Teilgraph in einem ungerichteten Graphen $G = (V, E)$ in dem je zwei beliebige Knoten v_1 und v_2 über eine Kante miteinander verbunden sind. Eine Clique U wird als *maximal* bezeichnet, wenn diese durch Miteinbeziehen eines weiteren Knotens nicht erweitert werden kann. Gibt es in einem Graphen keine Clique, die mehr Knoten als U enthält, wird U als *größte Clique* bezeichnet. Die *Cliquenzahl* $\omega(G)$ ist definiert als Knotenanzahl der größten Clique.

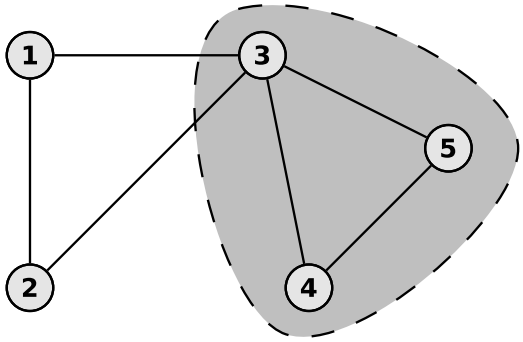

Abbildung 0.12: Clique der Größe 3

[1] Steger, *Diskrete Strukturen*, S. 96.
[2] vgl. ebd., S. 59.

0.4.5 Isolierte Knoten

Zu beachten gilt, dass nicht jeder Knoten in einem Graph über Kanten mit anderen Knoten verbunden sein muss. Derartige Knoten werden als *isolierte Knoten* bezeichnet.

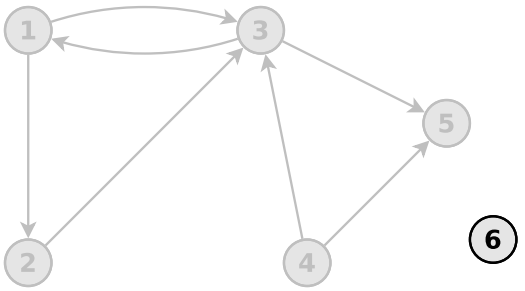

Abbildung 0.13: Graph mit isolierten Knoten

0.4.6 Zyklischer Graph

Wenn in einem Graphen ein Zyklus enthalten ist, wird dieser als *zyklicher Graph* bezeichnet.

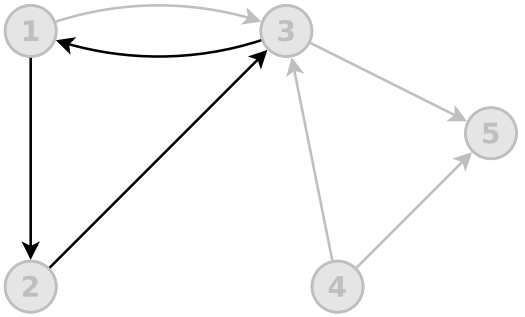

Abbildung 0.14: Graph mit einem Zyklus

0.4.7 Grad eines Knotens

Der *Grad* eines Knotens ist definiert als die Anzahl der Kanten dieses Knotens. Bei einem gerichteten Graphen unterscheidet man zwischen dem *Eingangsgrad* und dem *Ausgangsgrad*. Der Eingangsgrad ist die Anzahl der eingehenden Kanten und der Ausgangsgrad die Anzahl der ausgehenden Kanten eines Kontens.
Einen Knoten ohne Eingangskanten nennt man *Quelle*, einen Knoten ohne Ausgangskanten nennt man *Senke*.

Eingangsgrad des Knotens 1: $g^-_{G'}(1) = 1$
Ausgangssgrad des Knotens 1: $g^+_{G'}(1) = 2$

Wäre der Graph G' ungerichtet würden sich folgende Grade ergeben:
Grad des Knotens 1: $g_{G'}(1) = 3$
»Der Grad des Knotens 1 des Graphen G' hat einen Grad von 3«
Grad des Knotens 2: $g_{G'}(2) = 4$
(Schleifen werden doppelt gezählt)
Grad des Knotens 5: $g^-_{G'}(5) = 2$

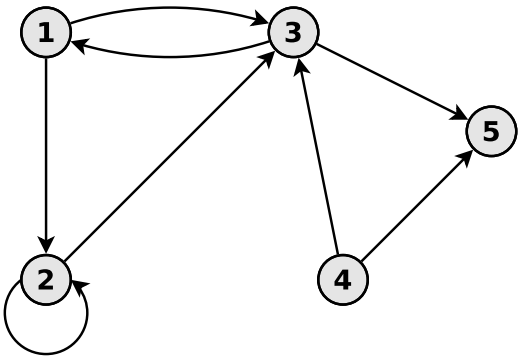

Abbildung 0.15: Graph G'

Anmerkung: Nachfolgend handelt es sich, wenn nicht anders erwähnt, immer um gerichtete Graphen ohne Multikanten.

1 Intuitive Berechenbarkeit

Bernd Prünster

Jeder hat eine gewisse Vorstellung davon, was prinzipiell berechenbare Probleme sind und ob diese eher *leicht* oder eher *schwer* lösbar sind.
Beispielsweise werden Stundenpläne händisch durch Probieren und mit viel Erfahrung erstellt. Hierbei handelt es sich offenbar um ein *schwieriges* Problem - Sortieren ist im Gegensatz dazu vergleichsweise *einfach*.
Mathematische Beweise sind im Allgemeinen gar nicht berechenbar. Aus diesem Grund dauerte es auch fast 400 Jahre, bis ein Beweis zu Fermats letztem Satz gefunden wurde.
Generell ist es sinnvoll Aussagen über Lösbarkeit und Komplexität von Problemen treffen zu können. Hierzu ist es notwendig formale Begriffe für Algorithmen, Probleme, Berechenbarkeit, Effizienz, ... einzuführen. Diese Begriffe sollen möglichst präzise, konzeptionell möglichst einfach und intuitiv, sowie unabhängig von konkreten Einschränkungen (wie Ausführunggeschwindigkeit, Hardware, Programmiersprache, ...) sein. Generell können Probleme wie in Abb. 1.1 dargestellt eingeteilt werden.

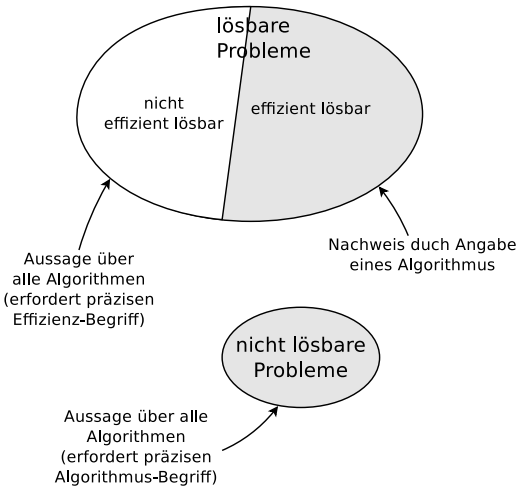

Abbildung 1.1: Intuitive Berechenbarkeit

Der Nachweis der Lösbarkeit gestaltet sich einfach, da es genügt einen Algorithmus anzugeben, welcher das Problem löst.

Um ein Problem als *unlösbar* definieren zu können, muss bewiesen werden, dass es *keinen* Algorithmus geben kann, der dieses Problem löst. Folglich muss der Algorithmusbegriff präzise definiert werden, da eine Aussage über *alle* Algorithmen getroffen wird.

Gleiches gilt, wenn man die Frage nach der Effizienz stellt: Der Nachweis der *effizienten Lösbare* erfolgt duch Angabe *eines* effizienten Algorithmus'. Um ein Problem als *nicht effizient lösbar* zu klassifizieren, muss bewiesen werden, dass es *keinen* Algorithmus gibt, der dieses Problem effizient löst. Folglich wird eine Aussage über *alle* Algorithemn getroffen. Hierzu ist ein wohl definierter Begriff der Effizienz notwendig.

1.1 Graph-Erreichbarkeit

Das Erreichbarkeitsproblem in Graphen (`REACH` oder `STCON`, `PATH`) ist ein Basisproblem der Theoretischen Informatik. Es handelt sich dabei um ein Entscheidungsproblem, welches die Frage behandelt, ob es in einem Graphen $G = (V, E)$ mit $n = |V|$ Knoten einen Pfad vom Knoten s zum Knoten t gibt. Hierbei handelt es sich um ein Basisproblem der Theoretischen Informatik, da jedes lösbare Entscheidungsproblem als gerichteter Graph dargestellt werden kann. Allgemein betrachtet gibt es verschiedene Antworten auf das Graphenerreichbarkeitsproblem:

- Weg existiert/existiert nicht (»ja«/»nein«)
- Liste alle Wege auf
- Welcher ist der kürzeste/längste/... Weg

In der Komplexitätstheorie werden nur Entscheidungsprobleme (»ja«/»nein«) betrachtet, da die Aussage unabhängig von der Kodierung der Antwort sein soll und da sich alle relevanten Probleme in Entscheidungsprobleme überführen lassen.
Eine Lösungsmöglichkeit für das Graphenerreichbarkeitsproblem ist der nachfolgend beschriebene `Search`-Algorithmus:

1. Starte bei Knoten s.
2. Iteriere rekursiv über alle ausgehenden Kanten.
3. Markiere alle besuchten Knoten.
4. Am Ende antworte mit »ja«, wenn Knoten t markiert ist, sonst »nein«.

Pseudocode

$S = \{s\}$

`markiere Knoten` s

`solange` $S \neq \{\}$:

    `wähle ein` $i \in S$ `und entferne` i `aus` S

    $\forall$ `Kanten` $(i, j) \in E$, `wenn` j `nicht markiert:`

        `füge` j `zu` S `hinzu und markiere` j

`Wenn Knoten` t `markiert ist, antworte mit »ja«, sonst mit »nein«`

1.1.1 Ressourcenbedarf

Zeitbedarf: Im schlimmsten Fall muss jede Kante einmal besucht werden
$\Rightarrow T(n) = \mathcal{O}(n^2)$ ($n = |V|$, $|E| = \mathcal{O}(n^2)$)
Speicherplatzbedarf: Im schlimmsten Fall wird jeder Knoten markiert
$\Rightarrow S(n) = \mathcal{O}(n)$

Beispiel 1.1

Gerichteter Graph $G = (V, E)$
$V = \{1, 2, 3, 4, 5\}$
$E = \{(1,2), (1,3), (2,3), (3,1), (3,5), (4,3), (4,5)\}$

Frage: Ist Knoten 5 ausgehend von Knoten 1 erreichbar?

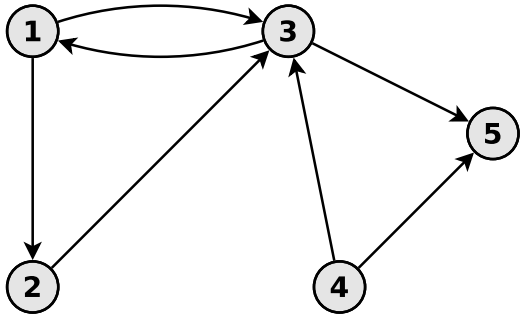

Abbildung 1.2: gerichteter Graph $G = (V, E)$

i	$S_i \subseteq V$	mark. Knoten $M_i \subseteq V$	unmark. Nachfolger X	nächster Schritt
0	$S_0 = \{1\}$	$M_0 = \{1\}$	$X_0 = \{2, 3\}$	wähle 1
1	$S_1 = \{2, 3\}$	$M_1 = \{1, 2, 3\}$	$X_1 = \{\}$	wähle 2
2	$S_2 = \{3\}$	$M_2 = \{1, 2, 3\}$	$X_2 = \{5\}$	wähle 3
3	$S_3 = \{5\}$	$M_3 = \{1, 2, 3, 5\}$	$X_3 = \{\}$	wähle 5
4	$S_4 = \{\}$	$M_4 = \{1, 2, 3, \mathbf{5}\}$	$X_4 = \{\}$	**Antwort:** »ja«

1.1.2 Formaler Beweis auf Korrektheit

In jedem Durchgang wird ein $s \in S_i$ ausgewählt und markiert:

$$X = \{t \in V \setminus M_i \ : \ s \to t\}$$
$$S_{i+1} = S_i \setminus \{s\} \cup X$$
$$M_{i+1} = M_i \cup X$$

Die Basis des nachfolgenden Beweises durch vollständige Induktion basiert auf der Tatsache, dass sich das Problem `REACH` als Frage nach der Existenz einer reflexiv transitiven Hülle formulieren lässt (siehe Kapitel 0.4.2). Da die Nummerierung der Knoten willkürlich ist, kann ohne Einschränkung der Allgemeinheit $s = 1$ und $t = n$ angenommen werden. Im Folgenden wird der Beweis daher für $s = 1$ und $t = n$ durchgeführt.

Annahme Schleifeninvariante von `Search` (wobei $S_i \subseteq M_i$):
$P(i) \equiv 1 \xrightarrow{*} n \iff \left[n \in M_i \ \vee \ \exists h \in S_i : h \xrightarrow[V \setminus M_i]{+} n\right]$

Basis $P(0)$, wobei $S_0 = \{1\}, \ M_0 = \{1\}$
$1 \xrightarrow{*} n \iff \left[n = 1 \ \vee \ 1 \xrightarrow[V \setminus \{1\}]{+} 1\right]$ ✓
(korrekt, da dies der Definition der reflexiv transitiven Hülle entspricht)

Induktionsschritt $P(i) \Rightarrow P(i+1)$:

$$
\begin{aligned}
P(i) \equiv 1 \xrightarrow{*} n &\iff \big[n \in M_i \ \vee\ \exists h \in S_i \ :\ h \xrightarrow[V \setminus M_i]{+} n && \big] && (1.1)\\
&\iff \big[n \in M_i \ \vee\ \exists h \in S_i \setminus \{s\} \ :\ h \xrightarrow[V \setminus M_i]{+} n \ \vee\ s \xrightarrow[V \setminus M_i]{+} n && \big] && (1.2)\\
&\iff \big[n \in M_i \ \vee\ \ldots \ \vee\ \exists \underbrace{t \in V \setminus M_i \ :\ s \to t}_{X} \xrightarrow[V \setminus M_{i+1}]{*} n && \big] && (1.3)\\
&\iff \big[n \in M_i \ \vee\ \ldots \ \vee\ \exists t \in X \ :\ t \xrightarrow[V \setminus \underbrace{M_{i+1}}_{M_i \cup X}]{*} n && \big] && (1.4)\\
&\iff \big[n \in M_i \ \vee\ \ldots \ \vee\ n \in X \ \vee\ \exists t \in X \ :\ t \xrightarrow[V \setminus M_{i+1}]{+} n && \big] && (1.5)\\
&\iff \big[\underbrace{n \in \{M_i \cup X\}}_{n \in M_i \,\vee\, n \in X} \vee \exists h \in \underbrace{S_i \setminus \{s\} \cup X}_{S_{i+1}} \ :\ h \xrightarrow[V \setminus M_{i+1}]{+} n && \big] && (1.6)\\
&\iff \big[n \in \underbrace{M_i \cup X}_{M_{i+1}} \vee \exists h \in S_{i+1} \ :\ h \xrightarrow[V \setminus M_{i+1}]{+} n && \big] && (1.7)\\
&\iff \big[n \in M_{i+1} \ \vee\ \exists h \in S_{i+1} \ :\ h \xrightarrow[V \setminus M_{i+1}]{+} n\big] && && (1.8)\\
&\equiv P(i+1)\ \blacksquare && && (1.9)
\end{aligned}
$$

Erläuterung Bei der in (1.2) vorgenommenen Erweiterung, handelt es sich um eine reine Umformung, da diese nichts am Wahrheitswert der Formel ändert:

$$
\begin{aligned}
&\exists h \in S_i \ :\ h \xrightarrow[V \setminus M_i]{+} n\\
\equiv\ &\exists h \in S_i \setminus \{s\} \ :\ h \xrightarrow[V \setminus M_i]{+} n \ \vee\ s \xrightarrow[V \setminus M_i]{+} n
\end{aligned}
$$

In Schritt (1.3) wird explizit ausformuliert, dass die transitive Hülle $s \xrightarrow[V \setminus M_i]{+} n$ einen Knoten t mit der reflexiv transitiven Hülle $t \xrightarrow[V \setminus M_{i+1}]{*} n$ enthält:

$$
\begin{aligned}
&s \xrightarrow[V \setminus M_i]{+} n\\
\equiv &\exists t \in \ V \setminus M_i \ :\ s \to t \xrightarrow[V \setminus M_{i+1}]{*} n
\end{aligned}
$$

Im Übergang zu Schritt (1.5) wird lediglich die reflexiv transitive Hülle auf die Form der transitive Hülle umgeschrieben:

$$
\begin{aligned}
&s \xrightarrow[V]{*} t \equiv s = t \ \vee\ s \xrightarrow[V]{+} t\\
&\exists t \in X \ :\ t \xrightarrow[V \setminus M_{i+1}]{*} n\\
\equiv &n \in X \ \vee\ \exists t \in X \ :\ t \xrightarrow[V \setminus M_{i+1}]{+} n
\end{aligned}
$$

Schritt (1.6) fasst die Ausdrücke $\exists h \in S_i \setminus \{s\} \ : \ h \xrightarrow[V \setminus M_i]{+} n$ und $\exists t \in X \ : \ t \xrightarrow[V \setminus M_{i+1}]{+} n$ zusammen. Wichtig hierbei sind folgende Eigenschaften der betroffenen Mengen:

1. $\exists t \in S \ \vee \ \exists h \in X \equiv \exists r \in S \cup X$
2. $\{V \setminus M_{i+1}\} \subset \{V \setminus M_i\}$

Unter Berücksichtigung dieser Eigenschaften, wird klar, dass es sich bei den Operationen in Schritt (1.6) um eine gültige Äquivalenzumformung handelt:

$$\begin{aligned} & \exists h \in S_i \setminus \{s\} \ : \ h \xrightarrow[V \setminus M_i]{+} n \ \vee \ \exists t \in X \ : \ t \xrightarrow[V \setminus M_{i+1}]{+} n \\ \equiv & \exists h \in S_i \setminus \{s\} \cup X \ : \ h \xrightarrow[V \setminus M_{i+1}]{+} n \end{aligned}$$

2 Registermaschine

Matthias Vierthaler

Wie im vorigen Kapitel ersichtlich, ist – neben der Frage nach der Lösungsmöglichkeit für Algorithmen – auch der konkrete Begriff *Algorithmus* zu definieren. Zur formalen Beschreibung von Algorithmen kann ein Maschinenmodell verwendet werden. Der Algorithmus ist dann durch ein Programm auf diesem Maschinenmodell gegeben. In Anlehnung an einen modernen Digitalrechner können wir folgende Anforderungen an den Algorithmusbegriff stellen:

- endliche Anzahl an Operationen
- endliche, nummerierte Anweisungsliste (Programm)
- Rechenoperationen: $+ \; - \; * \; \div$
- (Zwischen-)Ergebnisse: $r_0, r_1, \ldots, r_k, \ldots$
- bedingte Verzweigung (JZERO), Sprung (GOTO)
- Konstanten $\#k$, direkte Adressierung r_k
- indirekte Adressierung r_{r_k}
- definierter Haltebefehl: END

Diese Überlegungen können in einem theoretischen Modell, der sogenannten *Registermaschine* (RM), zusammen gefasst werden.[1] Abb. 2.1 zeigt den Aufbau einer RM.

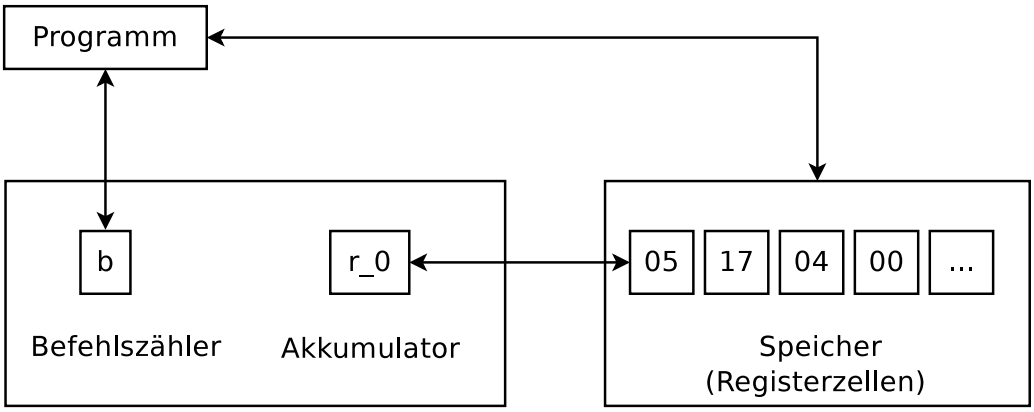

Abbildung 2.1: Registermaschine - Aufbau

[1] Asteroth und Baier, *Theoretische Informatik. Eine Einführung in Berechenbarkeit, Komplexität und formale Sprachen.*

- Das Programm, in weiterer Folge $\mathcal{R}$ genannt, ist eine endliche Anweisungsliste an Befehlen.
- Der Befehlszähler b läuft die Anweisungsliste $\mathcal{R}$ ab. Ohne Sprünge bzw. Verzweigungen wird b stetig inkrementiert.
- Der Akkumulator r_0 wird für alle Berechnung benötigt. Auf ihn werden die meisten Operationen ausgeführt.
- Der Speicher wird in Form von Registern angegeben. Er wird für die Speicherung von End- und Zwischenergebnissen benutzt.

2.1 Befehlssatz

Die Befehle einer Registermaschine sind folgende:

`LOAD` x	$r_0 := v(x)$
`STORE` k	$r_k := r_0$
`STORE` $*k$	$r_{r_k} := r_0$
`ADD` x	$r_0 := r_0 + v(x)$
`SUB` x	$r_0 := \max\{0, r_0 - v(x)\}$
`MULT` x	$r_0 := r_0 * v(x)$
`DIV` x	$r_0 := \lfloor \frac{r_0}{v(x)} \rfloor$
`GOTO k`	$b := k$
`JZERO k`	**if** $r_0 = 0$ **then** $b := k$
`END`	$b := 0$

danach: $b := b + 1$ (außer nach Verzweigungen)

Register:
$\forall i \in \mathbb{N}_0 : r_i \in \mathbb{N}_0$

$b = 0 \equiv$ HALT

Genauere Erläuterungen:

- LOAD und STORE - Befehle werden für das Schreiben und Lesen der Register benutzt (Speicherbefehle)
  - LOAD laden von einem Register in den Akkumulator $r_0 \leftarrow v(x)$
  - STORE laden vom Akkumulator in ein Register $r_0 \rightarrow v(x)$
- ADD/SUB/MULT/DIV benutzt den Operator und führt ihn mit der gewählten Funktion auf den Akkumulator aus
  - verarbeitet werden nur ganzzahlige natürliche Zahlen.
  - damit dieser Bereich nicht verlassen wird verhindert SUB negative Zahlen und DIV rundet ab um rationelle Zahlen zu vermeiden.
- GOTO/JZERO können den Verzweigungspfad verändern und greifen auf den Befehlszähler zu.
  - wird einer dieser Befehle benutzt wird der Befehlszähler nach der Ausführung nicht noch einmal verändert um die Manipulation zu behalten.
- END lässt das Programm terminieren
- Falls $\mathcal{R}$ terminiert, erwarten wir das Ergebnis der Berechnung im Akkumulator r_0.

Die relativ umständliche Umgebung in der alle Operationen bis auf GOTO und END im Akkumulator ausgeführt müssen, wird deshalb verwendet, da in diesem Fall von einer einfachen Beschreibung ausgegangen werden kann. Man kann in einzelnen Schritten sehr einfach auf die sich verändernden Elemente schließen.

Argumente: Damit zwischen Konstanten, direkten und indirekten Registerzugriffen unterschieden werden kann, werden folgende Konventionen getroffen.

x	$v(x)$	Beschreibung
$\#k$	k	der übergebene Wert wird direkt übernommen (Konstante)
k	r_k	das durch die Zahl referenzierte Register addressiert (direkte Adressierung)
$*k$	r_{r_k}	der Inhalt des durch die Zahl referenzierten Registers wird als Registerreferenz benutzt (indirekte Adressierung)

2.2 Formale Beschreibungen

Ein **RM-Programm** $\mathcal{R} = (\pi_1, \ldots, \pi_m)$ ist eine endliche Sequenz von Anweisungen. Das Programm der RM hängt nicht vom Input ab und ist daher bereits eine vollständige Beschreibung für einen Algorithmus. Um die Arbeit der RM während der Ausführung zu beschreiben, ist eine formale Beschreibung der Register notwendig. Eine **Registerbelegung** $R = \{(j_1, r_{j_1}), (j_2, r_{j_2}), \ldots, (j_l, r_{j_l})\}$ ist eine endliche Menge von Register-Wert-Paaren. Alle Register, die nicht in R aufgezählt sind, haben den Wert 0 (Achtung: $\mathcal{R} \neq R$). Die Eingabe für eine RM $\mathcal{R}$ soll ein Tupel von k Zahlen $x_1, \ldots, x_k \in \mathbb{N}_0$ sein.

