Selbstreduzierbarkeit

Dennis Thomsen

2. März 2021

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Bachelorarbeit/Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 2. März 2021

_____

Vorname Nachname

Inhaltsverzeichnis

1 Einleitung

Ein Thema in der theoretischen Informatik ist das Bestimmen der Komplexität von Problemen beziehungsweise Sprachen. Komplexität kann beispielsweise daran gemessen werden, wie viel Zeit oder Speicher für das Lösen des Problems beziehungsweise das Entscheiden der Sprache benötigt wird. Anhand dieser Maße kann man dann solche Probleme und Sprachen in Komplexitätsklassen unterteilen. Diese Klassen können dann in Beziehung zueinander stehen, wobei nicht zwangsläufig klar ist, wo klare Grenzen sind.

Eine übliche Methode die Komplexität eines Problems einzuschätzen ist ein Berechnungsmodell auszuwählen und für dieses einen Algorithmus zu entwerfen, welches das Problem löst. Von diesem Algorithmus kann man dann den Zeitbedarf beziehungsweise den Speicherbedarf einschätzen. Allerdings kann man nicht zwangsläufig Erkenntnisse, die man mit einem solchen Algorithmus erlangt hat, auf andere ähnliche Probleme oder Sprachen anwenden.

Es gibt aber ein Werkzeug in der theoretischen Informatik, mit dem das Lösen eines Problems helfen kann ein anderes zu lösen. Dieses Werkzeug ist Reduktion. Es existieren verschiedene Arten von Reduktionen, die sich durch ihre Funktionsweise oder Komplexität unterscheiden.

Anstatt direkt einen Algorithmus zu entwickeln um ein gegebenes Problem zu lösen beziehungsweise eine gegebene Sprache zu entscheiden, kann es manchmal reichen zu zeigen, dass bestimmte Eigenschaften vorhanden sind. Solche Probleme und Sprachen kann man wieder in eigene Klassen unterteilen und allgemein Algorithmen entwickeln, die ganze Klassen lösen können.

Eine besondere Eigenschaft, welche diese beiden Möglichkeiten der Komplexitätsbestimmung verbindet, ist die sogenannte Selbstreduktion. Hierfür wird die Reduktion erweitert in dem Probleme beziehungsweise Sprachen auf eine Teilmenge von sich selbst reduziert werden. Dies hat Ähnlichkeiten zu einem in der Informatik bekannten Konzept, nämlich Divide-and-Conquer. Bei Divide-and-Conquer teilt man ein Problem in mehrere Teilprobleme, und löst das ursprüngliche Problem durch das Zusammenführen der Lösungen der Teilprobleme. Bekannte Beispiele hierfür sind QuickSort und MergeSort. Der Unterschied zu Selbstreduzierbarkeit ist, dass die Teilprobleme bei Divide-and-Conquer andere Probleme als das ursprüngliche Problem sein können, während bei Selbstreduzierbarkeit nur kleinere Instanzen des gleichen Problems verwendet werden. Diese Arbeit wird sich mit den Grundlagen von Selbstreduzierbarkeit auseinandersetzen.

In Kapitel 2 sollen zunächst die theoretischen Grundlagen gezeigt und erklärt werden, die für das Verständnis der restlichen Arbeit nötig sind. Speziell das Thema Reduktion ist für diese Arbeit relevant, da Selbstreduzierbarkeit darauf aufbaut.

In Kapitel 3 werden die grundlegenden Definition von Selbstreduzierbarkeit gezeigt.

Zunächst wird beschrieben, wann eine Instanz eines Problems kleiner als eine andere Instanz ist. Dies wird mit einer besonderen Art von Ordnung realisiert. Danach werden ausgewählte Arten von Selbstreduzierbarkeit vorgestellt und zuletzt werden Beziehungen zwischen diesen Arten aufgezeigt.

In Kapitel 4 werden die im vorherigen Kapitel vorgestellten Selbstreduzierbarkeiten in die Hierarchie der Komplexitätsklassen, wie sie in Kapitel 2 gezeigt wird, eingeordnet. Hierfür werden Turingmaschinen entworfen, die die Selbstreduzierbarkeit einer Sprache nutzen, um diese mit bestimmten Zeitbedarf beziehungsweise Speicherbedarf zu entscheiden.

In Kapitel 5 soll die Relevanz von Selbstreduzierbarkeit aufgezeigt werden. Hierfür soll gezeigt werden, dass die bekannte Sprache SAT selbstreduzierbar ist. Darüberhinaus soll an einem Beispiel gezeigt werden, wie sich durch Hinzufügen weiterer Eigenschaften zu einer selbstreduzierbaren Sprache die Komplexität dieser Sprache ändern kann.

2 Grundlagen

In diesem Kapitel sollen Kenntnisse der theoretischen Informatik einmal vorgestellt oder wiederholt werden, welche für das Verständnis von Selbstreduzierbarkeit notwendig sind.

2.1 Komplexitätsklassen

In der Komplexitätstheorie gibt es verschiedene Maße mit denen man Sprachen in Klassen unterteilen kann. Zwei besonders wichtige sind der Zeitbedarf und der Speicherbedarf, also die Zeit beziehungsweise die Menge an Speicher, die benötigt wird um die Sprache zu entscheiden.

Als grundlegendes Modell zur Einschätzung dieser Maße soll in dieser Arbeit eine Variante der Turingmaschine verwendet werden. Eine Turingmaschine habe ein Eingabeband, auf dem das Eingabewort steht, eine beliebige Anzahl von Arbeitsbändern und ein Ausgabeband, auf dem am Ende der Berechnung die Antwort der Maschine stehen wird.

Turingmaschinen sind eines der mächstigsten Berechnungsmodelle. Laut der Church-Turing-These exisitert für jedes berechenbare Problem eine Turingmaschine, die dieses Problem berechnet. Man kann zum Beispiel Turingmaschinen entwerfen, die Funktionen berechnen sollen, beispielsweise eine Turingmaschine, welche die Summe von zwei Zahlen bestimmt. Am Ende der Berechnung soll das Ergebnis der Funktion auf dem Ausgabeband geschrieben sein.

Eine andere Anwendung von Turingmaschinen ist das Entscheiden von Sprachen. Darunter versteht man das Bestimmen, ob ein gegebenes Wort zur ausgewählten Sprache gehört. So eine Turingmaschine erhält dann ein Eingabewort und schreibt als Ergebnis eine 1 oder eine 0 auf das Ausgabeband. Eine 1 steht dafür, dass das Eingabewort zur Sprache gehört. Eine 0 wiederum steht dafür, dass das Eingabewort nicht zur Sprache gehört. Hierzu sagt man auch, dass die Turingmaschine das Eingabewort akzeptiert beziehungsweise ablehnt. Ein Beispiel einer Sprache, die sich von einer Turingmaschine entscheiden lässt, ist die Sprache aller binärer Zahlen, die durch 2 teilbar sind.

Man kann bei Turingmaschinen zwischen zwei Arten von Berechnungen unterscheiden, einer deterministischen Art und einer nichtdeterministischen Art. Eine nichtdeterministische Turingmaschine hat einen Berechnungsbaum, der aus mehreren Berechnungspfaden besteht. Eine nichtdeterministische Turingmaschine akzeptiert ein Eingabewort, sobald einer dieser Berechnungspfade zu einem akzeptierenden Zustand führt. Ansonsten lehnt die nichtdeterministische Turingmaschine das Eingabewort ab. Eine deterministische Turingmaschine wiederum hat nicht einen Berechnungsbaum sondern nur einen einzigen Berechnungspfad. Wenn dieser in einen akzeptierenden Zustand führt, dann akzeptiert

die deterministische Turingmaschine das Eingabewort, sonst lehnt sie es ab. Zu beachten ist auch, dass jede nichtdeterministische Turingmaschine von einer deterministischen Turingmaschine simuliert werden kann.

Das erste Maß für Komplexität soll der Zeitbedarf sein. Der Zeitbedarf einer Turingmaschine ist die Anzahl der Berechnungsschritte, die diese Turingmaschine benötigt um das Ergebnis zu berechnen und zu halten.

In dieser Arbeit wird der Zeitbedarf in Abhängigkeit der Länge des Eingabewortes mit streng monoton steigenden Funktionen beschrieben. Diese Funktionen kann man in Klassen unterteilen, die beschreiben, wie schnell sie wachsen. Dafür kann die $\mathcal{O}$-Notation verwendet werden. Wenn $f(n)$ eine Funktion ist, dann gehören zur Klasse $\mathcal{O}(f(n))$ all jene Funktionen $g(n)$, für die gilt $\exists n_0 \in \mathbb{N}, \exists c \in \mathbb{N}, \forall n \geq n_0 \colon cf(n) \geq g(n)$. Die Klasse $\mathcal{O}(f(n))$ umfasst also alle Funktionen $g(n)$, die nicht schneller wachsen als $f(n)$.