Die **initiale Registerbelegung** lautet:

$R[x_1, \ldots, x_k] = \{(1, x_1), (2, x_2), \ldots, (k, x_k)\}$.
Jedem Register ist also ein Index und ein Wert zugeordnet. Zum Beginn der Ausführung steht auf den ersten k Registern die Eingabe. Der Speicherbedarf an Registern kann aber während der Ausführung wachsen. Die Anzahl der insgesamt verwendeten Register ist bei Beginn der Ausführung also unbekannt. Weiters wird definiert, dass der Akkumulator Teil der Registerbelegung ist. Vorgegeben ist auch, dass r_0 den Index 0 besitzt. Eine Schreibweise für 2 Parameter y,x als [y,x] besagt, dass auf dem ersten Register nach dem Akkumulator y und auf dem zweiten x zu finden ist. Dies muss immer definiert sein, hat aber für die Implementierung keinen Einfluss auf die Laufzeit.

2.3 Konfigurationen und Relationen

Definition Konfiguration

Es ist möglich den aktuellen Zustand κ der RM in einem Tupel zu speichern. Die veränderlichen Teile der Maschine sind lediglich der Befehlszähler und die Register (inkl. Akkumulator). Befehlszähler b ist der nächste auszuführende Befehl von $\mathcal{R}$ (RM-Programm). Die Registerbelegung R hält alle Register, die von der Maschine seit Beginn der Ausführung beschrieben wurden als Index-Wert-Paar.

$$\kappa = (b, R)$$
$$b \in \mathbb{N}_0$$

$$R \subset \mathbb{N}_0^2 \quad \text{bzw.} \quad R \in P(\mathbb{N}_0^2)$$

Konfigurationsrelation $\underset{\mathcal{R}}{\rightarrow}$

Durch den einfachen Aufbau der RM und die simplen Befehle, die entweder nur den Akkumulator und evtl. ein Register verändern, kann bei gegebenem $\mathcal{R}$ und der initialen Registerbelegung R zu jedem Zeitpunkt auf die folgenden Konfigurationen geschlossen werden. Diese Übergänge nennt man Konfigurationsrelationen.

$$\kappa_1 \rightarrow \kappa_2 \rightarrow \kappa_3 ...$$

Da die RM deterministisch arbeitet, ist die nachfolgende Konfiguration zu jedem Zeitpunkt durch das RM Programm eindeutig bestimmt. Eine Konfiguration κ' ist von einer Konfiguration κ erreichbar wenn das gegebene Programm $\mathcal{R}$ ab aktuellem b zwischen ihnen eine Verbindung erzeugt. Dies passiert durch die Überlegung, dass der nächste Befehl entweder ein Sprungbefehl auf κ' ist, oder die nächste Konfiguration (mit neuem b und veränderten Registerwert) selbst eine Relation auf κ' besitzt:
Sei $\kappa = (b, R)$ und $\kappa' = (b', R')$, dann ist $\kappa \underset{\mathcal{R}}{\rightarrow} \kappa'$ wahr, gdw. entweder π_b ein Sprungbefehl nach b' und $R' = R$ ist, oder $b' = b + 1$ und $R' = \{(j, x)\} \cup (R \setminus \{(i, y) \in \mathbb{N}_0^2 | i = j\})$, wobei j das von π_b adressierte Register und x dessen neu berechneter Inhalt ist.

Eine Konfiguration ist in 0 Schritten zu erreichen wenn Anfangs- und Endzustand gleich sind.

$$\kappa \xrightarrow[\mathcal{R}]{0} \kappa' \text{ genau dann wenn } \kappa = \kappa'$$

Von einer Konfiguration κ ist eine Konfiguration κ' dann in n+1 Schritten erreichbar, wenn es eine Zwischenkonfiguration κ'' gibt. κ'' besteht dabei aus einer beliebig langen Anordnung an Registern. Die Zwischenkonfiguration ist dabei von der Startkonfiguration in n Schritten erreichbar und die Endkonfiguration ist von der Zwischenkonfiguration in einem Schritt erreichbar.

$$\kappa \xrightarrow[\mathcal{R}]{n+1} \kappa' \Leftrightarrow \exists \kappa'' \in \mathbb{N}_0 \times P(\mathbb{N}_0^2) : \ \kappa \xrightarrow[\mathcal{R}]{n} \kappa'' \ \wedge \ \kappa'' \underset{\mathcal{R}}{\rightarrow} \kappa'$$

Zur Erklärung der Zwischenkonfiguration κ'': Was hier beschrieben wird, ist der Raum aus dem die Werte ausgewählt sind. Deswegen bezeichnet diese formale Beschreibung nicht eine konkrete Zwischenkonfiguration, sondern die Existenz einer solchen:

- $\mathbb{N}_0^2$: Ein Register ist immer ein Tupel mit (Index, Wert) z.B.: $(1,2)/(2,3)/\ldots$
- $P(\mathbb{N}_0^2)$: Potenzmenge von allen möglichen Tupeln. Es handelt sich also um eine Menge von Tupeln, z.B.:
  $\{(1,2),(2,3),(1,3),\ldots\}/\{(1,3),(4,5),\ldots\}/\ldots$
- $\mathbb{N}_0 \times P(\mathbb{N}_0^2)$: Kreuzmenge führt den Befehlszähler b ein, der ebenfalls aus dem Raum der natürlichen Zahlen inklusive 0 genommen wird z.B.:
  $\{(\underbrace{2}_{b}, \underbrace{\{(1,2),(2,3),(1,3),\ldots\}}_{\text{Raum von } R})/(\underbrace{10}_{b}, \underbrace{\{(1,3),(4,5),\ldots\}}_{\text{Raum von } R})/\ \ldots$

Eine Konfiguration besteht aus einem Befehlszähler, allen möglichen Registern, die wiederum aus Tupeln bestehen. $\xrightarrow[\mathcal{R}]{*}$ steht für eine beliebige Anzahl an Konfigurationsübergängen – die transitive Hülle der Zustandsübergänge (siehe Kapitel 0.4.2).

2.4 Partielle Funktion einer RM

Um nun die Abbildungsfähigkeit einer RM definieren zu können, wird der Begriff der partiellen Funktion einer RM eingeführt. Dabei geht es um die Frage welche Funktionen (bzw. Algorithmen) durch eine RM dargestellt werden können.
Ausschlaggebend ist hierbei das Erreichen des END-Befehles, der (wie erwähnt) den Befehlszähler b auf 0 setzt. Sollte es eine Endkonfiguration R' geben, die dieses Kriterium erfüllt, befindet sich im Akkumulator das Ergebnis der Funktion. Wenn die Maschine niemals anhält, also divergiert, ist das Ergebnis der berechneten Funktion undefiniert. Wir schreiben $\perp$ (Bottom »Bot«) um dies zu zeigen. Dies ist auch der einzige Unterschied zu einer normalen Funktionsdefinition: Es ist möglich auch auf einen ungültigen Zustand abzubilden.
Die durch $\mathcal{R}$ berechnete partielle Funktion $f_{\mathcal{R}} : \mathbb{N}_0^k \mapsto \mathbb{N}_0$ bildet von einer Eingabe $(x_1, \ldots, x_k)$ auf eine natürliche Zahl aus $\mathbb{N}_0$ ab (genau die Zahl, die im Akkumlator stehen wird).

$$f_{\mathcal{R}}(x_1, \ldots, x_k) = \begin{cases} r_0 & : \exists R' : \kappa_0 \xrightarrow[\mathcal{R}]{*} (0, R') \wedge (0, r_0) \in R' \\ \perp & : \nexists R' : \kappa_0 \xrightarrow[\mathcal{R}]{*} (0, R') \end{cases}$$

wobei $\kappa_0 = (1, R[x_1, \ldots, x_k])$ die Startkonfiguration darstellt.

Eine partielle Funktion $f : \mathbb{N}_0^k \mapsto \mathbb{N}_0$ wird *RM-berechenbar* genannt, wenn es eine Registermaschine $\mathcal{R}$ gibt, sodass $f = f_{\mathcal{R}}$

Beispiel 2.1

Startkonfiguration: $\kappa_0 = (1, R[y, x]), \quad x, y \in \mathbb{N}_0$.

Gesucht: RM-Programm $\mathcal{R}$, sodass $f_{\mathcal{R}}(x, y) = x^y$.

Der Pseudocode für x^y sieht intuitiv wie folgt aus.

$z = 1;$

`while` $y > 0$

$\quad z := z * x;$

$\quad y := y - 1;$

`while-end`

»Multipliziere so lange, bis ständiges Dekrementieren der Potenz zu 1 führt«
Für den tatsächlichen Code auf der RM werden natürlich mehr Befehle benötigt. Es wird davon ausgegangen, dass

- $x \rightarrow r_1$
- $y \rightarrow r_2$
- $z \rightarrow r_3$

Programm der Registermaschine:

```
 1 LOAD #1   //lade den Wert 1 in den Akkumulator
 2 STORE 3   //speichere den Wert auf das Register 3
 3 LOAD 2    //lade y (Exponent) in den Akkumulator
 4 JZERO 11  //sollte sie Null sein springe auf 11
 5 SUB #1    //der Exponent kann dekrementiert werden
 6 STORE 2   //speichere die dekrementierte Potenz auf alte Position
 7 LOAD 3    //lade x (Basis) in den Akkumulator
 8 MULT 1    //multipliziere Basis mit sich selbst
 9 STORE 3   //speichere quadrtiertes Ergebnis auf alte Position
10 GOTO 3    //starte erneut mit Potenzierung bzw Abfrage
11 LOAD 3    //das Ergebnis steht sicher an Stelle von z, lade es in den Akkumulator
12 END       //terminiere
```

2.5 Kostenmaße

Um nun mit dem RM-Model Algorithmen analysieren zu können muss man ihren Platz- und Zeitbedarf wohl definieren. Hier gibt es zwei unterschiedliche Maße, die sich sowohl in ihrer Berechnung, als auch in ihrer Aussagekraft unterscheiden. Es gilt dabei folgende formale Beschreibung bezüglich des Endes einer Ausführung:

$$(1, R[x_1, \ldots, x_k]) \xrightarrow[\mathcal{R}]{i} (b_i, R_i), \text{ mit } b_N = 0$$

Programm $\mathcal{R}$ wird mit Registerstellung R und mit Befehlszähler $b = 1$ gestartet. Es werden i Schritte benötigt, bis ein Befehlszähler b_i und eine Registerstellung R erreicht wird, wobei b_i den Wert 0 haben muss. Aufgrund dieser Überlegungen können nun Kostenmaße bis zum Ausführungsende definiert werden.

2.5.1 Uniformes Kostenmaß

Das *uniforme Kostenmaß* ist das simplere von zwei möglichen. Es ist einfacher zu berechnen, ist aber nur mit einer Einschränkung der Registergröße hinsichtlich der Komplexität aussagekräftig.

Definition 2.1 (Uniforme Zeitkosten). *Die Uniforme Zeitkosten der RM $\mathcal{R}$ bei Eingabe von $x_1, \ldots, x_k$ sind definiert als*

$$t^u_{\mathcal{R}}(x_1, \ldots, x_k) = N$$

»Uniforme (u) Zeitkosten (t) für Programm $\mathcal{R}$ ($\mathcal{R}$) mit der Eingabe $x_1, \ldots, x_k$ gleich N«

Hier werden die Befehle i gezählt, die benötigt werden, um die Ausführung des Programms $\mathcal{R}$ abzuschließen.

Definition 2.2 (Uniforme Platzkosten). *Die uniformen Platzkosten der RM $\mathcal{R}$ bei Eingabe von $x_1, \ldots, x_k$ sind definiert als*

$$s^u_{\mathcal{R}}(x_1, \ldots, x_k) = \sum_{j=0}^{\infty} \begin{cases} 1 & \exists 0 \leq i \leq N : r_{i,j} \neq 0 \\ 0 & \forall 0 \leq i \leq N : r_{i,j} = 0 \end{cases}$$

Es werden alle Register einmal gezählt, die im Laufe des Programms einen Wert ungleich 0 hatten.

2.5.2 Warum reicht uniformes Kostenmaß nicht?

Beispiel 2.2

Als Beispiel sei das uniforme Kostenmaß für 2.1 gesucht.

Zeitbedarf

$t^u_{\mathcal{R}}(x,y) =$	$4+$	Zeile 1-4
	$+8y+2$	Zeile 5-10,3,4; y Durchläufe
	$+2$	Zeile 11,12
	$= 8y+6$	

$\rightarrow t^u_{\mathcal{R}}(x,y) = 8y+6$

Speicherbedarf Die zwei Operanden benötigen jeweils ein Register. Der Output wird während des Durchlaus auf r_3 gespeichert. Zwischendurch wird aber der Akkumulator benötigt - Insgesamt also 4 Register.
$\rightarrow s^u_{\mathcal{R}}(x,y) = 4$

Dies entspricht nicht der intuitiven Erwartung, da in den Registern unendlich grosse Zahlen gespeichert werden können (Platzbedarf), und eine längere Zeit benötigt wird, um größere Zahlen zu schreiben (Zeitbedarf). Daher wird die Frage der Kodierung behandelt und ein neues Kostenmaß eingeführt: das logarithmische Kostenmaß.

2.5.3 Kodierung / Logarithmische Länge

Es wird definiert, dass die Zahlen binär kodiert werden. Für die Berechnung macht eine abweichende Kodierung keinen Unterschied. Die folgenden Schritte (z.B. Basiswechsel) sind dann aber dementsprechend zu ändern.
Um nun eine binäre Zahl in ihrer Komplexität zu definieren, wird ihre Länge benutzt. Dies funktioniert mit dem Logarithmus der Basis 2. Beispiel: $log(10_{10}) = log(1010_2) = 3.321\ldots$ Um eine ganze Zahl zu erhalten (nicht jede Zahl ist als Basis von 2 anschreibbar) muss mit der floor-Funktion abgerundet und anschließend inkrementiert werden.
Für $x \in \mathbb{N}_0$ ist die **Logarithmische Länge** die Anzahl der für die binäre Darstellung von x benötigten Bits.

$$L(x) = \begin{cases} 1 & : \quad x = 0 \\ \lfloor \log x \rfloor + 1 & : \quad x \geq 1 \end{cases}$$

2.5.4 Logarithmische Kostenmaße

Definition 2.3 (Logarithmische Zeitkosten)**.** *Die logarithmischen Zeitkosten der RM $\mathcal{R}$ bei Eingabe von $x_1, \ldots, x_k$ sind definiert als*

$$t_{\mathcal{R}}(x_1, \ldots, x_k) = \sum_{j=1}^{N} \left\{ \begin{array}{ll} 1 & \pi_{b_j} = GOTO|END \\ L(r_{i_j,0}) & \pi_{b_j} = JZERO\ op \\ L(r_{i_j,0}) + L(op) & \pi_{b_j} = ADD|SUB|MULT|DIV\ \#op \\ L(r_{i_j,0}) + L(op) & \pi_{b_j} = STORE\ op \\ L(r_{i_j,0}) + L(op) + L(r_{i_j,op}) & \pi_{b_j} = ADD|SUB|MULT|DIV\ op \\ L(r_{i_j,0}) + L(op) + L(r_{i_j,op}) & \pi_{b_j} = STORE\ *op \\ L(r_{i_j,0}) + L(op) + L(r_{i_j,op}) + & \\ \quad + L(r_{i_j,r_{i_j,op}}) & \pi_{b_j} = ADD|SUB|MULT|DIV\ *op \\ L(op) & \pi_{b_j} = LOAD\ \#op \\ L(op) + L(r_{i_j,op}) & \pi_{b_j} = LOAD\ op \\ L(op) + L(r_{i_j,op}) + L(r_{i_j,r_{i_j,op}}) & \pi_{b_j} = LOAD\ *op \end{array} \right\}.$$

In der rechten Spalte findet man den Befehl. In der linken Spalte finden sich die Kosten als Funktion der logarithmischen Kosten. N ist hier wiederum die Anzahl der benötigten Schritte, bis der Endzustand erreicht wird. Der Index läuft bis N da jeder Befehl, auch der End-Befehl, beachtet wird. Der Index j steht für den Ausführungszeitpunkt; i_j ist der Index des Registers das zum Zeitpunkt j referenziert wird und b_j ist der Befehlszähler zum Zeitpunkt j.
Wie man sieht, müssen bei unterschiedlichen Adressierungsarten nicht nur die log. Kosten des Operators sondern auch evtl. die der indirekt angesprochenen Register in die Berechnung aufgenommen werden.

Definition 2.4 (logarithmische Platzkosten)**.** *Hier wird mit den log. Kosten der höchsten jemals gespeicherten Zahl gerechnet. Die Definition ist ähnlich den uniformen Zeitkosten:*

$$s_{\mathcal{R}}(x_1, \ldots, x_k) = \sum_{j=0}^{\infty} \left\{ \begin{array}{ll} \max_{0 \leq i \leq N} L(r_{i,j}) & \exists 0 \leq i \leq N : r_{i,j} \neq 0 \\ 0 & \forall 0 \leq i \leq N : r_{i,j} = 0 \end{array} \right.$$

2.5.5 Gegenüberstellung uniforme/logarithmische Zeitkosten

Zeitkosten pro Befehl hier werden die einzelnen Zeitkosten für die verwendeten Befehle aufgezeigt:

			uniform	logarithmisch
1	`LOAD #1`	$r_0 \leftarrow 1$	1	1
2	`STORE 3`	$r_3 \leftarrow r_0$	1	3
3	`LOAD 2`	$r_0 \leftarrow r_2$	1	$2 + L(r_2)$
4	`JZERO 11`	$r_0 \stackrel{?}{=} 0$	1	$L(r_0)$
5	`SUB #1`	$r_0 \leftarrow r_0 - 1$	1	$L(r_0) + 1$
6	`STORE 2`	$r_2 \leftarrow r_0$	1	$2 + L(r_0)$
7	`LOAD 3`	$r_0 \leftarrow r_3$	1	$2 + L(r_3)$
8	`MULT 1`	$r_0 \leftarrow r_0 * r_1$	1	$L(r_0) + 1 + L(r_1)$
9	`STORE 3`	$r_3 \leftarrow r_0$	1	$2 + L(r_0)$
10	`GOTO 3`		1	1
11	`LOAD 3`	$r_0 \leftarrow r_3$	1	$2 + L(r_3)$
12	`END`		1	1

Beispiel 2.3

Berechnung der Gesamtkosten Nun werden die Zeitkosten für beide Maße berechnet:
Uniforme Zeitkosten, bereits bekannt, siehe Beispiel 2.2: $= 8y + 6$

Logarithmische Zeitkosten: $(\sum_{i=1}^{y} L(i) \approx y(L(y) - 1))$

$$
\begin{aligned}
t_{\mathcal{R}}(x,y) &= 9 + 2L(y) + yL(x) && \text{Z 1-4,11-12}\\
&\quad +4\textstyle\sum_{i=1}^{y} L(i) + 3L(x)\sum_{i=1}^{y} i - yL(x) + 11y && \text{Z 5-10,3-4}\\
&\approx \frac{3}{2}y^2L(x) + (4y+2)L(y) + y(7 - \frac{3}{2}L(x)) + 9
\end{aligned}
$$

Um die Kosten zu vergleichen, wird der Begriff der Komplexität eingeführt.

2.6 Komplexität

Die Komplexität eines Systems ist eine vereinfachende Möglichkeit ähnliche Systeme in Klassen zusammen zu fassen. Man unterscheidet hier grundsätzlich durch mathematische Funktionen (linear, polynomiell, exponentiell). Für nähere Informationen siehe die LV Datenstrukturen und Algorithmen.

Definition 2.5 (Zeitkomplexität der RM). *Die **Zeitkomplexität** $T_{\mathcal{R}}(n)$ einer RM $\mathcal{R}$ in Abhängigkeit der logarithmischen Länge n der Eingabe ist definiert als*

$$T_{\mathcal{R}}(n) = \max_{(x_1,\ldots,x_k)\in\mathcal{N}_0^k:\, L(x_1)+\cdots+L(x_k)\leq n} t_{\mathcal{R}}(x_1,\ldots,x_k)$$

Definition 2.6 (Platzkomplexität der RM). *Die **Platzkomplexität** $S_{\mathcal{R}}(n)$ einer RM $\mathcal{R}$ in Abhängigkeit der logarithmischen Länge n der Eingabe ist definiert als*

$$S_{\mathcal{R}}(n) = \max_{(x_1,\ldots,x_k)\in\mathcal{N}_0^k:\, L(x_1)+\ldots+L(x_k)\leq n} s_{\mathcal{R}}(x_1,\ldots,x_k)$$

Beispiel 2.4

Berechnung der Komplexität von 2.1 unter dem Uniformen Kostenmaß

$$\begin{aligned} t^u_{\mathcal{R}}(x,y) &= 8y+4 && L(x)=n_x, L(y)=n_y && (2.1)\\ &= 8\cdot 2^{n_y}+4 && n=n_x+n_y && (2.2)\\ T^u_{\mathcal{R}}(n) &<= 8\cdot 2^{n-1+4} && =\mathcal{O}(2^n) && (2.3)\end{aligned}$$

(1.1) Wir benutzen die vorher berechneten uniformen Kosten und definieren die logarithmischen Eingabelängen als n_y und n_x.
(1.2) Hier setzen wir statt y n_y ein. n bezeichnet die Gesamtlänge der Eingabe.
(1.3) Durch Abschätzen können wir n einführen und erhalten insgesamt ein exponentielles Ergebnis.
→ Wir merken: Linearer Eingabewert resultiert zu exponentieller Komplexität in der Eingabelänge bei uniformen Kostenmaß.

Berechnung der Komplexität von 2.1 unter dem Logarithmisches Kostenmaß

$$\begin{aligned} t_{\mathcal{R}}(x,y) &= \frac{3}{2}y^2L(x)+(4y+2)L(y)+y(7-\frac{3}{2}L(x))+9 && (2.4)\\ &\leq \frac{3}{2}2^{2n_y}n_x+(4\cdot 2^{n_y}+2)n_y+2^{n_y}(6-\frac{3}{2}n_x)+9 && (2.5)\end{aligned}$$

$$\begin{aligned} T_{\mathcal{R}}(n) &\leq \frac{3}{2}2^{2n}n+(4\cdot 2^n+2)n+2^n(6-\frac{3}{2}n)+9 && (2.6)\\ &\leq \frac{3}{2}n2^{2n}+\frac{5}{2}n2^n+6\cdot 2^n+2n+9 && (2.7)\\ &= \mathcal{O}(n2^{2n}) && (2.8)\end{aligned}$$

also sogar superexponentielle Laufzeit ($\notin \mathcal{O}(poly(n))$)
Hinweis: Wir sind für Komplexitätsuntersuchungen immer nur an asymptotischen Größen interessiert. Es hätte daher im Beispiel genügt, den komplexesten Schleifenbefehl ($\frac{3}{2}y^2L(x)$) zu untersuchen.

Analyse der Ergebnisse Wie es scheint wird in beiden Fällen nicht die erwartete exponentielle Laufzeit berechnet. Im uniformen Maß gibt es zwar ein exponentielles Ergebnis, das sich aber aus linearen Zeitkosten ableitet. Es scheint als würden wir nicht einen effizienten Algorithmus verwenden.

Vereinfachung des Algorithmus Grundidee (Square & Multiply):

- Reduziere die Anzahl der Schleifendurchläufe
- Ersetze $z = z * x$ durch $z = z * z$

- Anzahl der Schleifendurchläufe nur mehr: $\log y$

Zeitbedarf unter dem uniformen Kostenmaß: $T^u_{\mathcal{R}}(n) = \mathcal{O}(n)$
Zeitbedarf unter dem logarithmischen Kostenmaß: $T_{\mathcal{R}}(n) = \mathcal{O}(2^n)$

Erklärung Wir haben nun die Diskrepanz geschlossen zwischen der intuitiv einfachen berechenbaren Funktion der Exponentialfunktion und einem effizienten Algorithmus. Wir sehen, dass die Berechnung im uniformen Kostenmaß unsere Erwartungen nicht erfüllt, dabei einfacher zu berechnen ist und unter gewissen Umständen ebenfalls als Komplexitätsbestimmung verwendet werden kann. Die Berechnungen im logarithmischen Kostenmaß hingegen zeigen eine korrekte Abbildung der Funktion in den logarithmischen Kosten als auch in ihrer Komplexität.

2.7 Polynomielle Zeitbeschränkung

Definition 2.7 (polynomiell zeitbeschränkte RM)**.** *Wir nennen eine RM $\mathcal{R}$ polynomiell zeitbeschränkt, falls die logarithmische Kostenfunktion $T_{\mathcal{R}}$ polynomiell zeitbeschränkt ist.*

$$T_{\mathcal{R}}(n) = \mathcal{O}\left(poly(n)\right)$$

d.h. es gibt ein Polynom p für das gilt

$$T_{\mathcal{R}}(n) \leq p(n) \quad \forall \quad n \in \mathcal{N}$$

mit Hilfe von uniformen Kostenmaß Solange man nur die Frage der polynomiellen Laufzeit beantworten möchte, kann man unter folgenden Voraussetzungen das uniforme Kostenmaß verwenden.