Es soll zwischen zwei grundlegenden Arten von Zeitbedarf unterschieden werden, zum einen den Zeitbedarf einer deterministischen Turingmaschine und zum anderen den Zeitbedarf einer nichtdeterministischen Turingmaschine. $\mathrm{TIME}(t(n))$ ist die Klasse der Sprachen, die von deterministischen Turingmaschinen entschieden werden können und diese Turingmaschinen jeweils einen Zeitbedarf haben, der in $\mathcal{O}(t(n))$ ist, wobei n die Länge des Eingabeworts ist. Analog dazu lässt sich $\mathrm{NTIME}(t(n))$ für nichtdeterministische Turingmaschinen definieren.

Das zweite Maß für Komplexität, welches in dieser Arbeit verwendet wird, soll der Speicherbedarf sein. Der Speicherbedarf einer Turingmaschine ist die Anzahl der Bandzellen, die auf den Arbeitsbändern während der Berechnung verwendet werden. Hierbei ist irrelevant, wie oft die einzelnen Zellen beschrieben werden, jede bereits verwendete Zelle kann beliebig oft verwendet werden. Es ist nur wichtig, dass sie mindestens einmal verwendet werden. Zu beachten ist außerdem, dass die Zellen des Eingabebands nicht zum Speicherbedarf beitragen. Denn ansonsten hätte jede Turingmaschine einen Speicherbedarf, der mindestens der Länge des Eingabeworts entspricht.

Auch beim Speicherbedarf soll zwischen zwei Arten unterschieden werden, eine Art für deterministische Turingmaschinen und eine Art für nichtdeterministische Turingmaschinen. $\mathrm{SPACE}(s(n))$ soll hierbei die Klasse der Sprachen sein, die von deterministischen Turingmaschinen entschieden werden können, welche einen Speicherbedarf, der in $\mathcal{O}(s(n))$ ist, wobei n die Länge des Eingabeworts ist. Wieder lässt sich analog dazu $\mathrm{NSPACE}(s(n))$ für nichtdeterministische Turingmaschinen definieren.

Mit diesen beiden Maßen ist man nun dazu in der Lage Sprachen in Klassen zu unterteilen. Für diese Arbeit wird sich auf wohl bekannte Klassen beschränkt, die folgendermaßen definiert sind.

- $\mathrm{P} = \bigcup_{k \in \mathbb{N}} \mathrm{TIME}(n^k)$
- $\mathrm{NP} = \bigcup_{k \in \mathbb{N}} \mathrm{NTIME}(n^k)$
- $\mathrm{PSPACE} = \bigcup_{k \in \mathbb{N}} \mathrm{SPACE}(n^k)$

Für diese Komplexitätsklassen gilt die Teilmengenbeziehung $\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE}$, wobei nicht bekannt ist, ob es sich hierbei um echte Teilmengenbeziehungen handelt.

Besonders die Beziehung P $\subseteq$ NP ist von besonderer Relevanz, da viele Probleme in der Praxis in die Klasse NP fallen und es unbekannt ist, ob diese effizient berechnet beziehungsweise entschieden werden können. Effizient heißt hierbei, dass sie von einer deterministischen Turingmaschine in polynomieller Zeit berechnet beziehungsweise entschieden werden, also dass sie zur Klasse P gehören.

Manchmal kann es wichtig sein Sprachen anhand dessen zu klassifizieren, zu welcher Klasse das Komplement der Sprache gehört. Relevant in dieser Arbeit ist dabei die Klasse coNP = $\{A \mid \overline{A} \in \text{NP}\}$.

2.2 Orakelmaschinen

Für alle Sprachen, die in den hier gezeigten Komplexitätsklassen sind, lassen sich Turingmaschinen entwerfen, die diese Sprachen deterministisch oder nichtdeterministisch unter bestimmten Vorgaben bezüglich Zeitbedarf und Speicherbedarf entscheiden können. Hierbei ist es aber notwendig alle benötigten Schritte dieser Berechnung explizit anzugeben. Manchmal kann es hilfreich sein sich anzuschauen, wie komplex eine Sprache ist unter der Annahme, dass eine ausgewählte Sprache oder Menge von Sprachen sehr einfach zu entscheiden ist. Dies nennt man relativierte Komplexität, man bestimmt die Komplexität von Sprachen relativ zu einer Sprache oder einer Menge von Sprachen. Ein Berechnungsmodell für diese relativierte Komplexität ist die sogenannte Orakelmaschine, eine Erweiterung der Turingmaschine.

Eine Orakelmaschine ist eine Turingmaschine mit folgenden Erweiterungen. Zunächst hat eine Orakelmaschine ein zusätzliches Arbeitsband mit dediziter Aufgabe. Dieses Arbeitsband wird Orakelband genannt. Weiterhin hat eine Orakelmaschine drei spezielle Zustände, QUERY, YES und NO. Zuletzt hat eine Orakelmaschine eine ausgewählte Sprache, die als Orakel bezeichnet wird. [**Stock77**]

Wenn eine Orakelmaschine in den Zustand QUERY geht, wird sie im nächsten Berechnungsschritt in den Zustand YES oder NO gehen, je nachdem ob das Wort, was zu diesem Zeitpunkt auf dem Orakelband steht, zur Sprache des Orakels gehört oder nicht. Wenn das Wort zur Sprache des Orakels gehört, geht die Maschine in den Zustand YES, wenn das Wort nicht zur Sprache des Orakels gehört, geht die Maschine in den Zustand NO. Diesen Prozess versteht man so, dass eine Frage an das Orakel gestellt wird und diese in einem Berechnungsschritt beantwortet wird. Es wird gefragt, ob das Wort auf dem Orakelband zur Sprache des Orakels gehört. Eine Orakelmaschine beendet ihre Berechnung ähnlich wie eine normale Turingmaschine, sobald sie in einem Endzustand hält und das Eingabewort entweder akzeptiert oder ablehnt.

Ähnlich wie bei normalen Turingmaschinen kann man auch bei Orakelmaschinen einen Zeitbedarf beziehungsweise einen Speicherbedarf bestimmen. Hierbei werden diese genau so wie bei normalen Turingmaschinen bestimmt. Die einzige Ausnahme ist, dass das Beantworten einer Frage an das Orakel mit nur einem Berechnungsschritt gezählt wird, unabhängig von der eigentlichen Komplexität der Sprache des Orakels.

Sei C eine beliebige Komplexitätsklasse, wie sie in Abschnitt 2.1 definiert wurden. Sei A eine Sprache, die folgend als Orakel bezeichnet wird. Es gilt für eine Sprache B, dass $B \in C^A$, wenn eine Orakelmaschine M mit dem Orakel A existiert, die die Sprache

B entscheidet, und dabei einen Zeitbedarf $t(n)$ und einen Speicherbedarf $s(n)$ hat, so dass jede Turingmaschine mit gleichem Zeitbedarf und Speicherbedarf zur Klasse C gehört. Das heißt, wäre die Komplexität der Sprache A vernachlässigbar, so wäre die Sprache B in der Klasse C.

Nun sei $\mathcal{D}$ eine Menge von Sprachen. Dann gilt $C^D = \{L \mid L \in C^A, A \in D\}$. Das heißt, eine Sprache B gehört zur Klasse C^D, wenn es ein Orakel $A \in D$ gibt, so dass $B \in C^A$. [7]

So kann man zum Beispiel die relativierte Komplexitätsklasse P^{NP} definieren. Eine Sprache L gehört zu dieser Klasse, wenn eine deterministische Orakelmaschine mit einem Orakel $A \in \mathrm{NP}$ existiert, die L in polynomieller Zeit entscheidet.

2.3 Reduktion

Eine Reduktion ist ein Werkzeug in der theoretischen Informatik, mit dem man Aussagen über die Berechenbarkeit von Funktionen beziehungsweise die Entscheidbarkeit von Sprachen treffen kann. Darüber hinaus kann eine Reduktion auch bei der Zuordnung von Sprachen zu Komplexitätsklassen, wie sie in 2.1 definiert wurden, helfen.

Es existieren verschiedene Reduktionsmethoden, mit denen Reduktion durchgeführt werden können. Sei D eine beliebige Reduktionsmethode und sei C eine beliebige Komplexitätsklasse, wie sie in Abschnitt 2.1 definiert wurden. Wenn nun eine Sprache A C-D-reduzierbar auf eine Sprache B ist, schreibt man $A \leq_D^C B$.

Die Reduktionsmethoden können unterschiedlich mächtig sein. Als Beispiel sollen sich zwei Sprachen A und B angeschaut werden. C sei hier eine beliebige Komplexitätsklasse, wie sie in Abschnitt 2.1 definiert wurden. Darüberhinaus seien D_1 und D_2 zwei verschiedene Reduktionsmethoden. Wenn jetzt gilt, dass $A \leq_{D_1}^C B$, aber nicht $A \leq_{D_2}^C B$, dann ist D_1 mächtiger als D_2. Nur weil eine Sprache A mit einer ausgewählten Methode reduzierbar auf eine Sprache B ist, heißt das nicht, dass dies für jede beliebige Methode gilt.

In dieser Arbeit werden drei Reduktionsarten vorgestellt und verwendet. Diese sind die Many-One-Reduktion, die Truth-Table-Reduktion und die Turing-Reduktion. Als Komplexitätsklasse soll für jede dieser Reduktionen in dieser Arbeit die Klasse P verwendet werden. Selbstverständlich lassen sich auch andere Komplexitätsklassen verwenden. Die folgenden Definitionen stammen aus der Arbeit von Ladner et al.