- $T^u_{\mathcal{R}}(n) = \mathcal{O}(poly(n)))$
- $\max_{i,j} L(r_{i,j}) = \mathcal{O}(poly(n))$ Die maximale Registerlänge (Eingabe als auch während Laufzeit entstanden) darf sich maximal um einen linearen Faktor von der Länge der Inputzahlen (n) unterscheiden

Sind beide Voraussetzungen erfüllt, dann gilt

- $T_{\mathcal{R}}(n) = \mathcal{O}(poly(n))$

2.8 RM mit/ohne Multiplikation

Das RM-Modell kann ohne Verlust der Berechnungsstärke eingeschränkt werden, indem (zB) auf die Multiplikation verzichtet wird:

- Eine RM $\mathcal{R}'$ ohne MULT

- Simuliert eine RM $\mathcal{R}$ mit MULT

- sodass $T_{\mathcal{R}'}(n) = \mathcal{O}(T_{\mathcal{R}}(n)^2)$ (ohne Beweis)

Um dies zu veranschaulichen wird auf einen graphischen Beweis zurückgegriffen.
Man nehme an, dass auf einer Zeitachse die Befehle eines Programms aufgetragen werden. Dabei benötigen einige Befehle länger als andere. Die fett markierten Befehle sind Multiplikationen. Abbildung 2.2 zeigt die Ausführungszeiten einzelner Befehle bei der Abarbeitung des Programmes

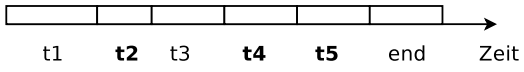

Abbildung 2.2: RM mit MULT

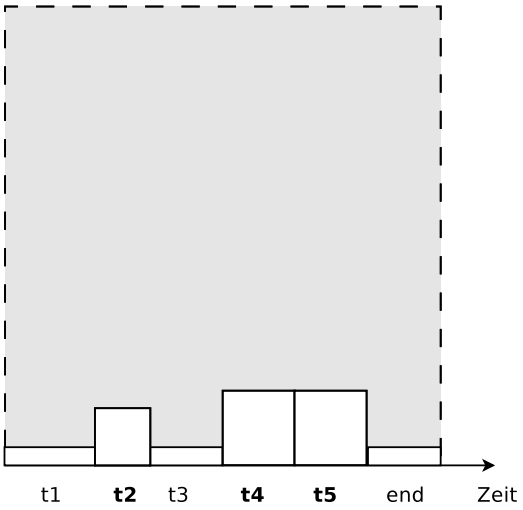

Abbildung 2.3: RM mit simulierten Mult-Befehlen

Die Gesamtlaufzeit t von dem Program $\mathcal{R}$ wird als Summe der einzelnen Zeiten $t_1 \dots t_n$ definiert. Nun wird angenommen, dass die simulierte Ausführungsgeschwindigkeit einer Multiplikation auf einer RM ohne Mult quadratisch ist. $t(MULT) = \mathcal{O}(n^2)$. Wir ersetzen nun alle MULT-Befehle durch eine quadratische Zeit, da wir sie in Kauf nehmen müssen, wenn wir den MULT-Befehl simulieren wollen, erkennbar an den Quadraten. Im grauen Gesamtquadrat wird weiters die Quadrierung über alle Befehle gezogen. Wir sehen nun, dass selbst, wenn das Programm einzig und allein aus MULT-Befehlen besteht, die Zeitkosten dieses Rechteck nicht verlassen können. Der Grund darin liegt, dass die Summe der Quadrate kleiner als das Quadrat der Summe ist. Die Nachstellung einer RM ohne MULT kann also in maximal quadratischer Zeit erfolgen (siehe Abbildung 2.3).

3 Turingmaschine

Dominik Hirner, Philipp Kober

3.1 Church-Turing-These

Der britische Mathematiker, Alan Turing wollte den *mathematisch arbeitenden Menschen* mittels eines Maschinenmodells beschreiben um einen formalen Begriff für mathematische Berechnungen zu kreieren. In seinen Überlegungen betrachtete Turing die Arbeit eines Mathematikers auf einem Blatt Papier. Zu Beginn steht eine Rechnung auf dem Papier. Der Mathematiker liest die Rechnung, schreibt Zwischenergebnisse und die Lösung wieder darauf – er benötigt also, seine Augen zum Lesen der Rechnung und von Zwischenergebnissen, seine Hände zum schreiben und einen Programm oder Algorithmus um die einzelnen Verarbeitungsschritte durchzuführen. Turing erkannte, dass sich dieses Modell vereinfachen lässt, ohne die Berechnungsstärke wesentlich einzuschränken. Das 2-dimensionale Blatt Papier kann durch ein 1-dimensionales Band ersetzt werden, ein Schreib- /Lesekopf repräsentiert die Hand zum Schreiben und die Augen zum Lesen als eine Einheit. Als Eingabe zählt die Rechnung auf dem Band, welches ebenso als Zwischenspeicher und als Ausgabe dient.

Dieses Maschinenmodell wird als *Turingmaschine* bezeichnet. Funktionen, die auf diesem Modell berechnet werden können nennt man Turing-berechenbar. Es wird angenommen, dass die Klasse der intuitiv berechenbaren Funktion genau der Klasse der Turing-berechenbaren Funktionen entspricht. Diese These, bekannt als *Church-Turing These*, ist eine Grundannahme der Komplexitätstheorie und rechtfertigt die Verwendung der Turingmaschine als universelles Berechnungsmodell.

3.1.1 Formale Definition der Turingmaschine

Eine Turingmaschine ist ein Modell eines sequentiellen Rechners und besteht in seiner einfachsten Variante aus einem unendlich langem Band mit sequentiell angeordneten Feldern, auf denen jeweils genau ein Zeichen (des Bandalphabets) gespeichert ist, und einem Lese-/Schreibkopf. Zu Beginn steht nur eine Eingabe auf dem Band, die die Turingmaschine nach einem gegeben Programm modifiziert, der Rest ist gefüllt mit »Blank-Symbolen« ($\square$). Um die Eingabe zu lesen und das Band zu modifizieren kann die Turingmaschine den Schreib/Lesekopf nach links und rechts bewegen und Zeichen lesen und schreiben. Die Maschine rechnet solange weiter, bis ein definierter Halt-Zustand erreicht wurde. Im Fall der Akzeptanz steht die Ausgabe rechts vom Schreib-/Lesekopf.

Definition 3.1 (Deterministische Turingmaschine)**.** *Eine deterministische Turingmaschine $\mathcal{T}$ ist ein 7-Tupel*

$$\mathcal{T} = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$$

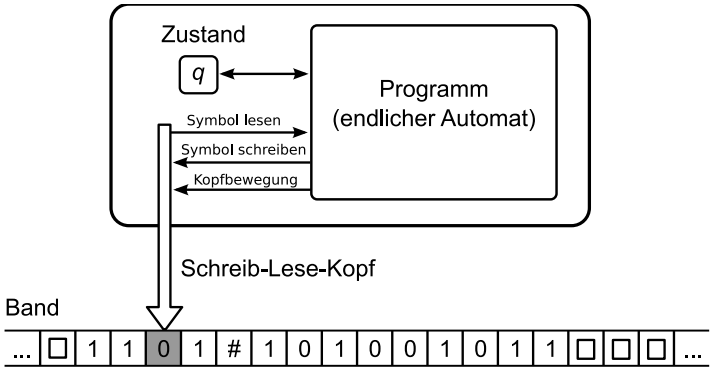

Abbildung 3.1: Schematische Darstellung einer Turingmaschine

- Q, *eine Menge von Zustaenden*
- Σ, *das Eingabealphabet,* $\square \notin \Sigma$
- Γ, *das Bandalphabet,* $\Sigma \subseteq \Gamma$, *Blanksymbol:*$\square \in \Gamma$
- $q_0 \in Q$, *der Anfangszustand*
- $F \subset Q$, *eine Menge von Endzustaenden*
- $\square$, *das Blanksymbol*
- $\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{\leftarrow, -, \rightarrow\}$, *die Uebergangsfunktion*

3.1.2 Endlicher Zustandsautomat

Das Programm der Turingmaschine wird durch einen *endlichen Zustandsautomaten* beschrieben. Ein endlicher Zustandsautomat (engl. finite state machine oder FSM) ist ein Modell bestehend aus einer endlichen Menge von Zuständen Q, den Zustandsübergängen δ, dem Startzustand q_0 und (einem oder mehreren) fixen Endzuständen F. In jedem Zustand wird entschieden welcher Folgezustand betreten wird. Um diese Entscheidung zu treffen wird das Symbol, dass unter dem Lesekopf steht gelesen. Beim Zustandsübergang wird dieses mit einem neuen Symbol überschrieben und der Schreib-/Lese-Kopf führt eine Bewegung aus $(\leftarrow, -, \rightarrow)$.

3.2 Konfiguration

3.2.1 Berechnungspfad

Durch die Abfolge verschiedener Zustände bzw. Operationen (bzw. durch ein Programm) einer Turingmaschine (TM) entsteht ein Berechnungspfad. Mit α wird der gesamte Bandinhalt links von der Kopfposition bezeichnet, mit β das gesamte restliche Band rechts von der Kopfposition und mit $\mathbf{q}$ der momentane Zustand sowie die momentane Position des Schreib/Lesekopfes.
$\alpha_0 q_0 \beta_0 \rightarrow \alpha_1 q_1 \beta_1 \rightarrow \ldots$

Ist der Endzustand des Berechnungspfades q_N Teil von F (Menge der Endzustände der Turingmaschine), dann ist die Berechnung akzeptierend, ansonsten verwerfend. Wird kein Endzustand erreicht, ist die Berechnung unendlich (Endlosschleife).

Für jede Berechnung einer Turingmaschine kann also einer von drei Zuständen eintreffen:

- Akzeptierender Zustand (es gibt einen Endzustand und er ist Teil von F)
- Verwerfender Zustand (es gibt einen Endzustand und er ist kein Teil von F)
- Nicht entscheidender Zustand (Endlosschleife)

oder formal:

$$f_{\mathcal{T}} : \Sigma^* \mapsto \{JA, NEIN, \perp\}$$

$$f_{\mathcal{T}}(w) = \begin{cases} JA & : q_0 w \xrightarrow[\mathcal{T}]{*} \alpha\, q\, \beta \quad \wedge\ q \in F \\ NEIN & : q_0 w \xrightarrow[\mathcal{T}]{*} \alpha\, q\, \beta \not\rightarrow_{\mathcal{T}} \quad \wedge\, q \notin F \\ \perp & : \text{sonst} \end{cases}$$

3.2.2 Ausgabe einer Turingmaschine

Wird ein akzeptierender Zustand erreicht, kann das Ergebnis vom Band gelesen werden. Es befindet sich dann vom Schreib/Lesekopf nach rechts bis zum ersten □ (Blank-Symbol).

3.2.3 Konfigurationsrelation einer Turingmaschine

Die schrittweise Abarbeitung der DTM $\mathcal{T}$ wird durch die Übergangsfunktion δ (das Programm) festgelegt. Daraus leitet sich die *Konfigurationsrelation* $\xrightarrow[\mathcal{T}]{}$ wie folgt ab:

$$\begin{aligned}
\delta(q,b) = (q', c, -) \quad &\Rightarrow \quad \alpha\, a\, q\, b\, \beta \xrightarrow[\mathcal{T}]{} \alpha\, a\, q'\, c\, \beta \\
\delta(q,b) = (q', c, \rightarrow) \quad &\Rightarrow \quad \alpha\, a\, q\, b\, \beta \xrightarrow[\mathcal{T}]{} \alpha\, a\, c\, q'\, \beta \\
\delta(q,b) = (q', c, \leftarrow) \quad &\Rightarrow \quad \alpha\, a\, q\, b\, \beta \xrightarrow[\mathcal{T}]{} \alpha\, q'\, a\, c\, \beta \\
\delta(q,b) = \perp \vee q \in F \quad &\Rightarrow \quad \alpha\, a\, q\, b\, \beta \not\rightarrow_{\mathcal{T}}
\end{aligned}$$

$\xrightarrow[\mathcal{T}]{*}$ ist die reflexive, transitive Hülle von $\xrightarrow[\mathcal{T}]{}$, analog $\xrightarrow[\mathcal{T}]{+}$, $\xrightarrow[\mathcal{T}]{i}$

$q, q' \in Q, \quad a, b, c \in \Gamma, \quad \alpha, \beta \in \Gamma^*$

3.2.4 Kostenmaß

Beim Kostenmaß einer Turingmaschine wird prinzipiell zwischen zwei Maße unterschieden:

- Platzkosten ($s_{\mathcal{T}}$ bei einer gewissen Eingabe w der Turingmaschine)
  Gibt die Länge des (endlichen) Berechnungspfades der TM $\mathcal{T}$ bei der Eingabe w an (bzw. ∞ falls der Berechnungspfad unendlich ist)
- Zeitkosten ($t_{\mathcal{T}}$ bei einer gewissen Eingabe w der Turingmaschine)
  gibt die Anzahl der Bandquadrate an, die während der Berechnug der TM $\mathcal{T}$ bei Eingabe von w besucht werden

Offensichtlich, sind die Platzkosten immer durch die Zeitkosten beschränkt bzw. $\forall\mathcal{T} : \forall w \in \Sigma^* : s_{\mathcal{T}}(w) \leq t_{\mathcal{T}}(w)$, da in jedem Zeitschritt maximal ein neues Bandquadrat besucht werden kann.

3.3 Mehrbändige Turingmaschine

3.3.1 Defnition

Die Turingmaschine besitzt ein Band, das sowohl als Eingabe- und Ausgabeband und als Speicher arbeitet. Oft ist es jedoch praktisch das Modell zu erweitern. Eine Mehrbändige Turingmaschine, auch k-band-DTM, besteht im Gegensatz dazu aus k Bändern, welche als Arbeitsbänder dienen, auf die lesen und/oder schreiben zugegriffen werden kann. Man kann eins dieser Bänder z.B. nur für die Eingabe reservieren oder spezielle Ausgabebänder definieren.
Jedes dieser Bänder besitzt einen eigenen Schreib-/Lesekopf und der nächste auszuführende Schritt hängt von den k Bandsymbolen und dem aktuellen Zustand ab. Wie in einer einbändigen DTM kann jedes dieser k Symbole überschrieben werden, jeder Schreib-/Lesekopf kann, unabhängig von den anderen, nach links, rechts oder überhaupt nicht bewegt werden.

Formell gesehen ist eine k-band-DTM:
$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{\leftarrow, -, \rightarrow\}^k$
wobei k die Anzahl der Bänder ist.

Der Ausdruck

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, \leftarrow, \rightarrow, \ldots, \leftarrow)$$

bedeutet, dass, wenn die Maschine im Zustand q_i ist und die Köpfe 1 bis k lesen die Symbole a_1 bis a_k, dann geht die sie in Zustand q_j, schreibt Symbol b_1 bis b_k und weist jeden der k Köpfe an, sich nach links/rechts oder garnicht zu bewegen, je nach Spezifikation des Zustands q_j.

Beispiel 3.1

Abbildung 3.2 zeigt das Programm einer Turingmaschine, welche binäre Palindrome entscheiden kann. Diese Maschine aktzeptiert z.B. die Eingabe `01010`. Die Regeln für die Zustandsübergänge dieser Maschine sind:

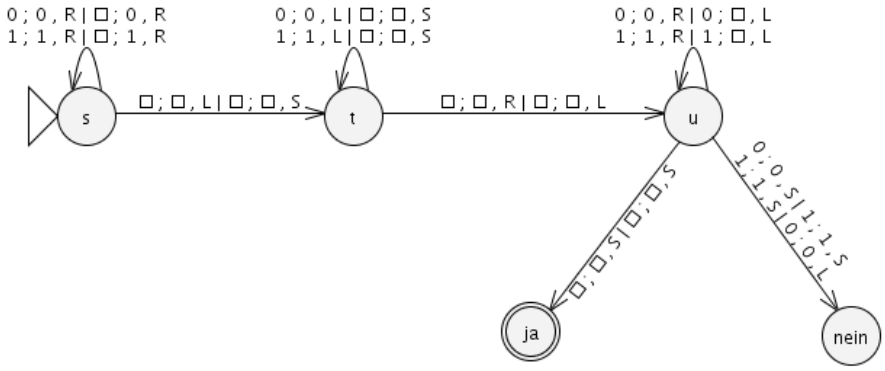

Abbildung 3.2: Turingmaschine für Palindrom-Entscheider

	Zustand	Band 1	Band 2	Regel			
1:	s	0	□	(	s,	$0, \rightarrow$,	$0, \rightarrow$)
2:	s	1	□	(	s,	$1, \rightarrow$,	$1, \rightarrow$)
3:	s	□	□	(	t,	$\square, \leftarrow$,	$\square, -$)
4:	t	0	□	(	t,	$0, \leftarrow$,	$\square, -$)
5:	t	1	□	(	t,	$1, \leftarrow$,	$\square, -$)
6:	t	□	□	(	u,	$\square, \rightarrow$,	$\square, \leftarrow$)
7:	u	0	0	(	u,	$0, \rightarrow$,	$\square, \leftarrow$)
8:	u	1	1	(	u,	$1, \rightarrow$,	$\square, \leftarrow$)
9:	u	0	1	(	$nein$,	$0, -$,	$1, -$)
10:	u	1	0	(	$nein$,	$1, -$,	$0, -$)
11:	u	□	□	(	ja,	$\square, -$,	$\square, -$)

3.3.2 Erweiterte Church'sche These

Algorithmen lassen sich oft einfacher auf einer k-Band-Turingmaschine darstellen als auf einer 1-DTM. Es stellt sich nun die Frage ob die k-DTM *wirklich* mächtiger ist als die 1-DTM. Intuitiv würde man erwarten, dass dies der Fall ist. Überraschenderweise kann aber gezeigt werden, dass jedes k-DTM Programm auf einer 1-DTM simuliert werden kann und, dass der Zeitverlust maximal quadratisch ist (der Platzbedarf bleibt sogar unverändert). Eine ähnliche Aussage kann für jedes bekannte Prozessmodell getroffen werden. Dies wird in der *erweiterten Church'schen These* verdeutlicht, die besagt:

Jedes sinnvolle Prozessmodell kann effizient auf einem Standardmodell (Turing-Maschine, Registermaschine, ...) simuliert werden.

Effizient heißt, dass der Zeitverlust maximal polynomiell ist. Diese These ist für alle *bekannten* Prozessmodelle bewiesen!

4 Die Klasse P

Florian Burgstaller, Andreas Derler, Gabriel Schanner

4.1 Äquivalenz von RM und TM

Laut der erweiterten Church'schen These kann jedes sinnvolle Prozessmodell effizient (Zeitverlust maximal polynomiell) auf einem anderen Standardmodell simuliert werden. So lässt sich auch eine Registermaschine auf einer Turingmaschine simulieren und umgekehrt.

Satz 4.1. *Zu jeder Registermaschine $\mathcal{R}$ gibt es eine Turingmaschine $\mathcal{T}$, sodass für die jeweils berechneten (partiellen) Funktionen gilt:*

$$f_{\mathcal{T}}(bin(x_1)\#bin(x_2)\#\ldots\#bin(x_k)) = bin(f_{\mathcal{R}}(x_1,\ldots,x_k)).$$

Satz 4.2. *Zu jeder Turingmaschine $\mathcal{T}$ (mit Ausgabealphabet $\{0,1\}$) gibt es eine Registermaschine $\mathcal{R}$, sodass für die jeweils berechneten (partiellen) Funktionen gilt:*

$$f_{\mathcal{R}}(w_1,w_2,\ldots,w_{|w|}) = bin^{-1}(f_{\mathcal{T}}(w))$$

4.1.1 Simulation RM durch DTM

Als Beweis zu Satz 1 wird nun gezeigt wie sich eine Registermaschine auf eine 4-Band DTM abbilden lässt. Dabei ist folgendes zu beachten:

- Die Registermaschine besitzt ein Programm mit p Befehlen
- und einen Speicher der aus den Registern $r_0, ..., r_m$ besteht.
- Die 4-Band DTM besitzt ein Eingabealphabet $\Sigma = \{0, 1, \#\}$
- und ein Bandalphabet $\Gamma = \{0, 1, \#, \square\}$, somit werden alle Werte binärkodiert gespeichert.
- Q_0-Zustände übersetzen Eingabe(DTM Band 1) in Registerstruktur(DTM Band 2)
- Q_{p+1}-Zustände übertragen Ergebnis r_0 in die Ausgabe (DTM Band 3)
- $Q_i, 1 \leq i \leq p$ simuliert Befehl i

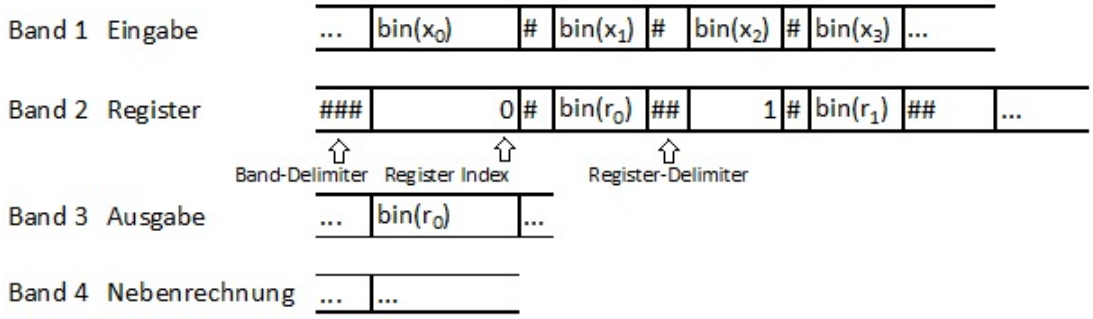

Abbildung 4.1: Simulation RM durch DTM - Bandbelegung

Die DTM arbeitet jeden Befehl der simulierten RM ab, indem jeder einzelne Befehl als aufeinanderfolgende Menge an Turingmaschinen Befehlen abgebildet wird. Dabei wird pro Befehl eine Menge an DTM Zuständen($Q_i = \{q_1 \dots q_b\}$) nötig sein um die jeweilige Funktionalität abzubilden. Bevor der eigentliche RM Befehlssatz ausgeführt werden kann, muss jedoch zunächst der Input vom Eingabeband (DTM Band 1) gelesen und das Registerband(DTM Band 2) befüllt werden. Die dafür benötigten DTM Zustände befinden sich der Zustandsmenge Q_0. Am Ende der Simulation muss noch die Ausgabe, welche sich in r_0 (in der k-Band DTM kodiert auf Band 2 gespeichert) befindet, auf das Ausgabeband (DTM Band 3) geschrieben werden. Für diesen Vorgang werden wieder DTM Zustände benötigt, die in der Menge Q_{p+1} zusammengefasst werden.

4.1.1.1 Beispiel DTM-Progamm für RM Befehl

Zur Verdeutlichung der Funktionsweise wird nun die Abarbeitung eines RM Beispiel Befehls `274: ADD *24`$\Longrightarrow Q_{274}$ erläutert. Dieser Befehl benutzt indirekte Adressierung und bewirkt die Addition des Wertes in r_0 mit dem Wert r_x, wobei x der Wert des Registers r_{24} ist. Das Ergebnis wird in r_0 gespeichert. Zur Verwirklichung dieses RM Befehls werden nun folgende Aktionen getätigt:

- Suche `##11000#`x`#` ($bin(24) = 11000$) auf dem Registerband (DTM Band 2).
- Kopiere x vom Registerband (DTM Band 2) auf das Nebenrechnungsband (DTM Band 4)
- Suche `##`x`#`y`#` auf dem Registerband (DTM Band 2)
- Kopiere y vom Registerband (DTM Band 2) auf das Nebenrechnungsband (DTM Band 4)
- Den Zeiger des Registerbands (DTM Band 2) auf das Ende des Akkumulators (r_0) stellen
- Bitweise Addition des Akkumulators zum Wert der auf dem Nebenrechnungsband (DTM Band 4) steht.
- Wert des Akkumulators durch den Wert des Nebenrechnugnsbandes (DTM Band 4) ersetzen.

Zur Verdeutlichung der Umsetzung einer obig genannten Aktion in ein k-Band DTM Programm, wird hier nun aufgezeigt wie die zweite Aktion beispielsweies als Zustand q_{274_2} implementiert werden könnte. Hierbei gelten folgenden Annahmen:

- Das Symbol unter dem Kopf auf Band 1 und Band 3 ist "#" wenn das erste mal in den Zustand q_{274_2} gewechselt wird.
- In Zustand q_{274_2} wird auf Band 1 und Band 3 stets weiterhin "#" geschrieben, deshalb werden diese Bänder in der folgenden Befehlstafel vernachlässigt.
- Das Nebenrechnungsband (DTM Band 4) ist gefüllt mit Blank Symbolen (□).
- Der Kopf auf Band 2 befindet sich am linkesten bit des binärkodierten x Wertes.

Zustand	B-2 Read	B-2 Write	Kopf-bewegung B2	B-4 Read	B-4 Write	Kopf-bewegung B4	nächster Zustand
q_{274_2}	0	0	$\rightarrow$	□	0	$\rightarrow$	q_{274_2}
q_{274_2}	1	1	$\rightarrow$	□	1	$\rightarrow$	q_{274_2}
q_{274_2}	#	#	-	□	□	$\leftarrow$	q_{274_3}

Auch die restlichen, oben aufgezeigten, Aktionen können auf ähnliche Weise umgesetzt werden, sodass die Zustandsmenge Q_{274} aus 27 Einzelzuständen besteht. Somit ist auch ersichtlich, dass jeder RM-Befehl mit einer endlichen Menge an DTM-Zuständen verwirklicht werden kann.

4.1.2 Simulation DTM auf RM

Bei der Simulation einer DTM durch eine RM stellt sich zunächst die Frage wie die Bänder dargestellt werden. Folgend werden 2 Lösungsvarianten vorgestellt.

4.1.2.1 Variante 1

Bei dieser Variante wird für jedes Bandfeld ein Register verwendet, startend bei r_{10} (die ersten Register $r_1 \dots r_9$ werden als Arbeitsregister frei gehalten). Zu beachten ist auch, dass das Bandalphabet der DTM in die natürlichen Zahlen kodiert werden muss. Weiters gilt folgendes:

- Die Bandfelder werden numeriert: $\dots, F_{-2}, F_{-1}, F_0, F_1, F_2, \dots$
- Die Kopfposition wird in r_1 festgehalten, der Inhalt des Bandfeldes, auf das der Zeiger zeigt, ist dementsprechend in r_{r_1}.
- Der aktuelle Zustand wird in r_2 festgehalten. Weiters ist zu beachten, dass die Zustände der simulierten DTM zunächst in natürliche Zahlen kodiert werden müssen, da die Register der RM natürliche Zahlen speichern.
- Die Umrechnung Bandfeld auf Register geschieht wie folgt: $F_{-k} \rightarrow r_{10+2k}$, $F_0 \rightarrow r_{10}$, $F_k \rightarrow r_{10+2k-1}$, somit alternieren die Felder mit positivem(ungerade Register) und negativem(gerade Register) Index.
- Im Registerprogramm müssen jetzt mehrere Fallunterscheidungen getroffen werden um für die korrekte Navigation auf dem simulierten Turingmaschinenband zu sorgen. Es muss ein Δ gesetzt werden, da aufeinanderfolgende Bandindizes, abgebildet in Registern, nicht direkte Nachbarn sind. Wenn man sich auf dem Band nach Rechts bewegen möchte, setzt man $\Delta = 2$, ansonsten $\Delta = -2$.

- Nachdem nun bekannt ist, in welche Richtung man den Kopf bewegen möchte, sorgt der folgende Code für die korrekte Berechnung der Kopfposition.
  `IF` r_1 `mod 2=1 THEN` r_1=r_1+Δ `ELSE` r_1=r_1-Δ
  `IF` r_1`=8 THEN` r_1`=11 ELSEIF` r_1`=9 THEN` r_1`=10`
  Hierbei wird in der ersten Zeile die neue Kopfposition gesetzt, je nachdem ob man sich zur Zeit auf einer negativen oder positiven (inklusive des Index 0) Kopfposition befindet. Die zweite Zeile kümmert sich um die Navigation von Index 0 zu Index 1 und umgekehrt. (Beispiel: Befindet man sich auf Positon 0 (Registerindex 10) und will sich zur Position 1 (Registerindex 11), also nach Rechts, bewegen, so wird in der ersten Zeile der Wert von r_1 auf 8 gesetzt. Die zweite Zeile korrigiert diesen Wert auf den Index 11.)
- Die Werte für Δ, r_2 und r_{r_1} müssen daher bei jedem Zustandsübergang der simulierten DTM über `IF`-Tabellen bestimmt werden.
- Ein Nachteil bei dieser Variante ist die Notwendigkeit der indirekten Adressierung.

Der Programmablauf könnte dann folgenderweise geschehen:
Init :
$position = 3 \quad (r_1 = 3)$
$zustand = q_0 \quad (r_2 = 0)$
Loop :
$IF\ zustand == q_0\ AND\ bandq == b_1\ THEN$
$zustand = q_i \quad (r_2 = i)$
$bandq = b_j \quad (*r1 = j)$
$position = position + 1 \quad (r_1 = r_1 + 1)$
$IF\ zustand == q_1\ AND\ bandq == b_1\ THEN$
...

4.1.2.2 Variante 2

Alternativ kann man auch 2 Stacks verwenden, die den Bandinhalt links und rechts vom Kopf darstellen und in jeweils einem Register gespeichert werden. Dies wird ermöglicht durch die Eigenschaft der RM-Register unendlich große Werte speichern zu können, wodurch es möglich ist einen Stack kodiert in einem Register zu speichern. Das Bandalphabet wird dabei binär kodiert. Die RM benötigt nur vier Register:

- Stack L in r_3 ist der Bandinhalt links vom Kopf
- Stack R in r_4 ist der Bandinhalt rechts vom Kopf
- Symbol unter dem Kopf wird in r_2 festgehalten.
- Der aktuelle Zustand wird in r_1 festgehalten.

Bei der DTM Konfiguration $\beta_m \dots \beta_0 \mathbf{q} \alpha_0 \dots \alpha_n$ sind die Register wie folgt befüllt:

- $r_3 = \beta_0 + \beta_1 * |\Gamma| + \dots + \beta_m * |\Gamma|^m$
  $r_4 = \alpha_1 + \alpha_2 * |\Gamma| + \dots + \alpha_n * |\Gamma|^{n-1}$
  $r_2 = \alpha_0 \qquad r_1 = \mathbf{q}$
  Wobei $|\Gamma|$ den nötigen Offset liefert, sodass sich die gespeicherten Werte nicht überschneiden.

(Beispiel: Direkt links vom Kopf befindet sich der Wert 3 und eins weiter links der Wert 1. Speichert man nun sämtliche Werte binärkodiert, so wäre der Inhalt des Registers r_3 1101.)

- Wandert der Kopf nun nach Rechts, ändert sich der Inhalt von r_3: $r_3 = r_3 * |\Gamma| + r_2; r_2 = r_4 \texttt{ mod } |\Gamma|;\ r_4 = r_4 \texttt{ div } |\Gamma|$
  Ins Register r_3 wird sozusagen der Wert der aktuellen Kopfposition "gepusht" während vom Register r_4 das neue Symbol unter dem Kopf "gepopt" wird.
- Wandert der Kopf nun nach Links, ändert sich der Inhalt von r_4, analog zu obrigem Beispiel: $r_4 = r_4 * |\Gamma| + r_2; r_2 = r_3 \texttt{ mod } |\Gamma|;\ r_3 = r_3 \texttt{ div } |\Gamma|$

Der folgende RM Programmcode gibt an wie die Zustandsübergänge realisiert werden.

```
Loop:
IF r1 = q1 ∧ r2 = α1 THEN r1 = q'1; r2 = α'1;MoveR;
ELSE IF r1 = q2 ∧ r2 = α2 THEN r1 = q'2; r2 = α'2;MoveL;
ELSE IF r1 = q3 ∧ r2 = α2 THEN r1 = q'3; r2 = α'3;MoveL;
ELSE IF ...
ELSE GOTO Rejected
IF r1 = qF1 ∨ ... ∨ r1 = qFk THEN GOTO ACCEPTED;
GOTO Loop
```

Beachtenswert bei dieser Variante ist auch die Tatsache, dass hier nicht auf indirekte Adressierung zurückgegriffen wurde. Dies erinnert an die Tatsache, dass eine RM mit indirekter Adressierung gleich mächtig ist wie ohne.

4.1.3 Kosten der Simulation

Bei einer RM mit Komplexität $T_{\mathcal{R}}(n)$ folgt für $T_{\mathcal{T}}(n)$ der äquivalenten DTM

$$T_{\mathcal{T}}(n) = \mathcal{O}\left(T^2_{\mathcal{R}}(n)\right)$$

Bei einer DTM mit Komplexität $T_{\mathcal{T}}(n)$ folgt für $T_{\mathcal{R}}(n)$ der simulierenden RM

$$T_{\mathcal{R}}(n) = \mathcal{O}\left(T^2_{\mathcal{T}}(n)\right)$$

4.2 Sprachprobleme

4.2.1 Problemarten

- *Konstruktionsprobleme*
  Zu einer bestimmten Eingabe, soll mithilfe einer Bewertungsfunktion f aus dem Lösungsraum, eine optimale Lösung gefunden werden.

- *Funktionsberechnungen*
  Zu einer bestimmten Eingabe x, soll mithilfe einer Berechnungsfunktion $f(x)$ eine eindeutige Lösung generiert werden.
- *Entscheidungsprobleme*
  Zu einer bestimmten Eingabe, soll »JA« oder »NEIN« zurückgegeben werden. In der theoretischen Informatik haben die Entscheidungsprobleme die größte Bedeutung, da viele Problemstellungen darauf reduziert werden können. Dabei ist zu beachten, dass die Eingabecodierung ein Teil der Problemdefinition ist.

4.2.2 Formale Sprachen

Eine Formale Sprache ist die Menge von Worten, welche aus den Symbolen eines Alphabets aufgebaut werden können. Ein Alphabet ist definiert als ein Zeichenvorrat. Jedes Entscheidungsproblem lasst sich als formale Sprachen darstellen. Hierbei wird untersucht, ob ein gegebenes Wort in der jeweiligen Sprachen liegt oder nicht.

4.2.3 Chomsky-Hierachie

Grammatiken welche formale Sprache erzeugen, können in vier Typen eingeteilt werden, je nach dem, welchen Einschränkungen deren Produktion unterliegt. Eine Sprache von Typ-0 ist in ihrer Produktion uneingeschränkt, je höher die Stufe wird, desto mehr Einschräkungen kommen hinzu. Das bedeutet, eine Typ-0 Grammatik kann auch Typ-1 Sprachen and alle darüber erzeugen. Formale Grammatiken werden in der Form $G = (S, P, V, \Sigma)$ notiert, wobei aus S dem Startsymbol und der Regelmenge P neue Zeichenfolgen erzeugt werden. Dieses Vokabular V besteht aus Nichtterminalen und Terminalen Σ, Nichtterminalsymbole können mithilfe von P weiter abgeleitet werden.

- Typ-0 Sprachen sind Sprachen welche von einer Turing Maschiene akzeptiert werden. Deren Produktion sieht folgendermaßen aus:
  $\alpha \rightarrow \beta : \alpha, \beta \in V$
- Typ-1 Grammatiken oder auch *Kontextsensitive Grammatiken* haben die zusätzliche Einschräkung, dass der rechte Teil der Produktion mindestens gleich lang ist, wie der Linke. Auch diese Sprachen können nur mit Turingmaschienen erkannt werden.
  $\alpha \rightarrow \beta : |\alpha| \leq |\beta|, \alpha, \beta \in V$
- Typ-2
  $\alpha \rightarrow \beta : |\alpha| \leq |\beta|, \alpha \in V_N, \beta \in V$
  V_N ist die Menge der Nichtterminalsymbole $V_N = \{V \setminus \Sigma\}$
  Typ-2 oder auch *Kontextfreie Sprachen* können von *Kellerautomaten* erkannt werden, dabei handelt es sich um einen endlichen Automaten welcher mittels push und pop Zugriff auf einen Stack hat.
- Typ-3 oder auch *reguläre Grammatiken* sind definiert als Grammatiken deren rechte Seite der Produktion genau ein Terminal und ein Nichtterminal beinhaltet. Es existieren zwei Arten von Typ-3 Grammatiken, linksreguäre und rechtsreguläre, diese beiden Typen unterscheiden sich darin, ob das Terminal oder das Nichttermial zuerst auftritt. Reguläre Sprachen sind genau jene

Sprachen welche von *endlichen Automaten* erkannt werden können.
$\alpha \to \beta : |\alpha| \leq |\beta|, \alpha \in V_N, \beta \in V$

linksreguläre Grammatik: β hat die form bB oder b, wobei $b \in V_T$
rechtsreguläre Grammatik: β hat die form Bb oder b, wobei $b \in V_T$

4.2.4 Turing-Berechenbarkeit

Rekursiv aufzählbare Sprachen sind definiert als die Menge der Sprachen L für die eine Turingmaschiene entscheiden kann ob ein gegebenes Wort in dieser Sprache liegt. Das bedeutet, die Turingmaschine hält in einem Haltezustand, genau dann wenn das Eingabewort w in L liegt. Es ist jedoch nicht garantiert dass die TM hält wenn das Wort w *nicht* in der Sprache liegt.

Satz 4.3. *Eine Sprache $L \subset \Sigma^*$ wird von einer DTM akzeptiert, gdw. $\exists$ DTM $\mathcal{T}$: $w \in L \Leftrightarrow f_{\mathcal{T}}(w) = JA$*

Rekursive Sprachen sind definiert als die Menge der Sprachen für die eine Turingmaschiene entscheiden kann ob ein gegebenes Wort in dieser Sprache liegt oder nicht. Im Gegensatz zu rekursiv aufzählbaren Sprachen, wird eine Entscheidung in endlicher Zeit berechnet.

Satz 4.4. *Eine Sprache $L \subset \Sigma^*$ wird von einer DTM entschieden, gdw. $\exists$ DTM $\mathcal{T}$: $w \in L \Leftrightarrow f_{\mathcal{T}}(w) = JA \wedge w \notin L \Leftrightarrow f_{\mathcal{T}}(w) = NEIN$*

4.3 Die Zeit-Komplexitätsklasse DTIME

Definition 4.1. *DTIME(t(n)) ist die Menge aller Sprachen, die von einer $\mathcal{O}(t(n))$-zeitbeschränkten deterministischen Turingmaschine entschieden werden können.*

$$DTIME(t(n)) = \{A \subseteq \Sigma^* \mid \exists DTM\mathcal{T} : f_{\mathcal{T}} = f_A \wedge T_{\mathcal{T}}(n) = \mathcal{O}(t(n))\}$$

4.4 Die Klasse P

Definition 4.2. *Die Klasse P ist die Menge aller Sprachen, die von einer deterministischen Turingmaschine in polynomieller Zeit entschieden werden können.*

$$P = \bigcup_{k \in \mathbb{N}} DTIME(n^k)$$

Oder einfacher: Die Menge aller Sprachen, die von einer *polynomiell zeitbeschränkten DTM* entschieden werden können. Die Komplexitätsklassen sind zwar über die TM definiert, dennoch sind sie anhand der erweiterten Church'schen These, unabhängig vom Maschinenmodell. Dies bedeutet, sobald die Sprache mit einer polynomiell zeitbeschränkten DTM bewiesenermaßen entschieden werden kann, gilt dieses auch für jedes beliebig andere Maschinenmodell. Algorithmen, welche Teil dieser Klasse P sind, werden *effizient* bezeichnet.

4.5 Problembeispiele

4.5.1 REACH

Sobald ein konkreter Algorithmus mit polynomiell beschränkter Laufzeit zu einem vorhandenen Problem gefunden wird, kann dieses Problem der Klasse P zugeordnet werden. Ein Beispielalgorithmus, welcher das Graph-Erreichbarkeitsproblem in polynomieller Laufzeit entscheidet, wurde bereits in Kapitel 1.1 inklusive Beweis vorgestellt. Somit gilt `REACH` $\in$ P.

4.5.2 RELPRIME

Gegeben: 2 Zahlen x, y
Gesucht: Haben x und y einen gemeinsamen Primfaktor?

Definition 4.3. *RELPRIME* $= \{\langle x, y\rangle \mid \nexists k \geq 2 \in \mathbb{N} : k \,|\, x \vee k \,|\, y\}$

Pseudocode:

```
function RELPRIME(x,y):

a = GGT(x,y)

Wenn a == 1

    THEN RELPRIM

Sonst

    THEN NICHT


function GGT(x,y):

solange y > 0

    r = x  mod y

    x = y

    y = r
```

Mittels Euklid'schen Algorithmus wird der größte gemeinsame Teiler der Eingabewerte x und y berechnet und falls dieser gleich 1 ist, sind die beiden Werte x, y teilerfremd und somit relativ prim zueinander. Ansonsten wurde ein Teiler > 1 gefunden und somit sind x,y nicht teilerfremd.

Nun soll anhand des Algorithmus die Zugehörigkeit zur Klasse P gezeigt werden. Dazu muss die Eingabecodierung einer Turingmaschine in Form einer binären Darstellung beachtet werden. Eine Dezimalzahl n verfügt somit über eine Komplexität von 2^n und um eine polynomielle Laufzeit zu erhalten, dürfen nur logarithmisch viele Schleifendurchläufe ausgeführt werden. Da die Anzahl der Schleifendurchläufe von beiden Inputgrößen x,y abhängt, darf diese nicht größer als $L(max(x, y))$ sein.

Die Funktion `RELPRIME(x,y)` verfügt, abgesehen vom Aufruf der Funktion `GGT(x,y)`, über eine konstante Zeitkomplexität und erfüllt somit die Bedingung einer polynomiellen Laufzeit, sofern auch `GGT(x,y)` diese erfüllt. Bei jedem Schleifendurchlauf der Funktion `GGT(x,y)` wird der Rest r von x,y berechnet. Die Größenabhängigkeit von r kann durch 2 Möglichkeiten beschrieben werden:

1. $\frac{x}{2} \geq y$:
   $r = x \mod y \Rightarrow r < y < \frac{x}{2}$

2. $\frac{x}{2} < y$:
   $r = x \mod y = x - y < \frac{x}{2}$

Der Abbruch der Schleife bei maximalen Schleifendurchläufen erfolgt, wenn $x = y = 1$. Dieses wird nach k Schleifendurchläufen erreicht, wobei sich k aufgrund von $r < \frac{x}{2}$ auf $2^k = max(x, y)$ beschränkt. Dadurch werden maximal $L(max(x, y))$ Schleifendurchläufe ausgeführt, wodurch das Programm polynomiell zeitbeschränkt ist.

4.5.3 `CVP` (Circuit Value Problem)

Jede Boolsche Funktion lässt sich als Baum darstellen. Wobei Variablen und Boolsche Operatoren die Knoten belegen, welche unter Berücksichtigung der Klammernsetzung miteinander verknüpft werden. Verknüpfungen zweier Variablen $(\wedge, \vee)$ verfügen über zwei Eingänge und einen Ausgang, im Gegensatz zu Verknpüfungen einzelner Variablen $(\neg)$, welche über einen einzelnen Ein- und Ausgang verfügen. Anhand des Baumes, kann bei jeglicher gegebener Variablenbelegung das Problem *effizient*, also in P, beantworet werden. Zu beachten ist, dass dies nur für eine konkrete gegebene Variablenbelegung gilt und nicht für das Finden einer solchen Belegung.

Gegegen: Boolscher Schaltkreis C, Eingangsbelegung x
Gesucht: Liefert der (einzige) Ausgang von C eine 1?

Definition 4.4. $\mathit{CVP} = \{\langle C, x\rangle \mid C(x) = 1\}$

Beim Circuit Value Problem soll zu einem gegeben Boolschen Schaltkreis C mit einer Eingangsbelegung x entschieden werden, ob dieser einen Ausgang von 1 liefert. Wobei zu beachten ist, dass die Kodierung von C keine Makrobildung oder rekursive Definition zulässt.

Beispiel 4.1

Gegeben sei die Boolsche Funktion

$$C(x_1, x_2, x_3) = (x_3 \wedge \neg((x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)) \vee (x_2 \wedge x_3)),$$

mit Eingabe $x_1, x_2, x_3 \in \{0, 1\}$. Der Boolsche Schaltkreis der diese Funktion realisiert ist in Abbildung 4.2 dargestellt.

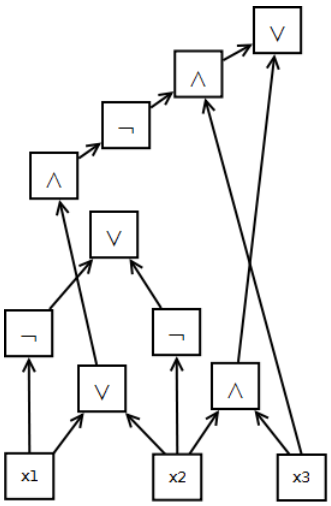

Abbildung 4.2: Boolscher Schaltkreis

4.5.4 PRIMES

Das Problem PRIMES entscheidet für eine gegebene Zahl x, ob diese eine Primzahl ist. Erst im Jahr 2004 konnte gezeigt werden, dass dieses Problem in der Klasse P liegt.[1]

4.5.5 Probleme (wahrscheinlich) nicht mehr in P

Diverse Probleme können bisher weder der Klasse P zugeordnet werden, noch kann ausgeschlossen werden, dass sie darin liegen. Da bisher noch kein Algorithmus in polynomiell beschränkter Laufzeit gefunden wurde, nimmt man jedoch an, dass sie außerhalb der Klasse P liegen. Beispiele dafür wären:

1. Hat Graph G einen Hamilton-Kreis?
2. Hat die Zahl x einen Teiler kleiner als y?
3. Hat Graph G eine Clique der Größe k?

Allerdings ist Beispielsweise das Problem, ob ein Graph eine Clique der Größe 4 hat, in der Klasse P, da ein festes $k = 4$ gewählt wurde und dadurch die Anzahl der möglichen Cliquen mit $\binom{m}{4} = \mathcal{O}(m^4)$ beschränkt ist.

[1] Agrawal, Kayal und Saxena, »Primes in P«.

5 Nichtdeterminismus

Florian Burgstaller, Andreas Derler, Gabriel Schanner

5.1 Motivation

Von sehr vielen Problemen ist nicht bekannt, ob sie in polynomieller Zeit (bzw. effizient) lösbar sind. Relevante Probleme dieser Art wären zum Beispiel das Traveling Salesman Problem (`TSP`) und das Erfüllbarkeitsproblem von Boolschen Funktionen (`SAT`). Mit nur einer kleinen Erweiterung des Maschinenmodells können nun allerdings polynomielle Algorithmen gefunden werden. Diese Erweiterung des Maschinenmodells entspricht der *nichtdeterministischen Turingmaschine*.

5.2 Nichtdeterminismus

Analog zur deterministischen Turingmaschine (DTM), bei welcher ein Programm einem *deterministischen* endlichen Automaten entspricht, gleicht bei einer Nichtdeterministischen Turingmaschine (NTM) das Programm einem *nicht − deterministischen* endlichen Automaten. Diese unterscheiden sich durch die Anzahl der Zustandsübergänge pro Symbol. Ein deterministischer endlicher Automat verfügt in jedem Zustand über maximal einen Zustandsübergang pro Symbol. Hingegen kann ein nichtdeterministischer endlicher Automat in jedem Zustand über *mehrere* Zustandsübergänge pro Symbol verfügen.

5.2.1 Nichtdeterministische Turingmaschine

Definition 5.1. *Eine k-Band-NTM ist ein Tupel* $\mathcal{T} = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ *dessen Komponenten wie für eine k-Band-DTM definiert sind, mit Ausnahme von*

$$\delta : Q \times \Gamma \mapsto \mathcal{P}(Q \times \Gamma \times \{\leftarrow, -, \rightarrow\}).$$

Anstatt eindeutig *einen* nächsten Schritt zu definieren, repräsentiert $\delta(q, b)$ nun die *Menge der nunmehr möglichen* nächsten Schritte. Dies hat zur Folge, dass die Konfigurationsrelation $\underset{\mathcal{T}}{\rightarrow}$ erweitert wird und dadurch die Konfigurationenfolge zu einem Berechnungsbaum wird.

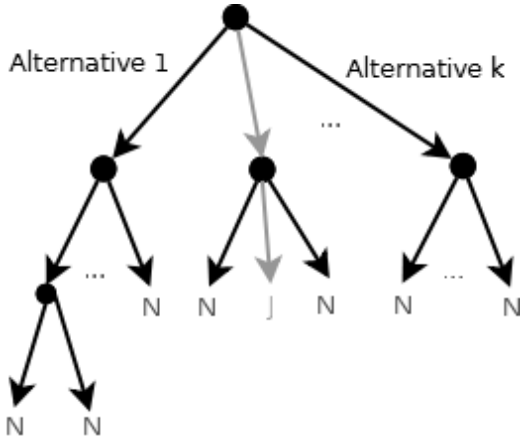

Abbildung 5.1: Berechnungsbaum NTM

Die Konfigurationenfolge einer NTM wird zu einem Berechnungsbaum, wobei die Nichtdeterministische Turingmaschine nun, wie ein allwissendes Orakel, bei jeder Abzweigung immer den »richtigen« Pfad des Baumes wählt.

5.2.2 Entscheidungsprobleme

Die algorithmische Leistung der NTM besteht darin, am Ende des Pfades zu akzeptieren oder zu verwerfen. Eine NTM, die eine Sprache entscheidet, hat ausschließlich Pfade die akzeptieren oder verwerfen (keine die divergieren). Die Entscheidung der NTM heißt »JA«, falls es *zumindest einen* akzeptierenden Pfad gibt. Die Entscheidung der NTM heißt »NEIN«, wenn es *keinen* akzeptierenden Pfad gibt.

5.2.3 Laufzeitmessung einer NTM

Definition 5.2. *Sei $\mathcal{T}$ eine NTM die eine Sprache entscheidet. Die Laufzeit $t_{\mathcal{T}}$ von $\mathcal{T}$ ist die maximal auftretende Tiefe aller Pfade durch den Berechnungsbaum von $\mathcal{T}$.*

Somit beziehen sich die Zeitkosten einer NTM stets auf die maximale Tiefe aller Pfade des Berechnungsbaumes. Daher wird die Laufzeit stets anhand des längsten Pfades (akzeptierend oder nicht) gemessen, auch wenn es davor schon akzeptierende Berechnungspfade gibt.