Definition 1 ([5])**.** *Gegeben seien die Alphabete Σ und Γ sowie die Sprachen $A \subseteq \Sigma^*$ und $B \subseteq \Gamma^*$. A ist dann* in polynomieller Zeit Turing-reduzierbar auf *B, in Notation $A \leq_T^P B$, wenn eine Orakelmaschine M existiert, so dass folgende Aussage gilt:*

$\forall x \in \Sigma^* : x \in A \iff$ M akzeptiert x mit einem Orakel B in polynomieller Zeit.

Definition 2 ([5])**.** *Gegeben seien die Alphabete Σ und Γ sowie die Sprachen $A \subseteq \Sigma^*$ und $B \subseteq \Gamma^*$. A ist dann* in polynomieller Zeit Truth-Table-reduzierbar auf *B, in Notation $A \leq_{tt}^P B$, wenn eine Funktion $f(x) = (\alpha, y_1, .., y_k)$ existiert, wobei $x \in \Sigma^*$, $k \in \mathbb{N}$, α eine k-stellige Boolsche Funktion ist, $y_1, .., y_k \in \Gamma^*$, f in polynomieller Zeit berechenbar ist und C_B die charakteristische Funktion der Sprache B ist, so dass folgende Aussage gilt:*

$\forall x \in \Sigma^* : x \in A \iff \alpha(C_B(y_1), ..., C_B(y_k)) = 1.$

Definition 3 ([5]). *Gegeben seien die Alphabete* Σ *und* Γ *sowie die Sprachen* $A \subseteq \Sigma^*$ *und* $B \subseteq \Gamma^*$. *A ist* in polynomieller Zeit dann Many-One-reduzierbar auf B, *in Notation* $A \leq_m^P B$, *wenn eine Funktion* $f(x) = y$ *existiert, wobei* $x \in \Sigma^*$, $y \in \Gamma^*$ *und* f *in polynomieller Zeit berechenbar ist, so dass folgende Aussage gilt:*
$\forall x \in \Sigma^* : x \in A \iff f(x) \in B.$

Für alle Komplexitätsklassen, die in Abschnitt 2.1 definiert wurden, gilt, dass diese für alle hier genannten Reduktionen abgeschlossen sind. Damit ist gemeint, dass wenn für zwei Sprachen A und B gilt, dass $A \leq_D^C B$ gilt, und B entweder in P, NP oder PSPACE ist, dass auch A in der entsprechenden Klasse ist. Dies soll hier für die drei Reduktionen gezeigt werden.

Zunächst soll die Abgeschlossenheit für Turing-Reduktion gezeigt werden. Gegeben seien zwei Sprachen A und B. Für diese Sprachen soll $A \leq_T^P B$ gelten. Laut Definition 1 existiert dann eine Orakelmaschine M_A^B, die die Sprache A entscheidet unter Anwendung eines Orakels B. Diese Orakelmaschine hat wiederum einen polynomiellen Zeitbedarf.

Während der Berechnung von M_A^B werden Wörter auf das Orakelband geschrieben und das Orakel B gefragt, ob diese Wörter zur Sprache des Orakels gehören. Das Beantworten dieser Fragen benötigt dabei nur einen Berechnungsschritt. Wenn die Sprache B entscheidbar ist, so existiert eine Turingmaschine M_B, die diese entscheiden kann. Anstatt das Orakel zu befragen, kann man stattdessen M_B für jede Frage simulieren um diese zu beantworten. Dadurch erzeugt man aus der Orakelmaschine M_A^B die Turingmaschine M_A, die die Sprache A entscheidet.

Nun gilt für die ursprüngliche Orakelmaschine, dass sie einen polynomiellen Zeitbedarf hat. Zum einen bedeutet das, dass nur polynomiell viele Wörter auf das Orakelband beziehungsweise nur polynomiell viele Fragen an das Orakel gestellt werden. Zum anderen bedeutet das aber auch, dass M_A^B maximal einen polynomiellen Speicherbedarf haben kann, da in jedem Berechnungsschritt maximal eine neue Bandzelle verwendet werden kann.

Wenn nun $B \in$ P gilt, heißt das, M_B ist eine deterministische Turingmaschine mit polynomiellen Zeitbedarf. Wie bereits geschrieben, wird das Orakel maximal polynomiell oft befragt. Das bedeutet die Turingmaschine M_A wird nur polynomiell oft die Turingmaschine M_B simulieren. Da M_B nur polynomiell oft simuliert wird und darüberhinaus nur einen polynomiellen Zeitbedarf hat, folgt daraus, dass auch M_A einen polynomiellen Zeitbedarf haben muss. Außerdem, da M_A eine Turingmaschine mit polynomiellen Zeitbedarf ist, die die Sprache A entscheiden kann, gilt $A \in$ P und damit schließlich $A \leq_T^P B \wedge B \in \text{P} \implies A \in \text{P}$.

Wenn nun aber $B \in$ NP gilt, dann ist M_B eine nichtdeterministische Turingmaschine mit polynomiellen Zeitbedarf. Diese zu Simulieren führt dazu, dass auch M_A eine nichtdeterministische Turingmaschine ist. Am Zeitbedarf von M_A ändert sich allerdings nichts, weshalb dann $A \in$ NP und somit $A \leq_T^P B \wedge B \in \text{P} \implies A \in \text{P}$ gilt.

Wenn $B \in$ PSPACE gilt, dann ist M_B eine deterministische Turingmaschine mit polynomiellen Speicherbedarf. Wenn eine andere Turingmaschine M_B simuliert, wird auch diese einen polynomiellen Speicherbedarf haben. Da aber immer nur eine Simulation zur Zeit laufen muss und der bereits verwendete Speicher wiederverwendet

werden kann, wird insgesamt für die polynomiell vielen Simulationen wieder nur polynomiell viel Speicher benötigt. Da der Rest von M_A eine polynomielle Laufzeit haben muss, kann dieser maximal auch nur polynomiellen Speicherbedarf haben. Insgesamt ist M_A also immernoch eine deterministische Turingmaschine, die hierbei allerdings polynomiellen Speicherbedarf hat. Somit gilt $A \in \text{PSPACE}$ und schließlich $A \leq_T^P B \wedge B \in \text{PSPACE} \implies A \in \text{PSPACE}$.

Als nächstes wird die Truth-Table-Reduktion auf ihre Abgeschlossenheit untersucht. Es sollen wieder zwei Sprachen A und B gegeben sein, für welche jetzt $A \leq_{tt}^P B$ gilt. Laut Definition 2 existiert dann eine Reduktionsfunktion f, die bei Eingabe x insgesamt k Wörter $y_1, ..., y_k$ und eine k-stellig Boolsche Funktion α erzeugt und ausgibt. Hierfür wird nur polynomielle Zeit benötigt, was bedeutet, dass f durch eine deterministische Turingmaschine in polynomieller Zeit simuliert werden kann.

Es gilt hierbei $x \in A \iff \alpha(C_B(y_1), ..., C_B(y_k))$, wobei C_B die charakteristische Funktion der Sprache B ist. Für die weitere Betrachtung soll gelten, dass B entscheidbar ist. Daraus folgt, dass eine Turingmaschine M_B existiert, die die charakteristische Funktion C_B berechnet beziehungsweise die Sprache B entscheidet. Anstatt also C_B für die Auswetung von α zu verwenden, kann man stattdessen M_B verwenden.

Nun kann man eine Turingmaschine M_A konstruieren, die die Sprache A entscheidet. Diese bekommt als Eingabe x und simuliert zunächst die Reduktionsfunktion f mit x als Eingabe. Das Ergebnis von f wird dann auf Arbeitsbändern gespeichert. Danach wird die Turingmaschine M_B jeweils mit den Wörtern $y_1, ..., y_k$ simuliert und die Ergebnisse hintereinander auf ein freies Arbeitsband geschrieben. Zuletzt wird die durch f erzeugte Boolsche Funktion α auf diesem Arbeitsband simuliert, sodass jede 1 und 0 auf diesem Arbeitsband als eine Eingabe verwendet wird. Das Ergebnis von α wird schlussendlich auf das Ausgabeband geschrieben.

Da jede Eingabe von α das Ergebnis von M_B ist und dieses gleich dem Ergebnis von C_B ist, ist das Ergebnis der Simulation von α genau dann 1, wenn $x \in A$ gilt. Da diese Ausgabe direkt auf das Ausgabeband geschrieben wird, ist auch die Ausgabe von M_A genau dann 1, wenn $x \in A$ gilt. Somit entscheidet die Turingmaschine M_A die Sprache A.

Die Simulation von f ist in polynomieller Zeit möglich. Das Ergebnis dieser Funktion auf Arbeitsbänder zu schreiben benötigt offensichtlich auch nur polynomielle Zeit. Aufgrund der polynomiellen Laufzeit von f muss die Anzahl der Wörter $y_1, ..., y_k$ polynomiel beschränkt sein. Das heißt, M_B kann maximal nur polynomiel oft simuliert werden. Die Ergebnisse dieser Simulationen bestehen jeweils nur aus einem Symbol. Diese hintereinander auf ein Arbeitsband zu schreiben ist offensichtlich in polynomieller Zeit möglich.