5.3 Die Klasse NP

5.3.1 Verifizierer

Für viele Algorithmen ist es nicht bekannt, ob eine Lösung effizient berechenbar ist, jedoch ist es oft trivial für eine gegebene Lösung zu entscheiden ob sie korrekt ist oder nicht. Den Vorgang des Überprüfens einer Lösung, nennt man Verifizierung.
Def. Ein Verifizierer für eine Sprache L ist ein Turingmaschine $\mathcal{T}$, wobei
L $= \{w \in \Sigma^* \mid \exists c \in \Sigma^* : \mathcal{T}$ akzeptiert $< w, c >\}$ c wird als Zertifikat bezeichnet und wird als Zusatzinformation benutzt um w zu verifizieren (es wird untersucht ob w in L liegt). Der Zeitbedarf der Turingmaschine L wird in Abhängigkeit von w berechnet.

5.3.2 Beispiel: Traveling Salesman Problem

Gegeben ist eine Liste von Städten und die Kosten, von einer Stadt zu einer Anderen zu reisen. Es soll berechnet werden, ob eine Reise, in der jede Stadt genau ein mal besucht wird, für ein gegebenes Budget möglich ist. Dabei wird ersichtlich, dass zwar keine bekannte effiziente Lösung für `TSP` existiert, jedoch das überprüfen einer bekannten Lösung trivial ist.
Idee: Nichtdeterministische Auswahl von Städten und anschließende überprüfung durch einen Verifizierer.

1. Nichtdeterministische Auswahl von Städten. (Es wird eine mögliche Abfolge geraten) Die Städte werden in der angenommenen Reihenfolge auf Band 2 geschrieben.
2. Berechnung der Kosten, der Rundreise auf Band 2.
3. Wenn die berechneten Kosten im Budget liegen, wird akzeptiert, sonst verworfen.

Zustandübergang der NTM aus Schritt 1:
δ: $(Q_{\#}, \#)$ Suche die nächste Stadt, und kopiere diese auf Band 2.
Nach Beedigung von Schritt 1, befindet sich auf Band 2 eine Auswahl von n Städten, diese kann im Bezug auf das Inputband in polynomieller Zeit verifiziert werden, da sich jede Stadt genau ein mal auf dem Band befindet und lediglich überprüft werden muss ob jede Stadt besucht wurde und die Summe der Kosten im Budget liegt.

5.3.3 Die Klasse NP

Im vorigen Beispiel wurde gezeigt, dass `TSP` in polinomieller Zeit verifiziert werden kann. Ein polynomieller Verifizierer kann nur auf Zertifikaten mit polynomieller Länge zugreifen(in w), da dies die maximale Zeitgrenze ist. Daraus lässt sich folgende Definition ableiten

Definition 5.3 (Die Komplexitätsklasse NP)**.** NP *ist die Klasse der Sprachen, welche in polynomieller Zeit verifiziert werden können.*

Eine alternative Definition ist über die Klasse NTIME möglich

Definition 5.4 (Die Komplexitätsklasse NTIME)**.** NTIME$(t(n))$ *ist die Menge aller Sprachen die von einer* $O(t(n))$*-zeitbeschränkten NTM entschieden werden können.*

$$\text{NTIME}(t(n)) = \{A \subseteq \Sigma^* \mid \exists NTM\, \mathcal{T} : f_{\mathcal{T}} = f_A \wedge T_{\mathcal{T}}(n) = O(t(n))$$

Die Klasse NP ist dann die Menge aller Sprachen, die von einer NTM in polynomieller Zeit entschieden werden können.

$$\text{NP} = \bigcup_{k \in \mathbb{N}} \text{NTIME}(n^k)$$

Es kann gezeigt werden, dass eine NTM in exponentieller Zeit (vielleicht schneller?) und in quadratischem Platz simuliert werden kann.

5.3.4 Beispiel CLIQUE

Um zu zeigen, dass ein Problem in NP liegt, kannn eine polynomielle NTM welche das Problem löst, angegeben werden. (Alternativ kann auch ein polynomieller Verifizierer welcher das Problem überprüft, konstruiert werden.)
geg: ungerichteter Graph G = (V, E), Zahl k
V = $\{v_1, v_2, v_3, ..., v_m\}$
E = $\{(v_{i_1}, v_{i_2}), ...\}$
ges: existiert im Graphen G eine Clique der größe k?
Beweis:
Die Turingmaschine $\mathcal{T}$ berechnet mit dem Input folgende Schritte:

1. Entscheide nichtdeterminisch, welche k Knoten aus G ausgewählt werden.
2. Überprüfe ob die ausgewählten Knoten, alle in G enthalten und miteinander verbunden sind.
3. Wenn ja, wird akzeptiert, sonst verworfen.

Alternativer Beweisansatz:
Wie man sieht, ist der erste Schritt, eine nichtdeterminische Entscheidung, Schritt 2 und 3 dagegen, können deterministisch berechnet werden. Deshalb könnte man den Output der NTM aus Schritt 1, als Zertifikat benutzen und dieses von einer DTM (Überprüfung von Schritt 2 und 3) verifizieren lassen.

5.4 Das P vs. NP Problem

Eines der nach wie vor größten ungelösten Probleme der theorethischen Informatik ist die Frage ob P = NP. Zur näheren Betrachtung der Relation von P und NP folgt nun eine kurze Zusammenfassung der Klassen P und NP, wobei »effizient« gleichgesetzt werden kann mit »in polynomieller Zeit«.

- P beschreibt die Klasse von Sprachen die effizient *entschieden* werden können.
- NP beschreibt die Klasse von Sprachen die effizient *verifiziert* werden können.

Hierbei ist zu beachten, dass die polynomielle Verifizierbarkeit ein scheinbar *mächtigeres* Konzept ist, als die polynomielle Entscheidbarkeit. Würde nur für ein NP-vollständiges Problem ein polynomieller Algorithmus entdeckt werden, so hätte dies zur Folge, dass alle in NP enthaltenen Probleme in polynomieller Zeit lösbar wären, womit P = NP gelten würde. Jedoch ist bis heute sowohl P = NP als auch P $\neq$ NP unbewiesen.

5.4.1 Hinweise auf P $\neq$ NP

Obwohl es zur Zeit nicht möglich erscheint den mathematischen Beweis zu erbringen, wird generell angenommen das P $\neq$ NP. Für viele Probleme in NP wird seit Jahrhunderten erfolglos nach effizienten Algorithmen gesucht. Weiters würde P = NP zu Konsequenzen führen würde, von denen ebenfalls angenommen wird, dass sie unwahrscheinlich sind.(z.B.: NP = CoNP) Somit kann davon ausgegangen werden, dass, wenn ein Problem als NP-vollständig bewiesen wird, es für die Lösung(Entscheidung) dieses Problems keinen polynomiellen Algorithmus gibt.

5.5 CoP und CoNP

Die Klassen CoP und CoNP beschreiben die komplementären Klassen zu P bzw. NP. Wir schreiben das Komplement einer Sprache L als $\bar{L}$. $\bar{L}$ ist die Sprache die durch Komplementbildung von L entsteht, d.h. die Menge aller Wörter die *nicht* in L liegen. Während angenommen wird, dass NP $\neq$ CoNP, so ist P abgeschlossen gegenüber dem Komplement. Dies bedeutet, dass, wenn L in P liegt, so liegt auch $\bar{L}$ in P.

5.5.1 Beweis CoP = P

Gegeben sei die Sprache $L \in$ P und die DTM $\mathcal{T}$ die L entscheidet. Nun simuliere man die DTM $\mathcal{T}$ mit einer zweiten DTM $\mathcal{T}'$. Die DTM $\mathcal{T}'$ wird adaptiert, sodass jeder akzeptierende Zustand von $\mathcal{T}$ ein verwerfender Zustand in $\mathcal{T}'$ ist. Analog dazu ist jeder verwefende Zustand in $\mathcal{T}$ ein akzeptierender Zustand in $\mathcal{T}'$. Da der Zeitverlust für diese Adaptierung konstant ist, liegen $\mathcal{T}$ und $\mathcal{T}'$ in derselben Komplexitätsklasse.

Für NTM ist der Beweis in dieser Form, wegen der Asymmetrie zwischen Akzeptieren und Verwerfen, nicht möglich. Würde eine NTM jeden verwerfenden Zustand einer anderen NTM akzeptieren bzw. jeden akzeptierenden Zustand der anderen NTM verwerfen, so würde, aufgrund der Definition einer NTM, diese NTM beim ersten verwerfenden Zustand der komplementären NTM sofort akzeptieren und somit die anderen Pfade vernachlässigen.

6 Strukturelle Komplexitätstheorie

Dominik Hirner, Philipp Kober

In der Komplexitätstheorie betrachtet man Probleme, welche in endlicher Zeit berechnet werden können. Bisher haben wir uns mit einigen Komplexitätsklassen auseinander gesetzt, für diese aber noch keine Einteilung getroffen. Dies geschied im folgenden Kapitel. So kann man zum Beispiel leichter Aussagen über die Effizienzschranken der Probleme treffen. Die Probleme werden anhand ihrer (gemeinsamen) Komplexitätseigenschaften (bzw. Platz- und Zeitkomplexität da Zeit und Platz die wichtigste Ressource eines Algorithmus darstellen) in eine einer sogenannte Komplexitätsklasse zugeordnet.

6.1 Platzkomplexität

Wie bereits oben erwähnt, ist die Platzkomplexität neben der Zeitkomplexität einer der wichtigsten Aspekte zur Einteilung von Problemen in Klassen (wobei die Platzkomplexität 'mächtiger' ist als diese Zeitkomplexität, da Platz wiederverwendet werden kann). Für das Messen des Platzes der gebraucht wird dient eine TM.

6.1.1 DSPACE, NSPACE

DSPACE ist die Menge der Sprachen, welche von einer O(s(n)) platzbeschränkten deterministischen Turingmaschine entschieden werden kann. Analog dazu ist NSPACE die Menge der Sprachen, welche von einer O(s(n)) platzbeschränkten nicht-deterministischen Turingmaschine entschieden werden kann. DTIME und NTIME sind gleich definiert, nur das die entscheidenden Turingmaschinen nicht platz- sondern zeitbeschränkt sind. Die formale Definition lautet:

$$\mathrm{DSPACE}(s(n)) = \left\{A \subseteq \Sigma^* \middle| \exists \mathrm{DTM}\, \mathcal{T} : f_{\mathcal{T}} = f_A \wedge S_{\mathcal{T}}(n) = \mathcal{O}(s(n))\right\}$$

$$\mathrm{NSPACE}(s(n)) = \left\{A \subseteq \Sigma^* \middle| \exists \mathrm{NTM}\, \mathcal{T} : f_{\mathcal{T}} = f_A \wedge S_{\mathcal{T}}(n) = \mathcal{O}(s(n))\right\}$$

6.1.2 häufig verwendete Platz- und Zeitkomplexitätsklassen

In der vorherigen Section haben wir die vier Sprachmengen DSPACE, NSPACE, DTIME und NTIME kennengelernt. Es gibt aber noch weitere Einteilungen in Klassen (Anmerkung: bei dieser Einteilung in Klassen braucht man einen gewissen Trade-off zwischen sehr fein und sehr grob granulierten Komplexitätsklassen, da viele Beiweise nur mit groben Klassen möglich sind). Sprachen können aufgrund ihrer Zeit- und Platzkomplexitätsklassen wie folgt eingeteilt werden.

deterministische Platzkomplexitätsklassen:

- $\mathrm{L} = \mathrm{DSPACE}(\log n)$
  L ist die Menge der Sprachen, die von einer deterministische Turingmaschine auf logarithmischen Platz entschieden werden kann.

- $\mathrm{PSPACE} = \bigcup_{k \in \mathbb{N}} \mathrm{DSPACE}(n^k)$ PSPACE ist die Menge der Sprachen, die von einer deterministische Turingmaschine auf polynomiellen Platz entschieden werden kann.

- $\mathrm{EXPSPACE} = \bigcup_{k \in \mathbb{N}} \mathrm{DSPACE}(2^{n^k})$ EXPSPACE ist die Menge der Sprachen, die von einer deterministische Turingmaschine auf exponentiellen Platz entschieden werden kann.

Nichtdeterministische Komplexitätsklassen:

- $\mathrm{NL} = \mathrm{NSPACE}(\log n)$
  Gleich wie L, nur das es von einer Nichtdeterministische Turingmaschine entschieden wird.

- $\mathrm{NPSPACE} = \bigcup_{k \in \mathbb{N}} \mathrm{NSPACE}(n^k)$
  Gleich wie NPSPACE, nur das es von einer Nichtdeterministische Turingmaschine entschieden wird.

- $\mathrm{NEXPSPACE} = \bigcup_{k \in \mathbb{N}} \mathrm{NSPACE}(2^{n^k})$
  Gleich wie EXPSPACE, nur das es von einer Nichtdeterministische Turingmaschine entschieden wird.

Deterministische Zeitkomplexitätsklassen:

- $\mathrm{P} = \bigcup_{k \in \mathbb{N}} \mathrm{DTIME}(n^k)$
  P ist die Menge der Sprachen, die von einer deterministische Turingmaschine in polynomieller Zeit entschieden werden kann.

- $\mathrm{EXP} = \bigcup_{k \in \mathbb{N}} \mathrm{DTIME}(2^{n^k})$
  EXP ist die Menge der Sprachen, die von einer deterministische Turingmaschine in exponentieller Zeit entschieden werden kann.

Nichtdeterministische Zeitkomplexitätsklassen:

- $\mathrm{NP} = \bigcup_{k \in \mathbb{N}} \mathrm{NTIME}(n^k)$
  P ist die Menge der Sprachen, die von einer nichtdeterministische Turingmaschine in polynomieller Zeit entschieden werden kann.

- $\mathrm{NEXP} = \bigcup_{k \in \mathbb{N}} \mathrm{NTIME}(2^{n^k})$
  P ist die Menge der Sprachen, die von einer nichtdeterministische Turingmaschine in polynomieller Zeit entschieden werden kann.

6.1.3 Zeit- und Platz-Hierarchiesätze

Die Hierarchiesätze formalisieren eine intuitive Annahme übe Turingmaschinen, nämlich das eine TM welcher mehr Ressourcen (Zeit und Platz) zur Verfügung gestellt werden, auch mehr Sprachen entscheiden kann.
Formal geschrieben lautet der Satz:
Für jede (zeit/platz-konstruierbare) Funktion $f : \mathbb{N} \to \mathbb{N}$ existiert eine Sprache A, die in $\mathcal{O}(f(n))$ Zeit/Platz entscheidbar ist aber nicht in $o(f(n))$ Zeit/Platz.
Der Beweis dieses Satzes wird hier an dieser Stelle nicht geführt!
Aus diesem Satz ergeben sich nun folgende Relationen:

- P $\subsetneq$ EXP
- NP $\subsetneq$ NEXP
- L $\subsetneq$ PSPACE $\subsetneq$ EXPSPACE
- NL $\subsetneq$ NPSPACE $\subsetneq$ NEXPSPACE

Bemerkung: Hier haben wir nun echte Teilmengen Beziehungen was gut ist um genauere Aussagen über die Komplexitätsklassen zu treffen, da Mengen eben nicht mehr gleich sein können und man somit bessere Einschränkungen für Probleme machen kann.
Hier sei auch erwähnt, dass die Hierarchiesätze lediglich Aussagen über gleiche Modelle mit gleichen Ressourcen getroffen wird (also zum Beispiel Zeit auf einer DTM). Es wird keine Aussagen zu Relationen zwischen Maschinenmodellen oder zwischen Ressourcen getroffen. Solche Aussagen können aber auch getroffen werden, zum Beispiel mit dem Satz von Savitch.

6.2 Satz von Savitch

In Kapitel 5.4 haben wir festgestellt, dass die Zeit-Komplexitätsklasse NP vermutlich mächtiger ist als P. Für die Platz-Komplexität gilt dies nicht, hier bringt der Nichtdeterminismus kaum Vorteile. Der Satz von Savitch besagt, dass sich eine NTM auf einer DTM mit geringen (Platz-)Mehraufwand (=> quadratisch höher) simulieren lässt und gilt als eines der wichtigsten Resultate für die Platzkomplexität.
formale Beschreibung:
Für jede Funktion $f : \mathbb{N} \to \mathbb{R}^+$, wobei $f(n) \geq n$, gilt

$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n))$

6.2.1 CANYIELD

Wir wollen diesen Satz nun Beweisen, indem wir eine quadratisch-platzbeschränkte($\mathcal{O}(f(n)^2)$) DTM bauen die eine $f(n)$-platzbeschränkte NTM simuliert. Vorüberlegungen:

- Wenn Platz $\mathcal{O}(f(n))$ (Anzahl der Bandquadrate), dann ist Zeit t $\mathcal{O}(2^{f(n)})$-beschränkt (=Anzahl der möglichen Zustände)
  $\#Konfigurationen^{\kappa} = |Q| * |\Gamma|^{f(n)} => O(2^{f(n)})$

- Einfachster Ansatz: Alle Pfade der NTM Aufzählen kostet $\mathcal{O}(2^{f(n)})$ Platz

- Konfiguration einer TM benötigt $\mathcal{O}(|Q| + f(n)) = \mathcal{O}(f(n))$ Platz

- Alle Konfiguration aufzählen ist Platz-mäßig viel billiger

Beweisidee:
Wir lösen das Yieldability Problem und beweisen somit den Satz von Savitch. Das Yieldability Problem besagt: Kann eine Turingmaschine von einer Konfiguration (hier κ_1) zu einer anderen Konfiguration (hier κ_2) in t Schritten kommen? Die Funktion $CANYIELD(\kappa_1, \kappa_2, t)$ entscheidet eben genau dieses Problem (gibt ja aus, wenn Konfiguration κ_1 der TM zu κ_2 in t Schritten erreichbar, nein wenn nicht in t Schritten erreichbar). Wir müssen also eine Version von CANYIELD finden, welche $\mathcal{O}(f^2(n))$ Platz läuft.

6.2.2 Canyield Algorithmus

In den folgenden Punkten wird der Canyield Algorithmus, welcher zum Beweis vom Satz von Savitch dient textuell bescrieben: Der Ansatz beruht auf den Divide and Conquer Prinzip

1. Wenn $t = 1$ (also ein Schritt) teste ob $\kappa_1 = \kappa_2$, oder ob κ_2 in einem Schritt von κ_1 aus erreicht werden kann. Berichte '*ja*' wenn beides zutrifft, sonst '*nein*'

2. Wenn $t > 1$ dann iteriere über alle möglichen Konfiguration κ_m mit Platzbedarf $f(n)$

3. Rufe $CANYIELD(\kappa_1, \kappa_m, \frac{t}{2})$ auf

4. Rufe $CANYIELD(\kappa_m, \kappa_2, \frac{t}{2})$ auf

5. Wenn beide Aufrufe '*ja*' ergaben, gib '*ja*' aus

6. Wenn noch keine Ausgabe, gib '*nein*' aus

Die Rekursionstiefe des Algorithmus ist logarithmisch bezüglich der Anzahl der Schritte t, daher kommt:
$\log \mathcal{O}(2^{f(n)}) = \mathcal{O}(f(n))$
Da in jeder Rekursion die Konfiguration einer TM ($\mathcal{O}(f(n))$ siehe oben) braucht der Algorithmus einen Platzbedarf von: $\mathcal{O}(f(n)) * \mathcal{O}(f(n)) = \mathcal{O}(f^2(n))$! Dies erfüllt unsere vorher definierten Anforderungen, also eine NTM durch eine DTM mit dem Aufruf $CANYIELD(\kappa_{start}, \kappa_{accept}, 2^{f(n)})$ simuliert werden.

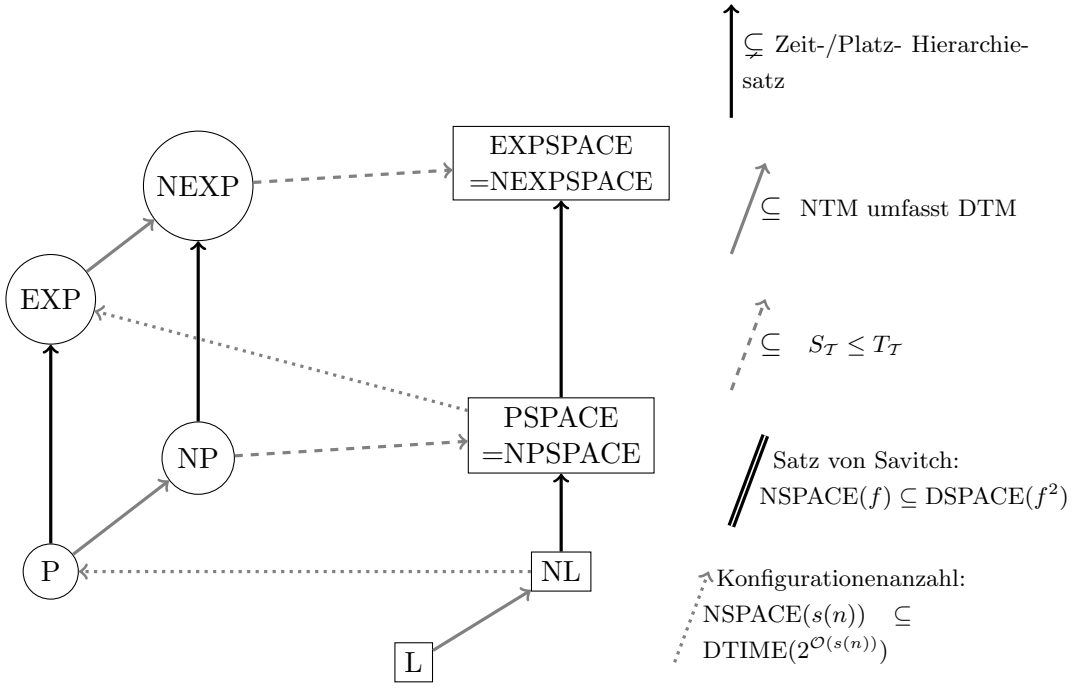

Abbildung 6.1: Veranschaulichung der Beziehungen zwischen den Komplexitätsklassen

6.3 Klassenstruktur

Wir haben nun schon einige Aussagen bezüglich der Komplexitätsklassen getroffen und können damit schon einige Inklusionen von häufig verwendeten Klassen treffen:

$$\underbrace{P \subseteq NP, EXP \subseteq NEXP}_{\text{DTM ist Spezialfall der NTM (NTM allgemeiner als DTM)}}$$

$$\underbrace{P \subsetneq EXP, NP \subsetneq NEXP, L \subsetneq PSPACE \subsetneq EXPSPACE, NL \subsetneq NPSPACE \subsetneq NEXPSPACE}_{\text{Hierarchiesätze(mehr Ressourcen mehr Sprachen entscheiden)}}$$

$$\underbrace{P \subseteq PSPACE, NP \subseteq NPSPACE}_{\text{Platzkomplexität} \leq \text{Zeitkomplexität}}$$

Die nächste Grafik zeigt die Inklusionen (und die Begründung) aller besprochener Komplexitätsklassen:

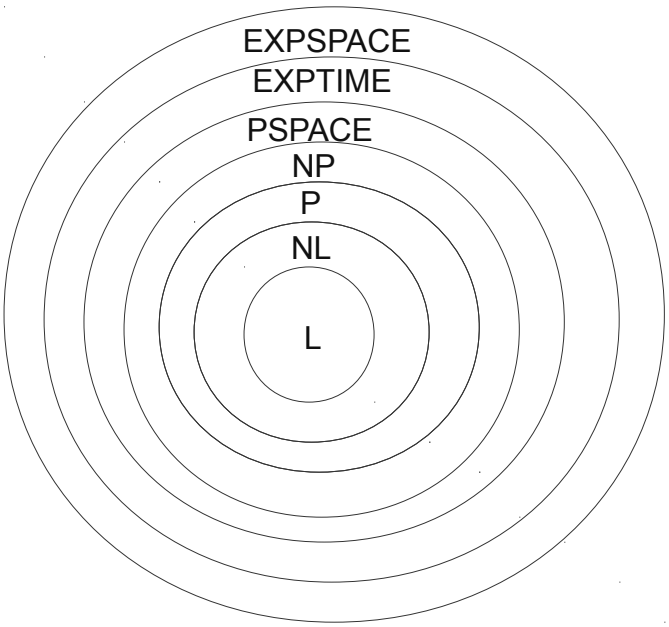

Abbildung 6.2: Inklusionen der Komplexitätsklassen

Bemerkung: Die hier angeführte Graphik wäre ein besonderer Fall der Hierarchierelationen. Da wir bei diesen Relationen kleiner-gleich Beziehungen haben, kann es sein, dass sich manche Klassen überdecken, oder im worst case alle Klassen gleich groß sind (und sich somit überdecken).

6.3.1 Zusammenhang Zeitkomplexität und Platzkomplexität

Bei Komplexitätsbetrachtungen ist es wichtig, sich den Zusammenhang von Zeit und Platz einmal genauer anzuschauen. Bei einem Algorithmus kann die Zeitkomplexität niemals kleiner (max. gleich) der Platzkomplexität sein, da man für das schreiben in ein Bandquadrat einen Rechenschritt braucht. Trotzdem ist Platz eine mächtigere Ressource, da man ihn öfters verwenden kann. Aber auch der Platz beeinflusst die Zeit. Hier nehmen wir das Beispiel einer k-Band TM und einer 1-Band TM her. Bei einer k-Band TM können in jedem Schritt k Bänder beschrieben werden, während bei der 1-Band TM eben in jedem Schritt nur ein Bandquadrat beschrieben werden kann. Wie bereits erwähnt kann eine k-Band TM auf einer 1-Band TM in $O(t(n)^2)$ Zeit simuliert werden.