Zuletzt wird α mit dem Inhalt dieses Arbeitsbandes als Eingabe simuliert. Es ist bekannt, dass die Auswertung eine Boolschen Funktion nur polynomielle Zeit in Abhängigkeit der Anzahl der Eingaben benötigt. Da diese Anzahl bereits polynomiell beschränkt ist, ist damit auch die Simulation von α in polynomieller Zeit möglich. Das Ergebnis davon auf das Ausgabeband zu schreiben benötigt nur konstante Zeit.

Damit benötigen alle Schritte der Turingmaschine M_A nur polynomielle Zeit und werden deterministisch durchgeführt. Nur die Simulation von M_B, welche polynomiell oft durchgeführt wird, bleibt noch offen. Hierfür muss bestimmt werden, welchen

Zeitbedarf beziehungsweise Speicherbedarf diese Turingmaschine hat.

Gilt nun $B \in \mathrm{P}$, dann folgt daraus, dass M_B eine deterministische Turingmaschine ist, die einen polynomiellen Zeitbedarf hat. Da diese von M_A polynomiell oft simuliert wird, benötigen alle Simulationen zusammen insgesamt polynomielle Zeit. Das bedeutet, dass auch M_A eine deterministische Turingmaschine mit polynomiellen Zeitbedarf ist. Aufgrunddessen gilt $A \in \mathrm{P}$ und somit auch $A \in P$ und $A \leq_{tt}^{P} B \wedge B \in \mathrm{P} \implies A \in \mathrm{P}$.

Wenn nun wiederum $B \in \mathrm{NP}$ gilt, ist M_B eine nichtdeterministische Turingmaschine mit polynomiellen Zeitbedarf. Das Einzige, was sich an M_A ändert, ist, dass die Simulationen von M_B damit auch nichtdeterministisch sind. Somit ist M_A ebenfalls eine nichtdeterministische Turingmaschine mit weiterhin polynomiellen Zeitbedarf. Daraus folgt $A \in \mathrm{NP}$ und $A \leq_{tt}^{P} B \wedge B \in \mathrm{NP} \implies A \in \mathrm{NP}$.

Wenn $B \in \mathrm{PSPACE}$ gilt, ist M_B eine deterministische Turingmaschine mit polynomiellen Speicherbedarf. Damit benötigt auch M_A für eine Simulation von M_B polynomiellen Speicher. Da aber der Speicher, der für eine Simulation benötigt wird, wiederverwendbar ist, wird für alle Simulationen von M_B insgesamt nur polynomieller Speicher benötigt. Alle anderen Schritte haben polynomiellen Zeitbedarf und können somit maximal nur polynomiellen Speicherbedarf haben. Insgesamt hat dann M_A auch polynomiellen Speicherbedarf und bleibt eine deterministische Turingmaschine. Folglich gilt $A \in \mathrm{PSPACE}$ und $A \in P$ und $A \leq_{tt}^{P} B \wedge B \in \mathrm{PSPACE} \implies A \in \mathrm{PSPACE}$.

Zuletzt soll noch die Abgeschlossenheit von Many-One-Reduktion gezeigt werden. Wieder seien die Sprachen A und B gegeben. Darüberhinaus soll $A \leq_{m}^{P} B$ gelten. Laut Definition 3 existiert dann eine Reduktionsfunktion f, die in polynomieller Zeit berechenbar ist. Das bedeutet, es existiert eine deterministische Turingmaschine, welche diese Funktion berechnet und einen polynomiellen Zeitbedarf hat.

Für die weitere Betrachtung soll gelten, dass B entscheidbar ist. Dann existiert eine Turingmaschine M_B, die die Sprache B entscheidet. Nun kann eine Turingmaschine M_A konstruiert werden, welche die Sprache A entscheidet. Diese soll das Eingabewort x nehmen und die Reduktionsfunktion f mit diesem als Eingabe simulieren. Die Ausgabe von f soll dann als Eingabe für M_B verwendet werden. Das von M_B berechnete Ergebnis soll schlussendlich auf das Ausgabeband geschrieben werden.

Wird f auf das Eingabewort x simuliert, so erhält man als Ausgabe das Wort y. Wenn für dieses Wort dann M_B eine 1 ausgibt, gilt $y \in B$. Gibt M_B eine 0 aus, so gilt stattdessen $y \notin B$. Da y durch die Reduktionsfunktion erzeugt wurde, gilt entsprechend auch für das Eingabewort x entweder $x \in A$ beziehungsweise $x \notin A$. Dies entspricht der Ausgabe von M_A, womit diese die Sprache A korrekt entscheidet.

Das Simulieren von f benötigt wie bereits beschrieben nur polynomielle Zeit. Das Ergebnis von M_B auf das Ausgabeband zu schreiben benötigt nur konstante Zeit. Schließlich fehlt zu bestimmen, wie es mit der Simulation von M_B aussieht.

Wenn nun $B \in \mathrm{P}$ gilt, dann heißt das, dass M_B eine deterministische Turingmaschine mit polynomiellen Zeitbedarf ist. Dann benötigt M_A für die Simulation von M_B auch nur polynomielle Zeit. Damit benötigt M_A insgesamt nur polynomielle Zeit und ist darüberhinaus deterministisch. Daraus folgt $A \in \mathrm{P}$ und außerdem $A \leq_{m}^{P} B \wedge B \in \mathrm{P} \implies A \in \mathrm{P}$.

Wenn wiederum $B \in \mathrm{NP}$ gilt, heißt dass M_B ist eine nichtdeterministische Turingmaschine mit polynomiellen Zeitbedarf. Am Rest der Argumentation verändert sich

allerdings nichts, nur dass M_A aufgrund der Simulation von M_B ebenfalls eine nichtdeterministische Turingmaschine ist. Da M_A dennoch einen polynomiellen Zeitbedarf hat, gilt nun $A \in \text{NP}$ und somit $A \leq_m^P B \wedge B \in \text{NP} \implies A \in \text{NP}$.

Wenn stattdessen $B \in \text{PSPACE}$ gilt, ist M_B eine deterministische Turingmaschine mit polynomiellen Speicherbedarf. Für M_A gilt, dass auch diese maximal polynomiellen Speicherbedarf haben kann. Die Reduktionsfunktion f zu simulieren benötigt polynomielle Zeit, somit kann dafür auch nur polynomieller Speicher benötigt werden. Da sowohl f als auch M_B polynomiellen Speicherbedarf haben und der von f verwendete Speicher wiederverwendet werden kann, ist nur das Maximum dieser beiden ausschlaggebend. Daraus folgt schließlich $A \in \text{PSPACE}$ und damit auch $A \leq_m^P B \wedge B \in \text{PSPACE} \implies A \in \text{PSPACE}$.

Damit ist für alle in dieser Arbeit verwendeten Reduktionen gezeigt, dass sie für dier hier verwendeten Komplexitätsklassen abgeschlossen sind.

2.4 Vollständigkeit

Reduktion kann als Hilfsmittel verwendet werden um Aussagen über Klassen von Sprachen zu treffen. Beispielsweise lassen sich daruch Aussagen über Komplexitätsklassen treffen.

Sei K eine beliebige Komplexitätsklasse, wie sie in Abschnitt 2.1 definiert wurden. Für jede Reduktion $\leq_D^C$ lassen sich jeweils nun die Klasse der K-schweren Sprachen und die Klasse der K-vollständigen Sprachen wie folgt definieren.

Definition 4. *Gegeben sei das Alphabet Σ und die Sprache $B \subseteq \Sigma^*$. Die Sprache B ist* K-schwer über Reduktion $\leq_D^C$*, wenn $\forall A \in K \colon A \leq_D^C B$ gilt.*

Definition 5. *Gegeben sei das Alphabet Σ und die Sprache $B \subseteq \Sigma^*$. Die Sprache B ist* K-vollständig über Reduktion $\leq_D^C$*, wenn B K-schwer über Reduktion $\leq_D^C$ ist und $B \in K$ gilt.*

Eine Sprache ist also K-schwer über $\leq_D^C$, wenn jede Sprache aus K reduzierbar auf diese Sprache ist. Wenn diese Sprache darüberhinaus auch selbst in K ist, dann ist sie sogar K-vollständig.

Nun kann man verschiedene Arten von Schwere und Vollständigkeit für jede Klasse definieren, welche unterschiedliche Verwendungszwecke haben können. In dieser Arbeit wird Schwere und Vollständigkeit mit $\leq_m^P$ verwendet. Mit dieser Reduktion sollen nun die Klassen der NP-schweren, NP-vollständigen, PSPACE-schweren und PSPACE-vollständigen Sprachen definiert werden.