7 Reduktion

Sebastian Hrauda, Mattias Rauter

Die Idee einer *Reduktion* ist das Lösen eines neuen Problems mit Hilfe einer bereits bekannten Lösung für ein anderes Problem. Hierzu wird eine Probleminstanz des ersten Problems in eine Probleminstanz des zweiten Problems übersetzt. Daher können Reduktionen auch dazu verwendet werden um die Beziehung zwischen Problemen zu verstehen ohne auf ein Maschinenmodell zurückgreifen zu müssen.

Seien zwei Probleme (A und B), deren Eingabeworte x (für A) und y (für B) und ein Algorithmus, der B löst, gegeben.
Es wird eine Reduktion R gesucht, die x in eine gültige Eingabe von B transformiert ($y = R(x)$), sodass der gegebenen Algorithmus auf y angesetzt $A(x)$ löst.

$R(x)$ wird »Reduktion von A nach B« genannt.

Es gibt verschiedene Arten von Reduktionen - in dieser Vorlesung werden nur *Mapping-Reduktion* bzw. *Beschränkte Reduktion* behandelt.

Definition 7.1 (Mapping-Reduktion)**.** *Eine Sprache A ist* (many-one) reduzierbar *auf B*

$$A \leq_m B$$

genau dann wenn eine berechenbare *Funktion $R : \Sigma^* \mapsto \Sigma^*$ existiert, sodass*

$$\forall w \in A \Longleftrightarrow R(w) \in B.$$

Es wird also die Probleminstanz von A auf die Probleminstanz von B *gemapped*.

Um Reduktion sinnvoll auf Komplexitätsklassen anzuwenden, wird Zeit- und Platzbedarf der Reduktionsfunktion R eingeschränkt.

Definition 7.2 (Polynomiell zeitbeschränkte Reduktion)**.** *Eine Sprache A muss mittels einer Reduktionsfunktion R auf eine Sprache B in* polynomieller Zeit *reduziert werden können.*

$$A \leq_{\mathrm{P}} B$$

Es können also Zeitklassen ab P untersucht werden.

Definition 7.3 (Logarithmisch platzbeschränkte Reduktion)**.** *Eine Sprache A wird mittels Reduktionsfunktion R auf eine Sprache B reduziert. Dabei muss R auf* logarithmischen Platz *berechnet werden können.*

$$A \leq_{\mathrm{L}} B$$

Hier sind Zeitklassen ab P und Platzklassen ab L untersuchbar.

Reduktionen (auch polynomiell zeitbeschränkte und logarithmisch platzbeschränkte) sind *transitiv*, d.h. wenn $A \leq_m B$ und $B \leq_m C$ gilt, gilt auch $A \leq_m C$.

7.1 Boolsche Schaltkreise

Unter *Boolschen Schaltkreisen* versteht man die Ansammlung von Logikgattern (UND/AND, ODER/OR, NICHT/NOT) g_i mit Eingängen x_i, die durch Leitungen verbunden sind. Zyklen sind nicht erlaubt, d.h. es darf kein Ausgang mit einem Eingang eines zuvor vorkommenden Bausteins verbunden sein.

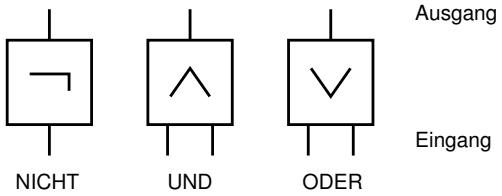

Abbildung 7.1: Darstellung der Logikgatter

Ein Beispiel für einen boolschen Schaltkreis sieht wie folgt aus:

Beispiel 7.1

$$y = (\neg x_1 \wedge x_2) \vee x_3$$

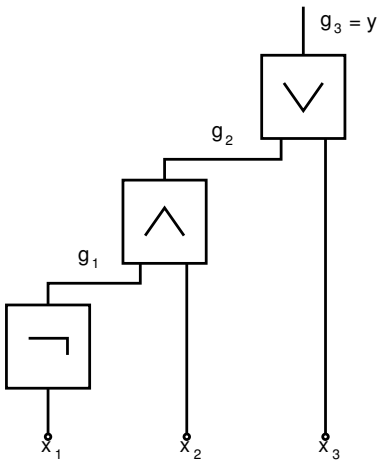

Abbildung 7.2: Beispiel eines boolschen Schaltkreises

7.1.1 Probleme

Im Zusammenhang mit boolschen Schaltkreisen sind viele wichtige Probleme definiert. Ein Problem, das in P liegt, sowie zwei in NP liegende Probleme, werden im folgenden vorgestellt.

Problem in P:

- `CIRCUIT-VALUE`
  Gegeben ist ein boolscher Schaltkreis (siehe Kapitel 4.5.3) eine Eingabe $x_1...x_l$ - zu berechnen ist die Ausgabe y.

Probleme in NP:

- `CIRCUIT-SAT`
  Gesucht ist eine Belegung für $x_1...x_l$ für die y wahr (true) wird.
  `CIRCUIT-SAT` wird auch »Schaltkreis-Erfüllbarkeits-Problem« genannt.
- `3SAT`
  Gesucht ist eine Belegung für $x_1...x_l$ für einen Boolschen Ausdruck in konjunktiver Normalform (KNF) mit *genau 3* Variablen pro Disjunktionsterm.

7.2 Reduktionsbeispiele

7.2.1 `REACH` $\leq_L$ `CIRCUIT-VALUE`

- `REACH (= Grapherreichbarkeit)`
  - **Gegeben:** Graph G=(V,E) und zwei gewählte Knoten u,v
  - **Frage:** Existiert ein Weg von u nach v innerhalb des Graphen G?
  - **Antwort:** JA oder NEIN
- `CIRCUIT-VALUE (= Schaltkreisberechnung)`
  - **Gegeben:** Boolescher Schaltkreis mit $\wedge, \vee, \neg$ Gattern mit Konstanten 0 oder 1 (=FALSCH , WAHR)
    Als Eingabe wird 0 oder 1 verwendet und keine Variablen (=Variablenfreier Schaltkreis)
    Bei `CIRCUIT-SAT` werden Variablen statt konstante Werte als Eingabe verwendet.

Reduktion

- **Idee:**
  Graph in Booleschen Schaltkreis kodieren und in diesem Graph nach einem Weg von u nach v suchen. Gibt es so einen Weg dann liefert der Algorithmus das Ergebnis 1 (= true) sonst 0.
- **Schaltkreiseigenschaften:**
  - Ergebnis des Schaltkreis ist genau dann 1 (= true), wenn ein Weg von 1 nach n existiert.
  - Konstruktion des neuen Graphen muss in logarithmischen Raum passieren.

- **Konstruktion von R(G) (=Boolscher Schaltkreis)**
  Für die Konstruktion des Schaltkreises verwenden wir direkt die Formulierung der Graph- Erreichbarkeit mittels der transitiven Hülle (siehe Kapitel 0.4.2). Diese lässt sich in einen bool'schen Ausdruck umformen. Dazu braucht man drei Indizes (i,j,k) wobei i,j Knoten aus dem Graph sind und k die Weglänge im Graphen bezeichnet. i und j wandern beliebig zwischen 1 und n (Knoten aus dem Graph) und k kann zwischen 0 und n groß sein.

  Der Schaltkreis lässt sich durch verschalten der Elemente g_{kij} und h_{kij} realisieren

$$
\begin{aligned}
1 \xrightarrow[V]{j} i \quad &\Leftrightarrow \quad \exists k \in V : 1 \xrightarrow[V]{j-1} k \xrightarrow[V]{} i \\
&\Leftrightarrow \quad \bigvee_{k=1}^{n} 1 \xrightarrow[V]{j-1} k \wedge \underbrace{k \xrightarrow[V]{} i}_{a_{ki}} \\
&\Leftrightarrow \quad \underbrace{\left(1 \xrightarrow[V]{j-1} 1 \wedge a_{1i}\right)}_{h_{j-1,i1}} \vee \underbrace{\left(1 \xrightarrow[V]{j-1} 2 \wedge a_{2i}\right)}_{h_{j-1,i2}} \vee \cdots \vee \underbrace{\left(1 \xrightarrow[V]{j-1} n \wedge a_{ni}\right)}_{h_{j-1,in}} \\
&\Leftrightarrow \quad \underbrace{\underbrace{\underbrace{h_{j-1,i1}}_{g_{ji1}} \vee h_{j-1,i2}}_{g_{ji2}} \vee \cdots \vee h_{j-1,in}}_{g_{jin}} \\
g_{jin} \quad &\Leftrightarrow \quad g_{ji,n-1} \vee h_{jin} \qquad \text{mit:} \quad h_{jik} \quad \Leftrightarrow \quad g_{j-1,kn} \wedge a_{ki}
\end{aligned}
$$

  1. $g_{i,j,k}$ (mit $1 \leq i, j \leq n, 0 \leq k \leq n$) soll genau dann 1 (=true) sein, wenn in G ein Weg von i nach j existiert, der nur über Knoten läuft $\leq$ k. Der Konten k muss also nicht durchwandert werden.
  2. $h_{i,j,k}$ (mit $1 \leq i, j, k \leq n$) soll genau dann 1 (=true) sein, wenn in G ein Weg von i nach j existiert, und keine Knoten $>$ k hat und durch den Knoten k gehen muss.

  Das Erstellen dieses Graphes wird induktiv über die Variable k durchgeführt. Der Aufbau ist in Abbildung 7.3 veranschaulicht.

  - k = 0
    Auf dieser Ebene gibt es nur $g_{i,j,0}$ Gatter und sie entsprechen den Blättern im Graphen. Anders gesagt sind das die Input-Gatter, welche genau dann 1 (=true) ergeben, wenn die Indizes j und i gleich sind oder wenn es zwischen den Knoten i und j eine Kanten oder einen Weg gibt.
  - k größer 0
    * $h_{i,j,k}$ entsprechen AND-Gatter mit zwei Eingängen

      1.) $g_{i,k,k-1}$ ist wahr wenn es einen Weg von i nach k gibt, der keinen Konten $\geq$ k hat
      2.) $g_{k,j,k-1}$ ist wahr wenn es einen Weg von k nach j gibt, der keinen Konten $\geq$ k hat
      Sind beide Eingänge 1 (=true), evaluiert $h_{i,j,k}$ zu 1 (=true) sonst zu 0.

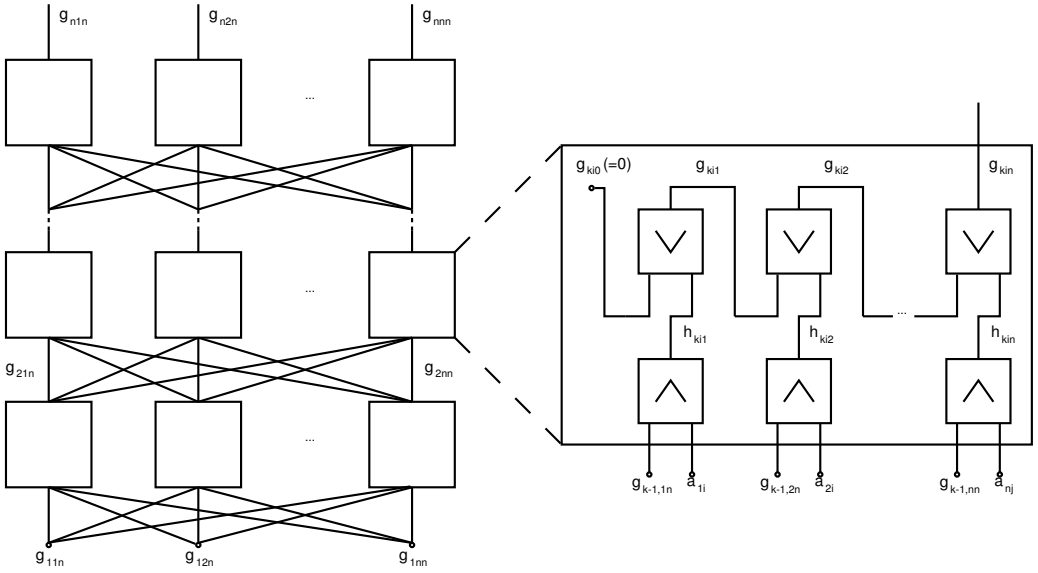

Abbildung 7.3: Beispiel Reach-Circuit Graph

* $g_{i,j,k}$ entsprechen OR-Gatter mit zwei Eingängen

  1.) $g_{i,j,k-1}$ ist wahr wenn es einen Weg von i nach j gibt, der keinen Konten $\geq$ k hat
  2.) $h_{i,j,k}$ ist wahr wenn es einen Weg von i nach j gibt, der den Konten k enthält
  Ist jetzt einer der beiden Eingänge 1 (= true), evaluiert $g_{i,j,k}$ zu 1 (=true) sonst zu 0.

* $g_{1,n,n}$ liefert am Ende genau die Ausgabe des Schaltkreises. Somit erfüllt unser Schaltkreisgraph am Ende `REACH`.

Durch die vollständige Induktion kann nun bewiesen werden, dass in $g_{1,n,n}$ 1 (=true) steht, wenn es einen Weg von 1 bis n gibt. Da die erstellende Turing-Maschine nur drei zusätzliche Zähler ($0 \leq i, j, k \leq n$) speichern muss. Jeder dieser Zahlen kann dabei maximal n groß werden. Somit ergibt sich ein Raumbedarf von $\mathcal{O}$ = (3 log n) = $\mathcal{O}$ (log n) Somit hat man eine LOGSPACE Reduktion von `REACH` auf `CIRCUIT-VALUE`.

7.2.2 `CIRCUIT-VALUE` $\leq_{\mathrm{L}}$ `CIRCUIT-SAT`

Das `CIRCUIT-VALUE` Problem ist recht trivial auf `CIRCUIT-SAT` reduzierbar, da es sich hierbei nur um Spezialfälle von `CIRCUIT-SAT` Problem handelt. Der einzige Unterschied besteht darin, dass `CIRCUIT-SAT` nicht konstante Eingaben von 0 oder 1 entgegen nimmt, sondern diese durche $x_1, ..., x_n$ Variablen ersetzt werden. Die Frage erweitert sich dahingehend ob es eine Belegung der Variablen $x_1, ..., x_n$ gibt, welche den Schaltkreis erfüllen. Wenn nun eine beliebige Belegung ausgesucht wird und sie den Variablen zugeordnet wird, erhält man so ein `CIRCUIT-VALUE` Problem. Dieser Reduktionsbeweis lässt sich nur in eine Richtung `CIRCUIT-VALUE` $\leq_{\mathrm{L}}$ `CIRCUIT-SAT` druchführen. Doch hierbei wird gar kein Aufwand benötigt um die Reduktion durchzuführen.

7.2.3 CIRCUIT-SAT $\leq_L$ 3SAT

Auf den ersten Blick scheint ein beliebiger Schaltkreis viel mächtiger zu sein als ein Schaltkreis in 3SAT Form. Daher würde man erwarten, dass CIRCUIT-SAT viel mehr Sprachen entscheiden kann als 3SAT. Überraschenderweise, kann man aber durch Reduktion zeigen, dass beide gleich mächtig sind. 3SAT $\leq_L$ CIRCUIT-SAT ist trivial zu sehen, da 3SAT ja ein Spezialfall von CIRCUIT-SAT ist. Es gilt jedoch auch die andere Richtung CIRCUIT-SAT $\leq_L$ 3SAT.

Beweis

Sei C ein Schaltkreis. Erstelle $R(C)$ in 3-KNF in L mit der Eigenschaft:
C besitzt eine erfüllende Belegung und auch $R(C)$, wobei beide Belegungen komplett identisch sind. Einfach gesagt, der Schaltkreis wird in KNF Formel umgeformt.Hierzu werden die Ausgänge der einzelnen Gatter des Schaltkreises durch *raten* berechnet.

Konstruktion

$R(C)$ enthält alle Variablen $x_1, ..., x_n$ aus C und für jedes Gatter wird eine eigene Variable g (=Gatter Variablen) eingeführt. Die Variablen g repräsentiert die Ausgabe des Gatters. (true, false,$\wedge, \vee, \neg$) Hierzu muss eine Abhängigkeit der Variablen g und der Variablen $x_1, ..., x_n$ erzeugt werden. Die Abhängigkeit wird in Form von KNF-Klauseln in $R(C)$ eingefügt.

Der 3SAT Schaltkreis überprüft dann lediglich, ob die geratenen Gatter-Ausgaben auch wirklich einer gültigen Berechnung durch den Schaltkreis C entsprechen. Hierzu müssen für jede geratene Belegung folgende Regeln eingefügt werden.

Die Regeln für das Erzeugen der KNF Form vom Schaltkreis C für jedes Gatter g: [1]

Gatter	Regel
g = true	g
g = false	$\neg$g
g = $\neg$h	($\neg$g $\vee$ $\neg$h) $\wedge$ (g $\vee$ h)
g = x	(g $\vee$ $\neg$x) $\wedge$ ($\neg$g $\vee$ x)
g = h $\vee$ f	($\neg$h $\vee$ g) $\wedge$ ($\neg$f $\vee$ g) $\wedge$ (h $\vee$ f $\vee$ $\neg$g)
g = h $\wedge$ f	($\neg$g $\vee$ h) $\wedge$ ($\neg$g $\vee$ f) $\wedge$ ($\neg$h $\vee$ $\neg$f $\vee$ g)

Die Regeln für alle Gatter von C werden und-verknüpft und bilden das Ergebnis der Reduktion. Der gesamte Ausdruck ist in KNF und es werden maximal 3 Literale pro Klausel verwendet. Daher lassen sie sich in einem 3SAT Schaltkreis darstellen. In einem Schaltkreis mit weniger als 3 Literalen pro Klauseln ist dies nicht möglich.

Durch die bereits erwähnte Transitivität der Reduktion gilt:
REACH $\leq_L$ CIRCUIT-VALUE $\leq_L$ CIRCUIT-SAT $\leq_L$ 3SAT

[1] Hasso-Plattner-Institut Komplexitätstheorie (SS 2010) Reduktion und Vollständigkeit 2/3 Prof. Dr. Christoph Meinel Link

8 Vollständigkeit

Michaela Heschl, Dominik Ziegler

Im Allgemeinen betrachtet man ein Problem als vollständig gegenüber einer Komplexitätsklasse, wenn es kein anderes Problem, welches bewiesenermaßen in dieser Klasse ist, gibt, das schwerer zu berechnen ist, und dieses Problem außerdem in der Klasse enthalten ist. Wie schwer eine Klasse zu berechnen ist, wird also ausschließlich durch die in ihr enthaltenen Probleme angegeben. Mit Hilfe von vollständigen Problemen kann man nun also die Äquivalenz von Klassen, die sich in ihrer Definition unterscheiden, beweisen.

Definition 8.1 ($\mathcal{C}$-Vollständigkeit)**.** *Sei $\mathcal{C}$ eine Komplexitätsklasse (z.B.:* P*,* NP*, ...). Eine Sprache A heißt $\mathcal{C}$-vollständig, wenn sie $\mathcal{C}$-schwer ist und in $\mathcal{C}$ liegt.*

$$A \text{ ist } \mathcal{C}\text{-vollständig} \iff A \in \mathcal{C} \land \forall L \in \mathcal{C} : L \leq_{\mathrm{L}} A$$

In jeder Komplexitätsklasse $\mathcal{C}$ kann man nach $\mathcal{C}$-vollständigen Sprachen suchen. Diese Sprachen bilden die Klasse der schwersten Sprachen *innerhalb* der Klasse.

8.1 Schwere

Lassen sich alle Sprachen aus einer Komplexitätsklasse $\mathcal{C}$ auf eine bestimmte Sprache (effizient) reduzieren, dann ist diese Sprache gleich schwer oder schwerer zu entscheiden, wie **jede** andere Sprache aus $\mathcal{C}$. Diese Sprachen werden $\mathcal{C}$-schwer genannt.

Definition 8.2 ($\mathcal{C}$-Schwere)**.** *Sei $\mathcal{C}$ eine Komplexitätsklasse (z.B.:* P*,* NP*, ...). Eine Sprache A heißt $\mathcal{C}$-schwer, wenn alle Sprachen $L \in \mathcal{C}$ auf A reduziert werden können.*

$$A \text{ ist } \mathcal{C}\text{-schwer} \iff \forall L \in \mathcal{C} : L \leq_{\mathrm{L}} A$$

In der deutschsprachigen Literatur wird auch häufig der Begriff *Härte* (direkte Übersetzung vom Englisch *hardness*) als Synonym verwendet.

8.1.1 P-Schwere

Um zu beweisen, dass ein Problem P-Schwer ist, muss laut Definition 8.2 für jede Sprache $L \in \mathcal{P}$ gezeigt werden, dass L das Problem entscheidet und L unter logarithmischem[1] Platzbedarf auf das

[1] Formal gesehen reicht poly. Zeitaufwand um P-Vollständigkeit zu zeigen

Problem reduziert werden kann.

8.1.1.1 CIRCUIT-VALUE ist P-schwer

Satz 8.1. *CIRCUIT-VALUE ist* P*-schwer.*

$$\forall L \in \mathrm{P} : L \leq_{\mathrm{L}} \textit{CIRCUIT-VALUE}$$

Hier muss eine Aussage über alle Algorithmen in P getroffen werden. Um solch eine Aussage zu treffen nehmen wir den Umweg über die Turingmaschine. Wir zeigen, dass für jede beliebige poly. zeitbeschränkte TM $\mathcal{T}$ ein äquivalenter Schaltkreis existiert, der genau dann wahr ausgibt, wenn die $\mathcal{T}$ akzeptiert.

Beweisidee Damit Satz 8.1 gilt, also CIRCUIT-VALUE P schwer ist, muss gezeigt werden, dass für jede Sprache $L \in \mathrm{P}$ ein *Schaltkreis* konstruiert werden kann der L entscheidet. In Folge muss also eine DTM $\mathcal{T}_L$ existieren die L in polynomieller Zeit entscheidet. Der erzeugte Schaltkreis soll schließlich genau dann wahr ausgeben, wenn $\mathcal{T}_L$ akzeptiert.[2]

Beweis Sei $w \in L$ die Eingabe für $\mathcal{T}_L$, mit $|w| = n$ und $\mathcal{T}_m$ n^k-zeitbeschränkt, mit konstantem k. Wir benutzen die *Berechnungstabellen-Methode* um die Konstruktion des Schaltkreises zu zeigen. Die Berechung von von $\mathcal{T}_m$ wird als eine $n^k \times n^k$-Tabelle C mit Feldern c_{ij} dargestellt.[3] Die Zeilen beschreiben jeweils einen Berechnungsschritt und die Spalten entsprechen einem Bandquadrat.

Um den Beweis leichter führen zu können werden einige Einschränkungen getroffen. Wichtig hierbei ist, dass keine dieser Beschränkungen die Berechnungsstärke der Maschine mindert, oder die Kompexität wesentlich vergrößert:

- Wir betrachten lediglich die 1-Band TM, $\mathcal{T}_L = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$
- Die TM hält nach maximal $n^k - 2$ Schritten
- Hält die Maschine nach weniger als n^{k-2} Schritten, werden überzählige Zeilen der Tabelle mit Kopien der letzten Iteration aufgefüllt
- Die TM startet auf dem ersten Symbol der Eingabe
- Der Lesekopf besucht nie die erste Spalten ($\triangleright$)
- Die TM hält stets auf dem ersten Bandquadrat

Um eine Reduktion durchzuführen, muss eine Codierung erstellt werden. Hierfür gilt:

- Die Berechnungstabelle $c_{ij} = \sigma_s$, für das Wort $\sigma \in \Gamma$ im Zustand $s \in Q$ gilt genau dann, wenn der Kopf der TM zum Zeitpunkt i über dem Symbol j steht und sich die Maschine im Zustand s befindet
- Das Alphabet der Berechnungstabelle wird definiert als $c_{ij} \in \Gamma \cup \{\sigma_s | \sigma \in \Gamma, s \in Q\}$

[2] Papadimitriou, *Computational Complexity*, S. 165–172.
[3] Indizes werden hier von 0 weg gezählt

- Eine Berechnungstabelle heißt akzeptiert, falls gilt:
  $c_{n^k-1,1} = \sigma_s, \quad \sigma \in \Gamma, \quad s \in F$
- Es gilt: eine TM akzeptiert genau dann, wenn die Berechnungstabelle akzeptiert.