Definition 6. *Gegeben sei das Alphabet Σ, sowie die Sprache $B \subseteq \Sigma^*$. Dann ist B* NP-schwer*, wenn $\forall A \in NP$: $A \leq_m^P B$ gilt. B ist darüberhinaus* NP-vollständig*, wenn zusätzlich $B \in NP$ gilt.*

Definition 7. *Gegeben sei das Alphabet Σ, sowie die Sprache $B \subseteq \Sigma^*$. Dann ist B* PSPACE-schwer*, wenn $\forall A \in PSPACE$: $A \leq_m^P B$ gilt. B ist darüber hinaus* PSPACE-vollständig*, wenn zusätzlich $B \in PSPACE$ gilt.*

Mit diesen Definitionen zu Vollständigkeit kann man versuchen Fragen wie P = NP? und P = PSPACE? zu beantworten. Denn wenn man es schafft zu zeigen, dass für eine NP-vollständige Sprache A gilt $A \in$ P, dann folgt daraus P = NP. Selbiges gilt dann auch für PSPACE mit einer PSPACE-vollständigen Sprache A.

In Abschnitt 2.3 wurde bereits gezeigt, dass die Klassen P, NP und PSPACE abgeschlossen über die in dieser Arbeit verwendeten Reduktionen sind. Insbesondere sind sie also abgeschlossen über $\leq_m^P$. Wenn nun für eine Sprache B gilt, dass sie NP-vollständig ist, dann gilt für alle Sprachen $A \in$ NP, dass $A \leq_m^P B$. Wenn darüberhinaus gilt, dass $B \in$ P, folgt aufgrund der Abschlusseigenschaft, dass dann auch für die genannten Sprachen $A \in$ P gilt. Damit gilt NP $\subseteq$ P und da bereits P $\subseteq$ NP gilt, folgt schließlich P = NP.

Gilt wiederum B ist $PSPACE$-vollständig, so gilt für alle Sprachen $A \in$ PSPACE, dass $A \leq_m^P B$. Wenn für B wieder gilt, dass $B \in$ P, dann folgt durch die Abschlusseigenschaft, dass für die genannten Sprachen $A \in$ P gilt. Daraus folgt wiederum, dass PSPACE $\subseteq$ P. Da aber auch P $\subseteq$ PSPACE gilt, folgt $P =$ PSPACE.

Natürlich wurden bisher keine Sprachen gefunden, die NP-vollständig beziehungsweise PSPACE-vollständig und in der Klasse P sind. Dennoch könnte durch Vollständigkeit zusammen mit Reduktion eine Antwort auf diese Fragen gefunden werden.

3 Grundlegende Definitionen

Selbstreduktion ist eine Erweiterung des normalen Reduktionsbegriffs. Hierbei geht es nicht darum, ob eine Sprache A auf eine andere Sprache B reduzierbar ist, sondern ob eine Sprache A auf sich selbst reduzierbar ist, daher Selbstreduktion. Selbstverständlich müssen dafür Einschränkungen eingeführt werden, da sonst jede Sprache trivialerweise auf sich selbst reduzierbar wäre.

3.1 Polynomially related Ordnungen

Dafür werden die Wörter, die durch eine Reduktion erzeugt werden, durch eine Ordnung eingeschränkt. Diese Ordnung bestimmt, ob ein Wort "kleiner" als ein anderes Wort ist, wobei das nicht für die tatsächliche Länge der Wörter zutreffen muss. Wenn nun für eine Sprache eine Selbstreduktion durchgeführt wird, sind alle Wörter, die während dieser Selbstreduktion erzeugt werden, "kleiner" als das Eingabewort der Reduktion. Eine mögliche Vorstellung davon ist, dass man ein Problem in potenziel mehrere kleinere Teilprobleme aufspaltet.

Man kann verschiedene Ordnungen für eine Selbstreduktion verwenden, es gibt aber auch grundlegende Eigenschaften, die diese Ordnungen haben müssen um für eine Selbstreduktion verwendet werden zu können. Diese werden in der folgenden Definition angegeben.

Definition 8 ([4])**.** *Gegeben sei das Alphabet Σ. Seien $x, y \in \Sigma^*$ beliebige Wörter. Dann ist eine Ordnung $\prec$* polynomially related*, wenn ein Polynom p existiert und folgende Eigenschaften gelten.*

1. $\{z \in \Sigma^* \mid z \prec y\} \in P$
2. $x \prec y \implies |x| \leq p(|y|)$
3. *Die Länge jeder beliebigen $\prec$-Kette ist kleiner als $p(n)$, wobei n die Länge des "längsten" Worts in der Kette ist.*

Mit der ersten Eigenschaft ist sichergestellt, dass sich zwei Wörter mit dieser Ordnung in polynomieller Zeit vergleichen lassen. Die zweite Eigenschaft sorgt dafür, dass die echte Länge vom "kleineren" Wort x höchstens polynomiel in Abhängigkeit der Länge vom "größeren" Wort y sein kann, wenn $x \prec y$ gilt. Zuletzt verhindert die dritte Eigenschaft Endlosschleifen, wenn Selbstreduktion rekursiv angewendet wird.

Jede Selbstreduktion, die in dieser Arbeit verwendet wird, benutzt diese Art von Ordnung. Darüberhinaus wird die Laufzeit der Selbstreduktionen mit P beschränkt, sie

sind also in polynomieller Zeit berechenbar. In der Literatur gibt es noch weitere Definitionen, zum Beispiel kann statt dem Zeitbedarf auch der Speicherbedarf beschränkt werden, so zum Beispiel mit logarithmischen Speicherbedarf. [2]

Es gibt aber auch die Möglichkeit andere Ordnungen zu verwenden, die nicht alle drei hier genannten Eigenschaften erfüllen. So existiert zum Beispiel word-decreasing Ordnungen, bei denen ein Wort "kleiner" als ein anderes Wort ist, wenn das erste echt kürzer oder bei gleicher Länge lexikographisch kleiner ist. Hierbei kann es für ein Eingabewort exponentiel viele Wörter geben, die "kleiner" sind, womit die dritte Eigenschaft von Definition 8 nicht erfüllt wird. [2]

Ein Beispiel für eine Ordnung, die die drei Eigenschaften von Definition 8 erfüllt und sogar die zweite Eigenschaft weiter einschränkt, sind die length-decreasing Ordnungen. Bei diesen gilt zusätzlich, dass $x \prec y \implies |x| < |y|$. Wird diese Ordnung verwendet, sind "kleinere" Wörter echt kürzer als die "größeren" Wörter. Dies erfüllt offensichtlich die zweite Eigenschaft, aber auch die dritte Eigenschaft wird erfüllt. Da für jedes Wort einer $\prec$-Kette das nächste "kleinere" Wort echt kürzer sein muss, können nur linear viele Wörter in dieser $\prec$-Kette sein. [2]

3.2 Arten von Selbstreduktion

Wie auch schon bei Reduktion gibt es verschiedene Arten von Selbstreduktionen, die man verwenden kann. Da es sich bei Selbstreduktion in erster Linie um eine Erweiterung von Reduktion handelt, kann man Selbstreduktionen über diese definieren.

Wie der Name schon sagt, ist die Selbstreduktion einer Sprache eine Reduktion auf die Sprache selbst. Allerdings darf dabei nur auf Wörter reduziert werden, die laut der verwendeten polynomially related Ordnung $\prec$ "kleiner" sind, als das zu reduzierende Wort.

In dieser Arbeit soll Selbstreduktion mithilfe von zwei Reduktionen definiert werden, die in dieser Arbeit vorgestellt wurden. Dies sind die Turing-Reduktion und die Truth-Table-Reduktion. Die jeweiligen Selbstreduktionen sind wie folgt definiert.

Definition 9 ([4])**.** *Gegeben sei das Alphabet Σ. Eine Sprache $A \subseteq \Sigma^*$ ist* Turing-selbstreduzierbar*, kurz* T-selbstreduzierbar*, wenn eine polynomially related Ordnung $\prec$ existiert, sodass $A \leq_T^P A$ gilt, wobei für alle Eingabewörter $x \in \Sigma^*$ und Wörter $y \in \Sigma^*$, die während der Berechnung auf das Orakelband geschrieben werden, $y \prec x$ gilt.*

Definition 10 ([4])**.** *Gegeben sei das Alphabet Σ. Eine Sprache $A \subseteq \Sigma^*$ ist* Truth-Table-selbstreduzierbar*, kurz* tt-selbstreduzierbar*, wenn eine polynomially related Ordnung $\prec$ existiert, sodass $A \leq_{tt}^P A$ gilt, wobei für alle Eingabewörter $x \in \Sigma^*$ gilt, dass $y_i \prec x, \forall i \in [1, k]$.*

Diese Selbstreduktionen kann man dann weiter einschränken und so weitere Selbstreduktionen definieren. In dieser Arbeit wird dafür die tt-Selbstreduktion verwendet. Bei einer tt-Selbstreduktion wird eine Boolsche Funktion α benutzt. Wenn man die Arten von Boolschen Funktionen, die α sein kann, beschränkt, so bekommt man weitere Selbstreduktionen. Zwei für diese Arbeit relevanten Arten von Boolschen Funktionen

sind die disjunktiven und konjunktiven Funktionen. Mit diesen sollen nun die disjunktive Selbstreduktion und die konjunktive Selbstreduktion definiert werden:

Definition 11 ([4]). *Gegeben sei das Alphabet Σ. Eine Sprache $A \subseteq \Sigma^*$ ist* disjunktiv-selbstreduzierbar*, kurz* d-selbstreduzierbar*, wenn A tt-selbstreduzierbar ist und dabei die Boolsche Funktion α, die durch die Reduktionsfunktion $f(x)$ erzeugt wird, disjunktiv ist, also $\alpha(y_1, y_2, ..., y_k) = 0 \iff y_1 = y_2 = ... = y_k = 0$.*

Definition 12 ([4]). *Gegeben sei das Alphabet Σ. Eine Sprache $A \subseteq \Sigma^*$ ist* konjunktiv-selbstreduzierbar*, kurz* k-selbstreduzierbar*, wenn A tt-selbstreduzierbar ist und dabei die Boolsche Funktion α, die durch die Reduktionsfunktion $f(x)$ erzeugt wird, konjuktiv ist, also $\alpha(y_1, y_2, ..., y_k) = 1 \iff y_1 = y_2 = ... = y_k = 1$.*

3.3 Hierarchie selbstreduzierbarer Sprachen

Mithilfe dieser Selbstreduktionen kann man wieder verschiedene Klassen von Sprachen haben, die auf bestimmte Art selbstreduzierbar sind. Ähnlich wie bei den Komplexitätsklassen lässt sich auch hier mit den in dieser Arbeit definierten Selbstreduktionen eine Hierarchie von Klassen zeigen. Dafür soll zunächst der folgende Satz bewiesen werden:

Satz 1 ([4]). *Gegeben sei ein Alphabet Σ und eine Sprache $L \subseteq \Sigma^*$.* Wenn L tt-selbstreudzierbar ist, dann ist L auch T-selbstreduzierbar.

Beweis. Um diesen zu beweisen soll gezeigt werden, dass für jede tt-selbstreduzierbare Sprache L eine Orakelmaschine M existiert, die diese in polynomieller Zeit entscheiden kann und dabei nur Wörter auf das Orakelband schreibt, die "kleiner" als das Eingabewort ist.

Sei L eine beliebige tt-selbstreduzierbare Sprache. Sei f die Reduktionsfunktion von L, die laut Definition 10 existiert. Dann lässt sich die Orakelmaschine M_{tt}^L konstruieren, die durch Algorithmus 1 beschrieben wird.

Zunächst soll gezeigt werden, dass M_{tt}^L einen polynomiellen Zeitbedarf hat. Da f in polynomieller Zeit berechenbar ist, benötigt M_{tt}^L auch nur polynomielle Zeit um diese Funktion zu berechnen. Darüberhinaus gilt für die Ausgabe von f, dass diese eine polynomielle Länge hat, da sonst f nicht in polynomieller Zeit berechenbar wäre.

Die Schleife wird maximal k Mal durchlaufen, wobei k die Anzahl der Wörter ist, die durch f erzeugt wurde. Da diese Anzahl polynomiell beschränkt ist, wird die Schleife nur polynomiell oft durchlaufen.

In der Schleife wird eines der Wörter $y_1, ..., y_k$ auf das Orakelband geschrieben. Da auch jedes Einzelne dieser Wörter maximal eine polynomielle Länge hat, benötigt dieser Schritt ebenfalls nur polynomielle Zeit. Das Orakelband zu leeren ist ebenfalls in polynomieller Zeit möglich, da nur einzelne Wörter von $y_1, ..., y_k$ während der Berechnung auf diesem stehen.

In einen anderen Zustand zu wechseln benötigt nur einen Berechnungschritt. Aufgrund der Eigenschaften von Orakelmaschinen, wird M_{tt}^L direkt im nächsten Schritt entweder im Zustand YES oder im Zustand NO sein. Dies zu überprüfen ist offensichtlich in konstanter Zeit möglich. Danach eine einzelne 1 beziehungsweise 0 an das Wort auf

Algorithm 1: Orakelmaschine M_{tt}^L

Input: x

Simuliere f mit Eingabe x und schreibe die Wörter $y_1, ..., y_k$ auf Arbeitsband 1 und die Boolsche Funktion α auf Arbeitsband 2;
foreach $y_i \in y_1, ..., y_k$ **do**
    Leere das Orakelband;
    Schreibe y_i auf das Orakelband;
    Gehe in Zustand QUERY;
    if *Maschine ist in Zustand YES* **then**
        Füge eine 1 auf Arbeitsband 3 hinzu;
    end
    if *Maschine ist in Zustand NO* **then**
        Füge eine 0 auf Arbeitsband 3 hinzu;
    end
end
Simuliere α auf Arbeitsband 3;
Schreibe das Ergebnis auf das Ausgabeband;

Arbeitsband 3 anzufügen benötigt auch nur konstante Zeit, da sich die Orakelmaschine nach jedem Anfügen eines Symbols die nächste freie Bandzelle merken kann.

Es ist bekannt, dass man Boolsche Funktionen in polynomieller Zeit berechnen kann, dementsprechend benötigt auch die Simulation von α nur polynomielle Zeit. Zuletzt wird das Ergebnis dieser Simulation auf das Ausgabeband geschrieben. Da dieses Ergebnis entwedet eine 1 oder eine 0 ist, ist dies offensichtlich in konstanter Zeit möglich.

Insgesamt benötigen alle Schritte der Orakelmaschine M_{tt}^L maximal polynomieller Zeit und die Schleife wird auch nur polynomiell oft ausgeführt. Damit hat M_{tt}^L einen polynomiellen Zeitbedarf.

Nun muss noch gezeigt werden, dass M_{tt}^L die Sprache L korrekt entscheidet. Für jedes Wort y_i wird das Orakel L befragt. Immer, wenn die Orakelmaschine dann in Zustand YES wechselt, wird als das i-te Symbol auf Arbeitsband 3 eine 1 geschrieben. Wenn sie stattdessen in den Zustand NO wechselt, wird eine 0 an diese Stelle geschrieben. Das bedeutet, dass das i-te Symbol auf Arbeitsband 3 eine 1 ist, wenn $y_i \in L$, und eine 0 ist, wenn $y_i \notin L$. Das entspricht dem Ergebnis der charakteristischen Funktion C_L.

Das i-te Symbol auf Arbeitsband 3 entspricht also dem Ergebnis der charakteristischen Funktion C_L mit dem Wort y_i als Eingabe. Dann wird die Boolsche Funktion α mit diesen Werten als Eingabe simuliert. Die Ausgabe dieser Funktion ist dann also 1, wenn für das Eingabewort x der Orakelmaschine $x \in L$ gilt. Wenn $x \notin L$ gilt, ist die Ausgabe wiederum 0.

Da das Ergebnis von α direkt auf das Ausgabeband geschrieben wird, ist auch die Ausgabe von M_{tt}^L 1, wenn $x \in L$ gilt, und 0, wenn $x \notin L$ gilt. Damit entscheidet M_{tt}^L die Sprache L.

Als letztes muss noch gezeigt werden, dass alle Wörter, die von M_{tt}^L auf das Orakel-

band geschrieben werden, "kleiner" als das Eingabewort x laut der Ordnung $\prec$ ist. $\prec$ ist hierbei die polynomially related Ordnung, über die die Sprache L tt-selbstreduzierbar ist.

Während der Berechnung werden nur Wörter auf das Orakelband geschrieben, die durch die Reduktionsfunktion f mit Eingabe x erzeugt werden. Laut Definition 10 gilt $y_i \prec x$ für alle Wörter y, die durch f erzeugt werden. Alle Wörter, die auf das Orakelband geschrieben werden, sind also "kleiner" als das Eingabewort.

Zusammenfassend bedeutet das, dass M_{tt}^L eine Orakelmaschine mit polynomiellen Zeitbedarf ist, welche die Sprache L entscheidet und nur Wörter auf das Orakelband schreibt, die "kleiner" als das Eingabewort x sind. Damit ist L auch T-selbstreduzierbar. □

Da jede Sprache L, die d-selbstreduzierbar oder k-selbstreduzierbar ist, laut den Definitionen 11 und 12 auch tt-selbstreduzierbar sind, lassen sich auch für solche Sprachen die Orakelmachine M_{tt}^L konstruieren, womit folgende zwei Aussagen gelten.

Korollar 1. *Gegeben sei ein Alphabet Σ und eine Sprache $L \subseteq \Sigma^*$.* Wenn L d-selbstreduzierbar ist, dann ist L auch T-selbstreduzierbar.

Korollar 2. *Gegeben sei ein Alphabet Σ und eine Sprache $L \subseteq \Sigma^*$.* Wenn L k-selbstreduzierbar ist, dann ist L auch T-selbstreduzierbar.

Daraus folgt schlussendlich die folgende Hierarchie. L ist d-selbstreduzierbar oder k-selbstreduzierbar $\implies$ L ist tt-selbstreduzierbar $\implies$ L ist T-selbstreduzierbar.

Zwischen d-Selbstreduzierbarkeit und k-Selbstreduzierbarkeit besteht außerdem eine komplementäre Beziehung. Es gilt der folgende Satz.

Satz 2. *Gegeben sei eine Alphabet Σ und eine Sprache $L \subseteq \Sigma^*$. L* ist d-selbstreduzierbar, genau dann wenn $\overline{L}$ k-selbsreduzierbar ist.

Beweis. Um diesen Satz zu beweisen sollen beide Implikationen gezeigt werden. Zunächst soll gezeigt werden, dass das Komplement jeder beliebigen d-selbstreduzierbaren Sprache k-selbstreduzierbar ist.

Gegeben sei eine beliebige d-selbstreduzierbare Sprache $L \subseteq \Sigma^*$ mit dem Alphabet Σ. Laut Definition 11 existiert somit eine Reduktionsfunktion f_d für L. In diesem Beweis soll gezeigt werden, dass sich aus diesem f_d eine Reduktionsfunktion f_k für $\overline{L}$ konstruieren lässt, die konjunktiv ist.

Sei nun α die Boolsche Funktion, die von f_d mit Eingabe x ausgegeben wird, wobei $x \in \Sigma^*$ gilt. Diese Funktion ist l-stellig, wobei l der Anzahl an Wörter entspricht, die bei der selben Eingabe von f_d ausgegeben werden. Wenn nun $l = 0$ ist, so wird α mit einer leeren Eingabe berechnet und entspricht dabei einer konstanten 1, wenn $x \in L$ gilt. Ansonsten entspricht sie einer konstanten 0. Konstruiert man anstatt α das Komplement dieser Funktion, also $\overline{\alpha}$, so gilt genau das Umgekehrte. $\overline{\alpha}$ entspricht somit einer konstanten 1, wenn $x \notin L$ beziehungsweise $x \in \overline{L}$ gilt.

Da sowohl α als auch $\overline{\alpha}$ 0-stellige Boolsche Funktionen sind, sind beide sowohl disjunktiv als auch konjunktiv. Somit erfüllt $\overline{\alpha}$ für den Fall $l = 0$ bereits die Eigenschaften von k-Selbstreduzierbarkeit.

Nun soll $l > 0$ gelten. Die Boolsche Funktion α hat somit mindestens eine Variable. Da α disjunktiv ist, lässt es sich auch mit der aussagenlogischen Formel $\bigvee_{i=1}^{l} p_i$ beschreiben. Hierbei sind p_i die Variablen von α, wobei p_i der i-ten Eingabe von α entspricht. Bildet man nun von α das Komplement $\overline{\alpha}$, so lässt sich dieses mit der aussagenlogischen Formel $\bigwedge_{i=1}^{l} q_i$ beschreiben, wobei $q_i = \neg p_i$ gilt.

p_i ist dann 1, wenn das Wort y_i zur Sprache L gehört, und ist sonst 0. Umgekehrt gilt für q_i das Komplement, also ist q_i dann 1, wenn y_i nicht zur Sprache L gehört, und ist sonst 0. Beziehungsweise ist q_i dann 1, wenn y_i zur Sprache $\overline{L}$ gehört, und ist 0 sonst.

$\overline{\alpha}$ gibt somit 1 aus, wenn alle Variablen q_i auf 1 gesetzt werden. Dies ist dann der Fall, wenn alle Wörter y_i zur Sprache $\overline{L}$ gehören. Ansonsten ist die Ausgabe 0. Dies entspricht der Vorgabe aus Definition 12. Somit erfüllt $\overline{\alpha}$ auch für den Fall $l > 0$ die Eigenschaften von k-Selbstreduzierbarkeit.

Die Reduktionsfunktion f_k soll die gleichen Wörter $y_1, ..., y_l$ wie f_d ausgeben und als Boolsche Funktion $\overline{\alpha}$, wobei α die Boolsche Funktion ist, die bei gleicher Eingabe von f_d ausgegeben wird. Damit ist die Boolsche Funktion definitiv konjunktiv.

Man muss nur noch sicherstellen, dass f_k auch in polynomieller Zeit berechenbar ist. Es erzeugt die gleichen Boolschen Funktion α wie f_d mit einer Änderung. Diese Änderung ist, dass das Ergebnis einmal invertiert wird, eine 1 wird zur 0 und eine 0 wird zur 1. Damit ist das Komplement einer Boolschen Funktion konstruiert. Dies ist offensichtlich in polynomieller Zeit möglich.

Somit gilt nun also, dass die Sprache $\overline{L}$ k-selbstreduzierbar mit der Reduktionsfunktion f_k ist.

Damit wurde die erste Implikation bewiesen. Als nächstes soll gezeigt werden, dass das Komplement jeder beliebigen k-selbstreduzierbaren Sprache d-selbstreduzierbar ist.

Es soll eine beliebige k-selbstreduzierbare Sprache $L \subseteq \Sigma^*$ betrachtet werden, wobei Σ ein beliebiges Alphabet ist. Laut Definition 12 exisitert für L eine Reduktionsfunktion f_k. Sei $\prec$ die polynomially related Ordnung, über die L selbstreduzierbar ist.

Seien $y_1, ..., y_l$ die Wörter, die f_k ausgibt, wenn es das beliebig gewählte Wort $x \in \Sigma^*$ als Eingabe erhält. Im gleichen Zug wird auch die Boolsche Funktion α ausgegeben, welche konjunktiv ist. α ist dann l-stellig, sie besitzt also l verschiedene Variablen. Damit entspricht sie einer aussagenlogischen Formel der Form $\bigwedge_{i=1}^{l} p_i$. Hierbei erhält die Variable p_i den Wert 1, wenn für das entsprechende Wort $y_i \in L$ gilt. Gilt wiederum $y_i \notin L$, so erhält p_i den Wert 0.

Bildet man von dieser aussagenlogischen Formel das Komplement, erhält man die Formel $\bigvee_{i=1}^{l} q_i$, wobei q_i neue Variablen sind, für welche $q_i = \overline{p_i}$ gilt. Die Werte, die diese Variablen erhalten, sind somit genau umgekehrt. Das heißt, eine Variable q_i hat den Wert 1, wenn $y_i \notin L$ beziehungsweise $y_i \in \overline{L}$ gilt. Entsprechend hat q_i den Wert 0, wenn $y_i \notin \overline{L}$ gilt.

Diese aussagenlogische Formel ist genau dann 0, wenn alle Variablen 0 sind. Dies ist dann der Fall, wenn alle Wörter y_i zur Sprache L beziehungsweise nicht zur Sprache $\overline{L}$ gehören. Dies erfüllt die Eigenschaft aus Definition 11 von d-Selbstreduzierbarkeit, wenn man aus dieser aussagenlogischen Formel wieder eine Boolsche Funktion erzeugt. Diese soll $\overline{\alpha}$ sein.

$\overline{\alpha}$ lässt sich einfach erzeugen, indem man das Ergebnis von α einmal negiert, 1 wird

zu 0 und 0 wird zu 1. Dann kann man mithilfe der Reduktionsfunktion f_k für L eine Reduktionsfunktion f_d für $\overline{L}$ konstruieren. Wenn für ein beliebiges Wort $x \in \Sigma^*$ die Ausgabe von f_k die Boolsche Funktion α und die Wörter $y_1, ..., y_l$ ist, dann soll für die gleiche Eingabe die Ausgabe von f_d die gleichen Wörter $y_1, ..., y_l$ und die Boolsche Funktion $\overline{\alpha}$ sein.

Damit ist gezeigt, wenn eine Sprache L k-selbstreduzierbar ist, dann muss $\overline{L}$ d-selbstreduzierbar sein. Damit sind beide Richtungen des Satzes bewiesen. □

4 Einordnung in Komplexitätsklassen

Um eine Sprache einer Komplexitätsklasse zuzuordnen, ist eine Möglichkeit eine deterministische oder nichtdeterministische Turingmaschine anzugeben, welche diese Sprache entscheidet. Dann kann man den Zeitbedarf beziehungsweise den Speicherbedarf dieser Maschine untersuchen und hat damit eine obere Schranke für die Komplexität der untersuchten Sprache.

Eine andere Möglichkeit ist zu zeigen, dass die untersuchte Sprache selbstreduzierbar ist. Je nachdem auf welche Art eine Sprache selbstreduzierbar ist, kann daraus bereits eine obere Schranke für die Komplexität gezeigt werden. Dies soll in diesem Kapitel anhand von ausgewählten Arten von Selbstreduzierbarkeit gezeigt werden.

4.1 T-selbstreduzierbare Sprachen

Zunächst soll einmal die Turing-Selbstreduktion untersucht werden, da diese die voraussichtlich allgemeinste Selbstreduktion ist. In Kapitel 3 wurde bereits gezeigt, dass alle in dieser Arbeit verwendeten Selbstreduktionen auch mit Turing-Selbstreduktion beschrieben werden können. Egal auf welche Art genau eine Sprache selbstreduzierbar ist, sie kann nur höchstens die Komplexität haben, die folgend gezeigt wird.

Satz 3 ([4])**.** *Gegeben sei ein Alphabet* Σ *und eine Sprache* $L \subseteq \Sigma^*$*.* Wenn L T-selbstreduzierbar ist, dann gilt $L \in$ PSPACE.

Beweis. Um diese Satz zu beweisen, soll gezeigt werden, dass für jede T-selbstreduzierbare Sprache L eine Turingmaschine existiert, die diese Sprache entscheidet und dabei einen polynomiellen Speicherbedarf hat.