Beispiel:

$\triangleright$	0_s	1	1	0	$\square$	$\square$	$\square$	$\square$	$\square$	...
$\triangleright$	$\square$	1_{q_0}	1	0	$\square$	$\square$	$\square$	$\square$	$\square$	...
$\triangleright$	$\square$	1	1_{q_0}	0	$\square$	$\square$	$\square$	$\square$	$\square$	...
$\triangleright$	$\square$	1	1	0_{q_0}	$\square$	$\square$	$\square$	$\square$	$\square$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Vorüberlegungen:

Per Definition kann sich maximal das linke, rechte oder aktuelle Bandquadrat ändern. Das bedeutet also, dass c_{ij} nur von $c_{i-1,j-1}$, $c_{i-1,j}$ oder $c_{i-1,j+1}$ beeinflusst werden kann (siehe Abb: 8.1). Das Feld c_{ij} ändert sicht dabei nur, wenn eines der Vorgänger den Zustand der Maschine kodiert. Außerdem gilt: Die Felder der Berechnungstabelle für $i = 0$, $j = 0$ und $j = n^{k-1}$ sind *a priori* bekannt.

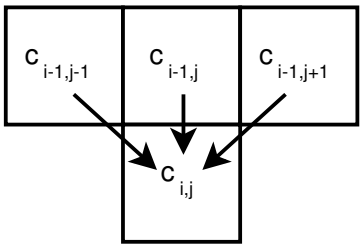

Abbildung 8.1: Abhängigkeit von c_{ij}

Da das Alphabet der Tabelle eine endliche Menge ist, lassen sich alle Felder binär (mit konstanter Bitzahl) kodieren. Daraus lässt sich eine binäre Tabelle, mit Einträgen s_{ijl} erstellen, wobei gilt: $s_{ij1} \ldots s_{ijm}$ kodiert c_{ij}.
Analog zu c_{ij} hängt jeder Eintrag s_{ijl} von $3{*}m$ Vorgänger-Eingängen ab. Diese beschreiben die Überführungsfunktion der TM. Mittels m boolschen Funktionen $F_1, ..., F_m$ lassen sich nun die Abhängigkeiten für jedes s_{ijl} bechreiben.

Darstellung:
Mittels oben genannten Vorüberlegungen, lassen sich nun folgende Sachverhalte darbieten:

- $s_{ijl} = F_l(s_{i-1,j-1,1}, ..., s_{i-1,j-1,m}, s_{i-1,j,1}, ..., s_{i-1,j+1,m})$
- $M_{ij} = (F_1, F_2, ..., F_m)$

Die Funktionen $F_1, ..., F_m$ können widerum als boolscher Schaltkreis dargestellt werden. Wie genau diese Schaltkreise realisiert werden ist für die Analyse unerheblich. Wichtig ist nur zu sehen, dass ihre Größe endlich ist und lediglich von der Größe des Alphabets der Berechnungstabelle abhängt. Wir stellen den gesamten Schaltkreis der die Überführungsfunktion der TM realisiert wie in Abbildung 8.2 dar. Die Berechnungstabelle kann durch aneinandersetzten solcher Schaltkreise wie in Abb. 8.3 dargestellt realisiert werden. Jede Schicht entspricht einer Zeile der Berechnungstabelle.

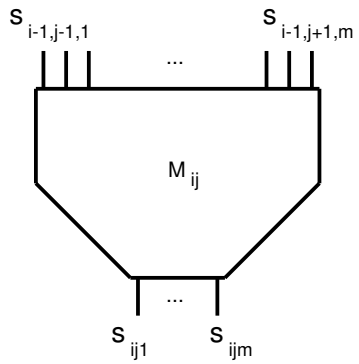

Abbildung 8.2: Boolscher Schaltkreis

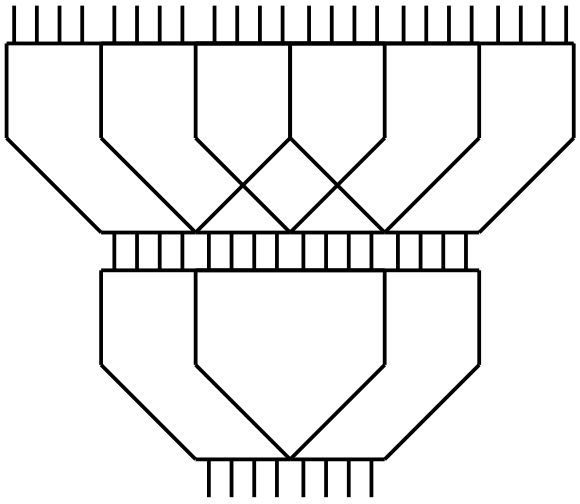

Abbildung 8.3: Homogener Aufbau des gesamten Schaltkreises

Reduktion in L:
Durch oben durchgeführte Reduktion erhält man für die Eingabe w einen Schaltkreis der die TM $\mathcal{T}_L$ simuliert. Dies kann offensichtlich mit logarithmischen Platzbedarf realisiert werden, da der erzeugte Schaltkreis einfache und wiederholende Strukturen besitzt.[4]
Um die erste Zeile der Berechnungstabelle zu erzeugen muss das Eingabewort als Schaltkreis-Gatter kodiert auf das Band geschrieben werden. Der Rest der Tabelle wird mit $\square$ aufgefüllt. Hierzu ist ein Zähler notwendig ($O(log(n))$ Platzbedarf). Da die Größe der Schaltkreise für die Funktionen F_1 ... F_m unabhängig vom Input ist, kann deren Konstruktion durch ein endliches Turingmaschinenprogramm erfolgen und benötigt daher keinen zusätzlichen Platzbedarf. Es sind jeweils zwei Zähler nötig um die quadratische Tabelle aufzubauen (jeweils $O(log(n))$ Platz).

8.2 P-Vollständigkeit

Definition 8.3 ($\mathcal{P}$-Vollständigkeit)**.** *Eine Sprache A heißt $\mathcal{P}$-vollständig, wenn sie $\mathcal{P}$-schwer (Reduktion aller Sprachen $\in \mathcal{P}$ unter logaritmischen Platzbedarf) ist und in $\mathcal{P}$ liegt.*

$$A \text{ ist } \mathcal{P}\text{-vollständig} \iff A \in \mathcal{P} \wedge \forall L \in \mathcal{P} : L \leq_{\mathrm{L}} A$$

8.2.1 `CIRCUIT-VALUE` ist P-vollständig

Bereits in Kapitel 8.1.1.1 wurde gezeigt, dass `CIRCUIT-VALUE` P-schwer ist und in P liegt. Dadurch gilt:

Satz 8.2. *`CIRCUIT-VALUE` ist* P*-vollständig.*

[4] Sipser, *Introduction to the Theory of Computation (Second Edition).*

8.3 NP-Vollständigkeit

Definition 8.4 ($\mathcal{NP}$-Vollständigkeit). *Eine Sprache A heißt $\mathcal{NP}$-vollständig, wenn sie $\mathcal{NP}$-schwer (Reduktion aller Sprachen $\in \mathcal{NP}$ in polynomieller Zeit auf A) ist und in $\mathcal{NP}$ liegt.*

$$A \text{ ist } \mathcal{NP}\text{-vollständig} \iff A \in \mathcal{NP} \wedge \forall L \in \mathcal{P} : L \leq_p A$$

8.3.1 `CIRCUIT-SAT` ist NP-vollständig

Satz 8.3 (Satz von Cook). *`CIRCUIT-SAT` ist* NP*-vollständig.*

8.3.1.1 Beweis

Es ist bekannt, dass `CIRCUIT-SAT` in NP liegt. Folglich muss noch gezeigt werden, dass `CIRCUIT-SAT` auch NP-schwer ist. Dafür muss also ein Schaltkreis konstruiert werden der genau dann *erfüllbar* ist, wenn für jede Sprache $L \in$ NP eine akzeptierende NTM $\mathcal{T}_L$ existiert.
Der Schaltkreis kann durch eine Erweiterung des Schaltkreises für `CIRCUIT-VALUE` realisiert werden. Hierzu treffen wir wieder eine Einschränkung: Es soll maximal 2 *nicht-deterministische* Pfade pro Zeitschritt geben. Diese Einschränkung wird hier zur Vereinfachung der Konstruktion getroffen. Die Verallgemeinerung der Konstruktion für eine beliebige Anzahl an nicht-deterministischen Zustandsübergängen ist aber problemlos möglich. In jedem Zeitschritt, d.h. in jeder Zeile der Berechnungstabelle ist ein nicht-deterministischer Zustandsübergang möglich. Welcher Pfad gewählt wird, wird durch zusätzliche Inputs realisiert. Zusätzlich werden n^k variable Inputs (`?`) $a_1 \dots a_{n^k}$ zum Schaltkreis hinzugefügt.M_{ij} erhälten somit zudem den Input von a_i. Das Entscheidungs-Bit a_i bestimmt nicht-deterministische die Entscheidung im Zeitschritt i.
Dies kann offensitlich in polynomieller Zeit realisiert werden: Es wurde bereits gezeigt, dass ein Schaltkreis in polynomieller Zeit, respektive n, erzeugt werden kann. Der Schaltkreis inklusive unseren variablen Inputs n^k lässt sich somit in $O(n^{2k})$ erstellen. Da der Aufbau weiterhin einfache und repetitive Strukturen verwendet lässt sich eine obere Schranke von $O(n^{2k})$ festlegen.[5]

8.3.2 `3SAT` ist NP-vollständig

Satz 8.4. *`3-SAT` ist* NP*-vollständig.*

Folgt sofort, da bereits bekannt ist, dass `3SAT` $\in$ NP *und* `CIRCUIT-VALUE` $\leq_{\mathrm{L}}$ `3SAT`.

8.4 `CLIQUE`

Beim Cliquenproblem handelt es sich um ein Entscheidungsproblem der Graphentheorie. Als eine Clique wird eine Teilmenge der Knoten in einem ungerichteten Graphen bezeichnet, deren Knoten allesamt mittels einer Kante verbunden sind (siehe Kapitel 0.4.4). Formal bedeutet das:

[5] Ebd.

Definition 8.5 (CLIQUE). *Gegeben: ungerichteter Graph $G = (V, E)$ mit m Knoten, $k \in \mathbb{N}$.*

$$\textit{CLIQUE} = \{\langle G, k\rangle \mid G \textit{ hat Clique der Größe } k\}.$$

8.4.1 CLIQUE ist NP-vollständig

Satz 8.5. *CLIQUE ist* NP*-Vollständig.*

$$\forall L \in \text{NP} : L \leq_{\text{P}} \textit{CLIQUE}$$

Um zu zeigen, dass CLIQUE NP-Vollständig ist, muss gezeigt werden, dass CLIQUE in NP liegt und CLIQUE NP-schwer ist. Der Beweis erfolgt über folgende Reduktion:

$$\texttt{3SAT} \leq_{\text{P}} \texttt{CLIQUE}$$

8.4.1.1 CLIQUE in NP

Um zu beweisen, dass CLIQUE $\in$ NP gilt, wird folgender Algorithmus entwickelt:

- Erzeuge eine Liste von Zahlen $p_1, ..., p_k$
- p_i werden Nicht-deterministisch zwischen 1 und k erzeugt
- Überprüfe ob Wiederholungen in der Liste vorkommen, wenn ja, verwerfe
- Überprüfe, ob Liste eine Clique ist, d.h. ob Kante (p_i, p_j) für jedes Knotenpaar i, j existiert, wenn nein verwerfe
- ansonsten akzeptiere

8.4.1.2 CLIQUE ist NP-schwer

Beweisidee:
Wie bereits erwähnt erfolgt der Beweis über die Reduktion von 3SAT auf CLIQUE. Hierfür muss eine Konstruktion eines ungerichteten Graphen $G = (V, E)$ erfolgen, der folgende Eigenschaft besitzt: Es existiert eine Clique der Große k in G, genau dann wenn 3-KNF Formel ϕ erfüllbar ist. Jedes Literal der Formel, a,b und c, ist in der Form x_i oder $\neg x_i$, wobei $x_i, ..., x_l$ die Variablen von ϕ repräsentieren.

Vorüberlegungen:
Die vorige Formel wird auf k Klauseln erweitert:

- $\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge ... \wedge (a_k \vee b_k \vee c_k)$

Es gilt: ϕ ist genau dann erfüllt, wenn jede der k Klauseln erfüllt ist. Eine Klausel wiederum ist dann erfüllt, wenn mindestens ein Literal erfüllt ist. Wenn ϕ allerdings nicht erfüllt ist, muss sie einen Widerspruch enthalten, also z.B.: x_1 und $\neg x_1$.
Für die Darstellung von ϕ in einem Graphen ist wichtig, dass jedes Literal, also a, b und c durch einen eigenen Knoten repräsentiert werden. Sobald eines der Literal erfüllt ist, soll dieses Teil der Clique sein.

Konstruktion des Graphen:
Die Konstruktion von G wird wie folgt durchgeführt:

- Der Graph G hat $3 * k$ Knoten, entsprechend den Literalen von ϕ
- Die Knoten von G sind in k Gruppen organisiert
- Jede Gruppe besteht aus 3 Knoten und entspricht einer Klausel
- Jedem Knoten wird ein Literal aus der entsprechen Klausel zugeordnet
- Die Kanten von G verbinden alle Knoten außer:
- Knoten innerhalb einer Gruppe (Klausel)
- Knoten mit widersprechenden Literalen (z.B.: x_1 und $\neg x_1$)

Nun muss noch bewiesen werden, dass sämtliche Überlegungen und die generelle Vorgehensweise auch korrekt sind. Für den Beweis der Korrektheit werden folgende zwei Annahmen getroffen:

ϕ ist erfüllbar:
Damit diese Annahme bzw. die Formel ϕ erfüllt ist, muss zumindest ein Literal in jeder der k Klauseln wahr sein. Falls mehr als nur ein Literal wahr ist, wird zufällig eines davon ausgewählt. Daraus ergibt sich, dass in jeder der 3-er Gruppen ein Knoten gewählt wird. Des Weiteren gilt, dass zwischen jedem Knotenpaar eine Kante existiert, da alle Knoten von unterschiedlichen Gruppen stammen und eine erfüllende Variablenbelegung keinerlei Widersprüche enthält. Daraus ergibt sich wiederum, dass der Graph G eine k-Clique enthält.

G enthält eine k-Clique:
Da keine Kanten innerhalb einer Gruppe existieren, wird ausgeschlossen, dass zwei Knoten aus derselben Gruppe gewählt werden. Wahrheitswerte werden den Variablen von ϕ zugewiesen indem die den gewählten Knoten entsprechenden Literale auf wahr gesetzt werden. Das ist immer möglich da widersprüchliche Zuweisungen nicht mit einer Kante verbunden sind und daher nicht in der Clique sind. Diese Zuweisung ist erfüllend, da sich in jeder Klausel ein erfüllendes Literal befindet. Daraus ergibt sich, dass ϕ erfüllbar ist.

Da beide Annahmen hiermit erfüllt sind, ist auch die Korrektheit bewiesen und es folgt:

`CLIQUE` ist NP-vollständig

Beispiel 8.1

Folgende bool'sche Formel sei gegeben:

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$$

Wie man sieht besitzt diese KNF Formel nur zwei Variablen, nämlich x_1 und x_2. Eine erfüllende Belegung dieser beiden Variablen wäre: $x_1 = 0$ und $x_2 = 1$. Abbildung 8.4 zeigt den entsprechenden Graphen mit einer 3-Clique. Die Verbindung zwischen x_1 und $\neg x_1$ wird nicht eingeführt, da diese sich selbst widersprechen würde.

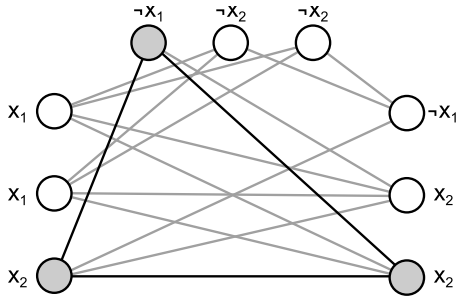

Abbildung 8.4: Beispiel: Reduktion einer 3-KNF Formel mit 3 Klauseln auf einen Graph mit einer 3-Clique

8.5 HAMILTON-PATH

Ein Hamilton-Pfad durch einen Graphen $G = (V, E)$, ist ein Pfad zwischen einem Startknoten s und einen Endknoten t der alle Knoten von G genau einmal besucht. Das dazugehörige Entscheidungsproblem ist wie folgt definiert:

Definition 8.6 (HAMILTON-PATH). *Gegeben: gerichteter Graph $G = (V, E)$, s, $t \in V$.*

$$\textit{HAMILTON-PATH} = \{\langle G, s, t\rangle \mid \textit{es existiert ein Hamilton-Pfad von } s \textit{ nach } t \textit{ in } G\}$$

8.5.1 HAMILTON-PATH ist NP-vollständig

Satz 8.6. *HAMILTON-PATH ist* NP*-vollständig.*

$$\forall L \in \text{NP} : L \leq_{\text{P}} \textit{HAMILTON-PATH}$$

Auch hier gilt, um zu zeigen, dass HAMILTON-PATH NP-vollständig ist, muss gezeigt werden, dass HAMILTON-PATH inNP liegt und NP-schwer ist. Der Beweis erfolgt über folgende Reduktion:

$$\texttt{3SAT} \leq_{\text{P}} \texttt{HAMILTON-PATH}$$

8.5.1.1 Hamilton-Path in NP

Um zu beweisen, dass HAMILTON-PATH $\in$ NP gilt, wird folgender Algorithmus entwickelt:

- Erzeuge eine Liste von Zahlen $p_1, ..., p_m$
- p_i werden Nicht-deterministisch zwischen 1 und m erzeugt
- Überprüfe ob Wiederholungen in der Liste vorkommen, wenn ja, verwerfe
- Überprüfe, ob die Liste ein gültiger Pfad ist, d.h. ob die Kante (p_i, p_{i+1}) für jedes Knotenpaar i, j existiert, wenn nein verwerfe
- ansonsten akzeptiere

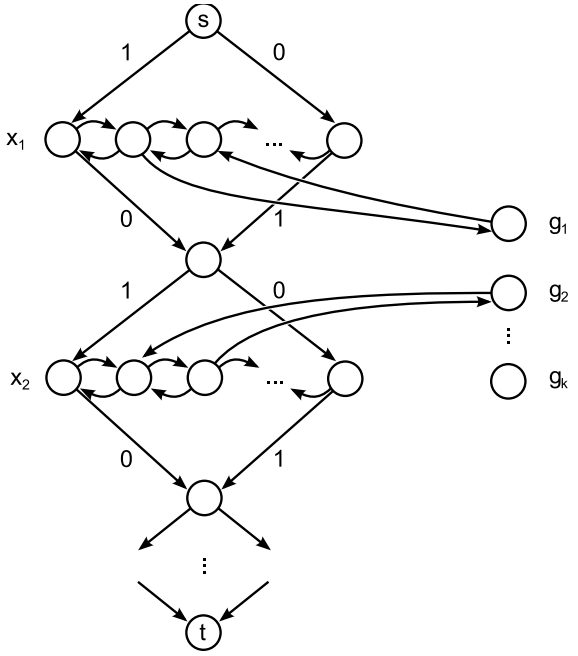

Abbildung 8.5: Veranschaulichung der Reduktion von 3SAT auf HAMILTON

8.5.1.2 HAMILTON-PATH ist NP-schwer

Beweisidee:
Der Beweis erfolgt wider durch Reduktion von 3SAT. Um 3SAT auf das HAMILTON-PATH Problem zu reduzieren, muss ein gerichteter Graph $G = (V, E)$ konstruiert werden. Dieser Graph muss folgende Eigenschaft erfüllen: Es existiert ein Hamilton-Pfad genau dann, wenn eine 3-KNF Formel ϕ erfüllbar ist. Für diese Formel ϕ gelten die gleichen Eigenschaften wie bereits bei der Reduktion von 3SAT auf CLIQUE, d.h.:

- $\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge ... \wedge (a_k \vee b_k \vee c_k)$
- jedes Literal a,b und c ist in der Form x_i oder $\neg x_i$
- $x_i, ..., x_l$ sind die Variablen von ϕ

Vorüberlegungen:
ϕ ist genau dann erfüllt, wenn jede der k Klauseln erfüllt ist. Eine Klausel wiederum ist dann erfüllt, wenn mindestens ein Literal erfüllt ist. Jede Variable wird durch einen Diamanten-Förmigen Teilgraph und jede Klausel durch einen einzelnen Knoten in diesem Graph dargestellt.

Konstruktion des Graphen:
Die Konstruktion von G ist in Abbildung 8.5 veranschaulicht.

- Diamant-Struktur: 2 Eingänge, 2 Ausgänge, für jede Variable von ϕ
- Diamant-Struktur kann nur in einer Richtung durchwandert werden
- Nicht-deterministische Entscheidung entspricht Variablenbelegung (links = 1, rechts = 0)
- Für jede Klausel i wird ein weiterer Knoten g_i erzeugt

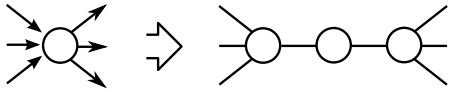

Abbildung 8.6: Veranschaulichung der Reduktion von HAMILTON-PATH auf UHAMILTON-PATH

- Für jede Verwendung der Variable in einer Klausel wird eine Abzweigung über den Klausel-Knoten eingefügt
- Rechts-nach-links, falls Literal negiert, sonst in links-nach-rechts Richtung
- Wenn Klausel-Knoten im Graphen besucht wurde, dann ist die Klausel erfüllt
- Wenn alle Knoten besucht wurden dann ist ϕ erfüllt

Wie auch schon bei der Reduktion von 3SAT auf CLIQUE muss die Korrektheit bewiesen werden. Hierfür werden folgende zwei Annahmen getroffen:

ϕ ist erfüllbar:
Die Klausel-Knoten werden zunächst ignoriert. Jedes Diamanten Modul wird von rechts nach links bzw. umgekehrt durchlaufen, wo eine wahr/falsch Zuweisung erfolgt. In einer erfüllenden Belegung von ϕ kann jeder Klausel-Knoten erreicht werden. Aus der Annahme, dass ϕ erfüllbar ist, folgt somit, dass der Graph G einen Hamilton-Pfad enthält.

G enthält einen HAMILTON-PATH:
Bei dieser Annahme wird Jeder Links-Rechts-Entscheidung in den Diamanten die Variablenbelegung (wahr,falsch) zugeordnet. Die daraus entstehende Belegung erfüllt wiederum ϕ, da jede Variable belegt sein muss (sonst würde ein Knoten übersprungen werden). Des Weiteren enthält ϕ dadurch keinerlei Widersprüche, da sonst mindestens ein Klausel-Knoten nicht besucht werden könnte. Durch die Annahme, dass G einen Hamilton-Pfad enthält, folgt also, dass ϕ erfüllbar ist.

8.5.2 UHAMILTON-PATH

UHAMILTON-PATH beschreibt einen Hamilton-Pfad in einem **un**gerichteten Graphen.

Definition 8.7 (UHAMILTON-PATH). *Gegeben: **un**gerichteter Graph $G = (V, E)$, s, $t \in V$.*

$$\textit{HAMILTON-PATH} = \{\langle G, s, t\rangle \mid \textit{es existiert ein Hamilton-Pfad von } s \textit{ nach } t \textit{ in } G\}$$

Um einen Hamilton-Pfad in einem ungerichteten Graphen erzielen zu können, muss jeder Knoten durch 3 Knoten ersetzt werden.[6] Abbildung 8.6 veranschaulicht die Reduktion von HAMILTON-PATH auf UHAMILTON-PATH. Um aus einem gerichteten Graphen einen ungerichteten zu erhalten, muss jeder Knoten des gerichteten Graphen durch eine solche Konstruktion mit Hilfe von 3 Knoten ersetzt werden. Für diese Konstruktion gilt: Wird einer der Eingangsknoten besucht muss der Graph in dieser Richtung durchlaufen werden, ansonsten kann der Pfad keinen Hamilton-Pfad mehr bilden. Es ist nicht möglich, dass der Pfad umkehrt und später die andere Seite besucht. Entweder kann der zentrale Knoten dann nicht mehr besucht werden oder der Pfad läuft in eine Sackgasse. Da bei der Reduktion lediglich jeder Knoten durch diese Konstruktion ersetzt werden muss ist sie effizient durchführbar.

[6] Schöning, *Theoretische Informatik - kurz gefasst.*

8.5.3 Pfad vs. Kreis

Analog zu HAMILTON-PATH bzw. UHAMILTON-PATH gibt es HAMILTON-CIRCLE bzw. UHAMILTON-PATH oder oft einfach HAMILTON bzw. UHAMILTON. Für diese ist allerdings eine zusätzliche Forderung nötig: $(p_m, ..., p_1)\ \exists\ E$ (damit ist die Angabe von s und t überflüssig). Aus analogen Überlegungen folgt, dass sowohl HAMILTON als auch UHAMILTON NP-Vollständig sind.

9 Randomisierte Algorithmen

Lukas Prokop, Jakob Hohenberger

Wir haben in den vorigen Kapiteln Probleme hinsichtlich ihrer Komplexität betrachtet, wobei Komplexität eine Form der Beurteilung des Ressourcenverbrauchs darstellt. Genauso können wir für gewisse Probleme beweisen, dass es eine untere Schranke hinsichtlich des Ressourcenverbrauchs gibt. Der Ansatz der randomisierten Algorithmen sieht vor, dass wir Zufall als Teil des Prozessmodells einführen und damit zu effizienteren Algorithmen gelangen, die bezüglich einer Eigenschaft (zB Exaktheit) Nachteile aufweisen. Es handelt sich somit um einen approximativen Ansatz.

Anwendungen finden sich etwa in den Gebieten

- Kryptographie
- Maschinelles Lernen
- Neuronale Netze
- Statistische Physik

9.1 Pi Approximation

Als erstes intuitives Beispiel wollen wir uns anschauen, wie wir Zufall nutzen können, um die Kreiszahl π zu approximieren. Dazu betrachten wir folgende Situation:

> Gegeben sei ein Quadrat, welches einen Kreis mit maximaler Ausdehnung enthält. Es fallen Regentropfen auf dieses Quadrat und treffen an einem zufälligen Punkt d auf.

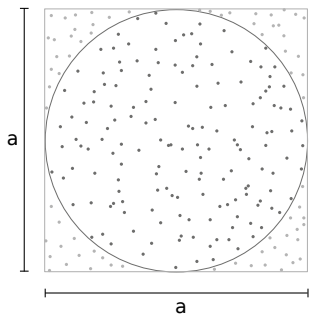

Abbildung 9.1: Pi-Approximation durch Zählen der Regentropfen

Die Fläche des Quadrats sei mit $S = a^2$ und die des Kreises mit $C = \left(\frac{a}{2}\right)^2 \pi$ beschrieben.

$$\frac{C}{S} = \frac{a^2}{4} \cdot \frac{\pi}{a^2}$$

$$\pi = 4 \cdot \frac{C}{S}$$

Da die Wassertropfen gleichverteilt (d.h. an jedem Punkt mit gleicher Wahrscheintlichkeit auftreffen) sind, ist die Anzahl der Tropfen proportional zur Fläche. Wir können nun die Flächeninhalte durch die Anzahl der enthaltenen Tropfen ersetzen. Damit können wir Pi approximieren.

Wir können diese Approximation wahrscheinlichkeitstheoretisch mit einer Zufallsvariable beschreiben:

$$X = \begin{cases} 1 & d \in C \\ 0 & \text{sonst} \end{cases}$$

$$p\left(X = 1\right) = \frac{a^2 \cdot \pi}{4 \cdot a^2} = \frac{\pi}{4}$$

Der Erwartungswert lässt sich als Summe der Wert-Wahrscheinlichkeits-Produkte ermitteln:

$$E[X] = 1 \cdot p\left(X = 1\right) + 0 \cdot \underbrace{p\left(X = 0\right)}_{1-p(X=1)} = \frac{\pi}{4}$$

9.2 Probabilitische Turingmaschine

Bei einer Probabilistischen Turingmaschine (PTM) wird eine NTM herangezogen, welche den Zufall statt eines Orakels zur Entscheidung des nächsten Zustandsübergangs verwendet. Der Algorithmus muss also einen Entscheidungspfad mit mehr als einem Zweig vorweisen, wovon dann einer der Zweige per Zufall gewählt wird. Handelt es sich um einen linearen Entscheidungspfad, verhält sich die PTM äquivalent zu einer TM. Die Verarbeitung eines Entscheidungspfades unter Verwendung des Zufalls bezeichnet man als *Random Walk*.

Es gibt auch andere Berechenbarkeitsmodelle, in denen am Band Zufallsbits liegen oder in ausgewählten Zuständen das gelesene Zeichen zufällig ist.[1] Auf diese Modelle wird hier nicht eingegangen. Es sei darauf hingewiesen, dass im Gegensatz zu einer NTM eine PTM wirklich implementierbar ist.

9.3 Klassifikation randomisierter Algorithmen

Man unterscheidet 2 Klassen von Algorithmen:

Las Vegas Algorithmen Darunter versteht man genau jene Algorithmen, die auf einer Probabilistischen TM ausgeführt werden und deren Ergebnis immer korrekt ist. Lediglich ihre Ausführzeit ist zufällig. Ein bekanntes Beispiel hierfür ist der Quicksort-Algorithmus mit zufälligem Pivot-Element.

Monte Carlo Algorithmen Menge jener Algorithmen, die auf einer Probabilistischen TM ausgeführt werden und und deren Ergebnis falsch sein darf. Wir betrachten dabei einen einseitigen Fehler, der false negatives erlaubt, dh. alle „Ja“ Antworten bei einem Entscheidungsproblem sind korrekt aber „Nein“ Antworten sind richtig *oder* falsch.

[1] Gill, »Computational complexity of probabilistic Turing machines«.

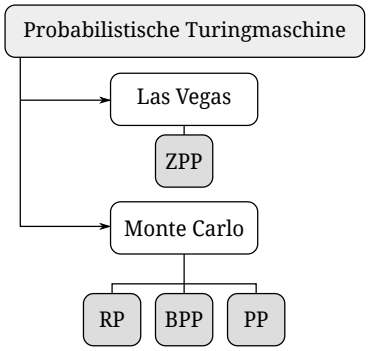

Abbildung 9.2: Komplexitätsklassen randomisierter Algorithmen

Wir unterscheiden hier zwischen 2 Ergebnissen. Wir vergleichen das „berechnete“ Ergebnis mit dem „theoretischen“ Ergebnis. Wir sprechen von einem „korrekten“ Ergebnis, wenn das berechnete Ergebnis mit dem theoretischen Ergebnis übereinstimmt oder keine Aussage getätigt wird („Unbekannt“ als Ergebnis).

9.3.1 Komplexitätsklasse ZPP

ZPP steht für „zero-error probabilistic polynomial time“ Algorithmen (fehlerfreie, polynomiell beschränkte Algorithmen auf einer PTM). Da es sich um einen Las Vegas Algorithmus handelt, muss das berechnete Ergebnis des Algorithmus stets korrekt sein. Er erfüllt folgende Kriterien:

$$f_{\mathcal{T}}(x) \in \{\text{Ja}, \text{Nein}, \text{Unbekannt}\}$$

1. Zeitlich polynomiell beschränkt.
2. Ist Ja theoretisch richtig, wird mit Wahrscheinlichkeit $\geq \frac{1}{2}$ Ja geantwortet (sonst Unbekannt)
3. Ist Nein theoretisch richtig, wird mit Wahrscheinlichkeit $\geq \frac{1}{2}$ Nein geantwortet (sonst Unbekannt)

9.3.2 Komplexitätsklasse RP

Die Klasse RP („randomized polynomial time“)—oder mehrdeutig R—weist die folgenden Einschränkungen auf:

1. Zeitlich polynomiell beschränkt.
2. Ist Ja theoretisch richtig, wird mit Wahrscheinlichkeit $\geq \frac{1}{2}$ Ja geantwortet (sonst Nein)
3. Ist Nein theoretisch richtig, wird Nein geantwortet.

$$L \subseteq RL \subseteq NL \subseteq P \subseteq RP \subseteq NP^2 \subseteq PSPACE = RPSPACE$$

[2]Unter der Annahme $P \neq NP$ gilt $RP \subset NP$. Bei $P \stackrel{?}{=} RP$ handelt es sich um ein offenes Problem der Theoretischen Informatik.

Formal lässt es sich auch wie folgt notieren:

$$p\left(f_{\mathcal{T}}(x) = 1 \mid x \in L\right) \geq \frac{1}{2}$$

$$p\left(f_{\mathcal{T}}(x) = 1 \mid x \notin L\right) = 0$$

Hier entspricht $x \in L$ einer theoretischen Antwort Ja und $f_{\mathcal{T}}(x) = 1$ einer berechneten Antwort Ja. Die erste Formel entspricht der zweiten Definition der Liste. Die zweite Formel entspricht der Aussage „Ist Nein theoretisch richtig, wird mit Wahrscheinlichkeit 0 mit Ja geantwortet“.

Interessant wird es, wenn wir einen Algorithmus in RP mehrfach durchlaufen lassen. Sei k die Anzahl der Durchläufe:

$$\begin{aligned}
k = 1 \qquad & p\left(\text{Nein} \mid x \in L\right) = 1 - p\left(\text{Ja} \mid x \in L\right) \\
k = 2 \qquad & p\left(\text{Nein}^2 \mid x \in L\right) = \left(1 - p\left(\text{Ja} \mid x \in L\right)\right)^2 \\
& \ldots \\
k = n \qquad & p\left(\text{Nein}^n \mid x \in L\right) = \left(1 - p\left(\text{Ja} \mid x \in L\right)\right)^n
\end{aligned}$$

Die Interpretation davon ist, dass die Fehlerrate eines Algorithmus in RP bei mehrfacher Ausführung *exponentiell* sinkt. Damit können wir bei geeigneter Wahl von n die Fehlerwahrscheinlichkeit unter einem Grenzwert ε halten.

In *co-RP* wird die theoretische Antwort für das Problem gewechselt. Daher Probleme, deren theoretische Lösung Nein ist, liefern eine fehlerbehaftete Lösung.

9.3.3 Komplexitätsklasse BPP

Die Klasse („bounded-error probabilistic polynomial time“) garantiert niemals, dass eine theoretische Antwort zur selben berechneten Antwort führt.

1. Zeitlich polynomiell beschränkt.
2. Die Wahrscheinlichkeit die korrekte Antwort zurückzugeben, ist größer $\frac{2}{3}$.

$$p\left(f_{\mathcal{T}}(x) = 1 | x \in L\right) > \frac{2}{3}$$

$$p\left(f_{\mathcal{T}}(x) = 1 | x \notin L\right) \leq \frac{1}{3}$$

9.3.4 Komplexitätsklasse PP

Äquivalent zu BPP liegt in der Klasse „probabilistic polynomial time“ die Wahrscheinlichkeit die korrekte Antwort zurückzugeben mit größer $\frac{1}{2}$ vor.

9.4 Beispiel Guess-Path

Als Beispiel wird hier der Algorithmus `Guess-Path` gezeigt, der REACH auf einer PTM löst. Die Maschine bewegt sich zufällig zwischen den Knoten des Graphen hin und her. Diese zufällige Bewegung beschreibt eine *Markov-Kette*. Unter einer Markovkette versteht man einen stochastischen Prozess deren nächster Zustand *nur* vom vorigen abhängig ist. Der Zustand beschreibt den gerade besuchten Knoten.

Gegeben sei ein ungerichteter Graph $G(V, E)$ mit $s, t \in V$ und $s \neq t$. Es sei zu entscheiden, ob ein Weg path (path $\subseteq E$, $\text{path}_1 = s$, $\text{path}_{|\text{path}|-1} = t$) gefunden werden kann, sodass von s ausgehend t erreicht werden kann. Es handelt sich somit um das `REACHABILITY`-Problem auf einer PTM.

$i := s$

`forever:`

    `if` $i = t$

        `return Yes`

    `select` j `randomly with` $(i, j) \in E$

    $i := j$

Der Zufall spielt also bei der Wahl des Nachbarknotens j zu einem gegebenen Knoten i eine Rolle. Wegen dem momentan besuchten Knoten i und den daraus resultierenden Nachbarknoten, hängt der nächste Schritt vom vorhergehenden ab.

Gegeben sei der Beispielgraph aus Abbildung 9.3. Die bedingten Wahrscheinlichkeiten von einem Knoten $t-1$ zu einem Knoten t zu kommen, lassen sich aus der Abbildung ablesen.

$$P(X_t = 2 \mid X_{t-1} = 1) = \frac{1}{2}$$

$$P(X_t = 3 \mid X_{t-1} = 1) = \frac{1}{2}$$

$$P(X_t = 4 \mid X_{t-1} = 1) = 0$$

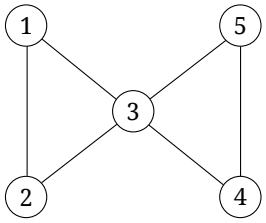

Abbildung 9.3: Beispielgraph für Guess-Path Algorithmus

Hierbei sollen die bedingten Wahrscheinlichkeiten illustrieren, dass die Wahrscheinlichkeit für nächsten Zustand vollständig durch die Angabe *eines* vorigen Zustands ermittelt werden kann. Damit ist das Kriterium der Markov-Kette erfüllt.

Es sei $R = \{1, 3, 4\}$ ein Random Walk von $s = 1$ nach $t = 4$. Die Wahrscheinlichkeit von s nach t mit dem Entscheidungspfad R zu kommen, ist gleich dem Produkt der Teilwahrscheinlichkeiten.

$$p\left(R_t = 3 \mid R_{t-1} = 1\right) = \frac{1}{2}$$

$$p\left(R_t = 4 \mid R_{t-1} = 3\right) = \frac{1}{4}$$

$$p\left(R_t = 5 \mid R_{t-1} = 4\right) = \frac{1}{2}$$

$$p\left(R\right) = \frac{1}{2} \cdot \frac{1}{4} \cdot \frac{1}{2} = \frac{1}{16}$$

Man bemerke, dass nur in ungerichteten, zusammenhängenden Graphen der Algorithmus versichern kann, dass von einem beliebigen Knoten s der Knoten t erreicht werden kann. Bei gerichteten Graphen kann eine Quelle t nicht mit Wahrscheinlichkeit 1 erreicht werden, da der Algorithmus in eine Sackgasse laufen könnte.

9.5 Beispiel `Monte-Carlo Algorithmus` für `2SAT`

Wir betrachten `2SAT` auf einer PTM. Wir möchten also für eine gegebene Formel ϕ in der Struktur einer 2-KNF eine erfüllende Belegung von boolschen Werten finden. Der Algorithmus beginnt bei einer beliebigen Belegung der Variablen.

In maximal m Schritten wird pro Schritt eine nicht-erfüllte Klausel zufällig gewählt. Aus dieser Klausel wird zufällig eines der Literale ausgewählt und negiert. Wurde nach der Negation eine erfüllende Belegung gefunden, bricht der Algorithmus mit einer Ja-Antwort ab.

Wir betrachten nun das Beispiel 9.1.

$$\Phi(x_1, x_2, x_3, x_4) = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (x_4 \vee \neg x_3) \wedge (x_4 \vee \neg x_1)$$

Wir starten mit der zufälligen Belegung $\{0, 0, 1, 0\}$. Im 1. Schritt hat der Zufall in der Entscheidung zwischen Klausel 3 und 4 zugunsten der 4. Klausel entschieden und daraus zufällig das Literal x_3, so ergibt sich zum 2. Schritt die Ausgangsbelegung, in der alle Variablen auf 0 stehen. Im 2. Schritt wird, wieder zufällig, die nicht erfüllte 3. Klausel gewählt und daraus zufällig das Literal x_2.

Der Algorithmus findet mit der 7. Iteration eine erfüllende Belegung und terminiert mit Ja.

Beispiel 9.1

x_1	x_2	x_3	x_4	$(x_1 \vee \neg x_2)$	$(\neg x_1 \vee \neg x_3)$	$(x_1 \vee x_2)$	$(x_4 \vee \neg x_3)$	$(x_4 \vee \neg x_1)$
0	0	1	0	✓	✓	×	×	✓
0	0	0	0	✓	✓	×	✓	✓
0	1	0	0	×	✓	✓	✓	✓
1	1	0	0	✓	✓	✓	✓	×
0	1	0	0	×	✓	✓	✓	✓
1	1	0	0	✓	✓	✓	✓	×
1	1	0	1	✓	✓	✓	✓	✓

Tabelle 9.1: 2-SAT Beispiel auf einer PTM

Der Algorithmus garantiert, dank beschränkendem m, zu terminieren. Ohne diese Schranke könnte der Algorithmus zwischen 2 Belegungen alternieren und damit unendlich lange laufen.

9.5.1 2SAT ist in RP

Für obigen Algorithmus gilt:[3]

- Wähle m (Anzahl der Schleifendurchläufe) mit $m = 2 \cdot l^2$
- $p\left(Z_0 > 2 * l^2\right) \leq \frac{1}{2} \rightarrow p\,(\text{Ja}) \leq \frac{1}{2}$

Daraus folgt folgender Satz:

Satz 9.1. *2SAT ist in* RP.

Um dies zu beweisen müssen wir zeigen, dass der Algorithmus nach polynomiell vielen Schritten mit einer Wahrscheinlichkeit von $\geq \frac{1}{2}$ eine erfüllende Belegung findet.

Interessant ist nun, nach jedem Schritt, wieviele Literale der einer erfüllenden Lösung entsprechen. Wir nutzen diese Darstellung zur Beschreibung unseres Random Walks. Sei S die Anzahl der Bits, die mit unserer erfüllenden Belegung überein stimmen. Folglich ist $S = \{0, 1, 2, 3, 2, 3, 4\}$ in unserem Beispiel die erfüllende Belegung.

Für eine Worst-case-Analyse kann man davon ausgehen, dass es nur 1 Lösung gibt. Mehrere Lösungen würden die Wahrscheinlichkeit des Findens einer Lösung erhöhen. Das Problem wäre also einfacher.

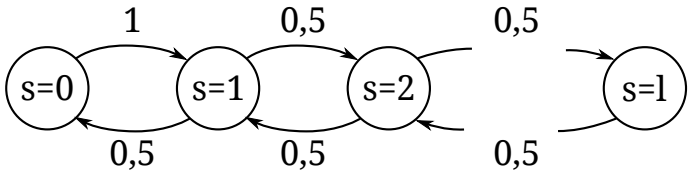

Abbildung 9.4: X sei als Entscheidungspfad beschrieben. Wir starten an einem beliebigen Knoten und wandern per Zufall 1 Schritt nach links oder rechts. Unser Ziel bildet dabei $s = l$; jener Knoten in dem alle Variablen eine erfüllende Belegung definieren

$$X^* = \{x_1^*, x_2^*, \ldots, x_l^*\} \qquad \text{sodass } \phi \text{ erfüllt ist}$$

Wir befinden uns an einem beliebigen Knoten in Abbildung 9.4. Wir wählen stets den linken oder rechten Weg (z_{j-1} oder z_{j+1}). z_j beschreibt eine Zufallsvariable, die die Entfernung von $s = l$ definiert. Die $+1$ beschreibt den zusätzlichen getätigen Schritt und $\frac{1}{2}$ beschreibt die Wahrscheinlichkeit, mit der eine Richtung gewählt wird. Wird s = l hat man die erfüllende Lösung gefunden. Der Abstand zur Lösung kann wieder als Zufallsprozess beschrieben werden. In jedem Schritt wird also, durch Flippen, der Abstand größer oder kleiner. Wir erhalten für den Erwartungswert

$$\mathbb{E}\,[z_j] = \mathbb{E}\left[\frac{1}{2}(1 + z_{j-1}) + \frac{1}{2}(1 + z_{j+1})\right]$$

[3] Mitzenmacher und Upfal, *Probability and computing: Randomized algorithms and probabilistic analysis.*

$$h_j = \mathbb{E}[z_j] = \frac{1}{2}(1 + h_{j-1}) + \frac{1}{2}(h_{j-1} + 1)$$
$$h_l = 0$$
$$h_0 = h_1 + 1$$
$$h_j = h_{j-1} + 2j + 1$$

Wir wollen diese Behauptung mittels einer vollständigen Induktion beweisen:

Basis $(j = 0)$

$$h_0 = h_1 + 2 \cdot 0 + 1 = h_1 + 1 \qquad \checkmark$$

Schritt $(j > 0)$

$$\begin{aligned} h_j &= \frac{1}{2}(1 + h_{j-1}) + \frac{1}{2}(1 + h_{j+1}) \\ \Leftrightarrow -\frac{1}{2}h_{j+1} &= \frac{1}{2}h_{j-1} + h_j + 1 \\ \Leftrightarrow h_{j+1} &= 2h_j - h_{j-1} - 2 \\ &= 2h_j - (h_j + 2(j-1) + 1) - 2 \\ &= h_j + 2j + 1 \qquad \square \end{aligned}$$

$$\mathbb{E}[Z_0] = h_0 = h_1 + 1 = h_2 + 1 + 3 = \ldots = \sum_{j=0}^{i-1}(2j + 1) = l^2$$

Daher, wenn Φ eine erfüllende Belegung besitzt wird diese im Erwartungswert nach l^2 Iterationen gefunden. Nun verwenden wir die *Markov-Ungleichung* um den Satz 9.1 zu beweisen. Sei X eine Zufallsvariable mit positiven Werten. Es gilt für alle $a > 0$ (Markov-Ungleichung):

$$P(X \geq a) = \frac{\mathbb{E}[x]}{a}$$

Beweis: Sei I eine Zufallsvariable mit

$$I = \begin{cases} 1 & \text{wenn } x \geq a \\ 0 & \text{sonst} \end{cases}$$

$$I \leq \frac{x}{a}$$
$$\mathbb{E}[I] = 1 \cdot p(I = 1) + 0 \cdot p(I = 0)$$
$$\mathbb{E}[I] = p(I = 1) = p(X \geq a)$$
$$\mathbb{E}[I] \leq \mathbb{E}\left[\frac{x}{a}\right]$$
$$\Leftrightarrow P(X \geq a) \leq \frac{\mathbb{E}[X]}{a}$$

Wir benutzen dies und wählen m (Anzahl der Schleifendurchläufe) mit $m = 2 \cdot l^2$. Daraus folgt:

$$p\left(Z_0 > 2 \cdot l^2\right) \leq \frac{1}{2} \rightarrow p\,(\text{Ja}) \geq \frac{1}{2},$$

was zu zeigen war.

9.6 Beispiel Monte-Carlo Algorithmus für 3SAT

Bei 3SAT liegt das Problem äquivalent zu 2SAT vor, jedoch können bis zu 3 Literale in einer Klausel vorkommen. Da 3SAT NP-Vollständing ist, ist zu erwarten, dass auch der randomisierte Algorithmus eine wesentlich höhere Laufzeit hat. Tatsächlich gilt, dass 3SAT nicht in RP liegt.

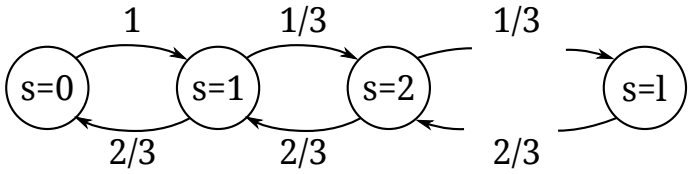

Abbildung 9.5: Äquivalent zu Abbildung 9.4 wird bei 3SAT für 3 Literale entschieden

Erwartungswert:

$$P(S_{i+1} = j+1 \mid S_i = j) = \frac{1}{3}$$
$$P(S_{i+1} = j+1 \mid S_i = j) = \frac{2}{3}$$
$$h_j = E[Z_j]$$
$$h_j = \frac{2}{3}h_{j-1} + \frac{1}{3}h_{j+1} + 1$$
$$h_j = 0$$
$$h_0 = h_1 + 1$$

Zeige durch vollständige Induktion:

$$h_j = h_{j+1} + 2^{j+2} - 3$$

Basis $(j = 0)$

$$h_0 = h_1 + 4 - 3 = h_1 + 1 \qquad \checkmark$$

Schritt $(j > 0)$

$$\begin{aligned} h_j &= \frac{2}{3}h_{j-1} + \frac{1}{3}h_{j+1} + 1 \\ \Leftrightarrow -\frac{2}{3}h_{j-1} &= -h_j + \frac{1}{3}h_{j+1} + 1 \\ \Leftrightarrow h_{j-1} &= \frac{3}{2}h_j - \frac{1}{2}h_{j+1} - \frac{3}{2} \\ \Leftrightarrow h_{j-1} &= \frac{3}{2}h_j - \frac{1}{2}\left(h_j - 2^{j+1} + 3\right) - \frac{3}{2} \\ \Leftrightarrow h_{j-1} &= h_j + 2^{j+1} - 3 \qquad \square \end{aligned}$$

$$\begin{aligned} E[Z_0] = h_0 = h_1 + 1 = h_2 + 1 + 5 = h_2 + 2^4 + 2^3 - 6 \\ = \ldots = 2^{l+2} - 2^2 - 3l \end{aligned}$$

$\Leftrightarrow$ Der Erwartungswert entwickelt sich exponentiell zur Anzahl der Literale

Literatur

Agrawal, M., N. Kayal und N. Saxena. »Primes in P«. In: *Annals of Mathematics* 160 (2004), S. 781–793.

Asteroth, Alexander und Christel Baier. *Theoretische Informatik. Eine Einführung in Berechenbarkeit, Komplexität und formale Sprachen.* München: Pearson Studium, 2003. ISBN: 3-8273-7033-7.

Gill III, John T. »Computational complexity of probabilistic Turing machines«. In: *Proceedings of the sixth annual ACM symposium on Theory of computing.* STOC '74. Seattle, Washington, United States: ACM, 1974, S. 91–95. DOI: 10.1145/800119.803889. URL: http://doi.acm.org/10.1145/800119.803889.

Mitzenmacher, M. und E. Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis.* Cambridge Univ Pr, 2005. ISBN: 978-0-512-83540-4.

Papadimitriou, C.H. *Computational Complexity.* John Wiley und Sons Ltd., 1994. ISBN: 0-201-53082-1.

Schöning, U. *Theoretische Informatik - kurz gefasst.* BI-Wissenschaftsverlag, 1992. ISBN: 978-3411156412.

Sipser, Michael. *Introduction to the Theory of Computation (Second Edition).* Boston, Massachusetts, 02210: Thomson Course Technology, 2006. ISBN: 978-0534950972.

Steger, Angelika. *Diskrete Strukturen.* 2. Aufl. Bd. Kombinatorik, Graphentheorie, Algebra. Springer-Lehrbuch. Berlin, Heidelberg: Springer, 2007. ISBN: 978-3540466604.

Index