Sei L eine beliebige T-selbstreduzierbare Sprache mit $L \subseteq \Sigma^*$, wobei Σ ein beliebiges Alphabet ist. Dann existiert laut Definition 9 eine Orakelmaschine M_T^L, die die Sprache L entscheidet und dabei einen polynomiellen Zeitbedarf hat. Die Ordnung $\prec$ soll die polynomially related Ordnung sein, für die laut Definition 9 L selbstreduzierbar ist.

Mithilfe der Orakelmaschine M_T^L soll nun eine Turingmaschine M_T konstruiert werden, die M_T^L simuliert und L entscheidet, wobei M_T insgesamt einen polynomiellen Speicherbedarf haben soll. Daraus folgt dann, dass $L \in$ PSPACE gilt. Die Turingmaschine M_T soll durch Algorithmus 2 beschrieben werden.

Diese Turingmaschine verwendet nummerierte Arbeitsbänder, von denen es potenziel unendlich viele geben kann. Eine Turingmaschine kann aber nur eine feste endliche Anzahl an Arbeitsbändern haben. Daher muss zunächst gezeigt werden, dass man diese nummerierten Arbeitsbänder auf einer Turingmaschine mit konstant vielen Arbeitsbändern umsetzen kann.

Hierzu soll gezeigt werden, wie das mit einer Turingmaschine mit genau zwei Arbeitsbändern möglich ist. Diese zwei Arbeitsbänder sollen als A und B bezeichnet werden. Auf dem ersten Arbeitsband A soll ein Zähler geschrieben werden, der inkrementiert und dekrementiert werden kann. Dieser Zähler kann zum Beispiel in Binärdarstellung geschrieben sein.

Auf dem zweiten Arbeitsband B sollen die Inhalte der nummerierten Arbeitsbänder geschrieben werden und hintereinander stehen. Dafür werden sie mit einem speziellen Symbol voneinander getrennt, welches sonst an keiner Stelle während der Berechnung verwendet wird, nur zum Trennen der nummerierten Arbeitsbänder. Damit ist sowohl Anfang als auch Ende jedes nummerierten Arbeitsbandes durch dieses Trennsymbol markiert.

Am Anfang der Berechnung soll der Zähler auf Arbeitsband A den Wert 1 haben und Arbeitsband B soll leer sein. Auf ein Arbeitsband mit der Nummer i soll nur ein Bandinhalt sein können, wenn auch für alle j mit $j < i$ gilt, dass Arbeitsband j einen Bandinhalt hat. Die einzige Ausnahme ist das Arbeitsband 1, welches das erste nummerierte Arbeitsband sein soll.

Immer, wenn ein nummeriertes Arbeitsband links um ein Symbol erweitert wird, soll der gesamte Inhalt von B von dieser Stelle an um eine Position nach rechts verschoben werden. Wird zum Beispiel das Arbeitsband mit der Nummer i erweitert, werden alle Symbole auf Arbeitsband B, die zu Arbeitsband i oder zu den folgenden Arbeitsbändern gehören, um eine Position nach rechts verschoben.

Wird stattdessen das Arbeitsband mit der Nummer i rechts um ein Symbol erweitert, so wird das zu diesem Arbeitsband rechte Trennsymbol und alle rechts folgenden Symbole auf B um eine Position nach rechts verschoben. Solange das Wort auf einem dieser nummerierten Arbeitsbändern nicht länger wird, muss nichts verschoben werden.

Wenn nun auf dem Arbeitsband mit der Nummer i eine Berechnung durchgeführt werden soll und der Zähler auf A aktuell kleiner als i ist, so soll die Turingmaschine auf Arbeitsband B solange nach rechts laufen und den Zähler für jedes gefundene Trennsymbol inkrementieren, bis der Zähler gleich i ist. Dann wurde der Bereich von B gefunden, der den Inhalt von Arbeitsband i enthält und die Berechnung kann durchgeführt werden. Ist der Zähler wiederum größer als i, so muss die Turingmaschine sich nach links bewegen und für jedes gefundene Trennsymbol den Zähler dekrementieren. Im Fall, dass der Zähler bereits gleich i ist, muss nichts gemacht werden, da die Turingmaschine sich bereits im richtigen Bereich befindet.

Wenn auf einem neuen nummerierten Arbeitsband, welches bisher nicht verwendet wurde, ein Wort geschrieben werden soll, dann soll die Turingmaschine an das Ende von B hinter das letzte Trennsymbol laufen. Dort soll dann das Wort und rechts davon ein weiteres Trennsymbol geschrieben werden. Wenn es das erste nummerierte Arbeitsband ist, dann soll stattdessen erst ein Trennsymbol auf B geschrieben werden, dann das Wort und schließlich noch ein Trennsymbol.

Damit ist gezeigt, wie man potenziel unendlich viele nummerierte Arbeitsbänder auf einer Turingmaschine mit genau zwei Arbeitsbändern umsetzen kann. Diese kann man natürlich um weitere Arbeitsbänder mit anderen Aufgaben erweitern, was bei M_T der Fall ist.

Die Simulation einer Orakelmaschine soll so funktionieren, dass beim Starten der

Simulation ein Arbeitsband angegeben wird, welches als Eingabeband dieser Orakelmaschine fungieren soll. Danach wird dann immer ein Berechnungsschritt dieser Simulation explizit durchgeführt, nach dem Prinzip "Berechne einen Schritt der Simulation auf Arbeitsband 3".

Wenn während so einer Simulation die Orakelmaschine in den Zustand QUERY übergeht, springt die Turingmaschine ein und berechnet deterministisch beziehungsweise nichtdeterministisch die Antwort. Die Simulation wird dann entweder im Zustand YES oder im Zustand NO fortgesetzt, nach dem Prinzip "Setze Simulation auf Arbeitsband 8 in Zustand YES".

M_T verwendet einen weiteren Zähler i, der die aktuelle Anzahl an laufenden Simulationen und somit auch die aktuelle Anzahl an verwendeten nummerierten Arbeitsbändern angibt.

Als erstes soll nun gezeigt werden, dass die Turingmaschine M_T tatsächlich nur einen polynomiellen Speicherbedarf hat. Zunächst gilt für den Zähler i, dass man diesen zum Beispiel in binärer Darstellung schreiben kann. Wenn dieser nun den Wert n hat, so werden $\lceil log_2(n) \rceil$ Bandzellen benötigt. Dieser Zähler benötigt also nur logarithmischen Speicher, insbesondere also auch nur polynomiellen Speicher. Das Inkrementieren und Dekrementieren dieses Zählers ändert nichts am Speicherbedarf für diesen.

Auf jedes nummerierte Arbeitsband wird entweder das Eingabewort x oder ein Wort y geschrieben, wobei y innerhalb einer Simulation von M_T^L erzeugt wird. Da der Speicherbedarf in Abhängigkeit der Länge von x bestimmt wird, benötigt x offensichtlich nur linearen Speicher und somit ebenfalls polynomiellen Speicher. Laut Definition 9 gilt jür jedes Wort y, dass während der Berechnung beziehungsweise der Simulation von M_T^L auf das Orakelband geschrieben wird, dass $y \prec x$. Somit ist die Länge von jedem dieser Wörter y polynomiell über die Länge von x beschränkt.

Wenn eine Simulation von M_T^L auf einem nummerierten Arbeitsband gestartet wird, soll der zu dieser Zeit beschriebene Teil des Arbeitsbandes als Eingabeband für die Simulation verwendet werden. Der Speicher, der ansonsten von der Simulation benötigt wird, soll durch ein weiteres spezielles Trennsymbol hinter dem Eingabewort stehen. Damit ist sichergestellt, dass nicht durch weitere dynamische Arbeitsbänder unnötiger Speicher gebraucht wird. Der Speicherbedarf von jedem nummerierten Arbeitsband ist dann die Länge der Wörter y, die während Simulationen von M_T^L erzeugt werden, plus den Speicherbedarf, den M_T^L abhänging vom jeweiligen Eingabewort hat.

Da die Orakelmaschine M_T^L in polynomieller Zeit berechenbar ist, kann in jedem Schritt maximal eine neue Bandzelle beschrieben werden. Da es nur polynomiell viele Schritte geben kann, können insgesamt auch nur polynomiell viele Bandzellen beschrieben werden. Somit benötigt auch jede Simulation von M_T^L nur polynomiellen Speicher. Darüberhinaus kann man den von abgeschlossenen Simulationen verwendeten Speicher für neue Simulationen wiederverwenden, sodass der insgesamte Speicherbedarf für alle Simulationen zusammen nur noch polynomiell von der Anzahl der zur gleichen Zeit laufenden Simulationen sein kann.

Die erste Simulation von M_T^L startet mit x als Eingabe. Das bedeutet, für alle Wörter y, die von dieser Simulation erzeugt und auf das Orakelband geschrieben werden, gilt $y \prec x$. x steht hierbei auf Arbeitsband 1 und die Wörter y werden nach und nach auf Arbeitsband 2 geschrieben, wobei immer nur eines zur Zeit auf diesem ist. Mit dem

Algorithm 2: Turingmaschine M_T

Input: x

Setze i auf 1;
Schreibe x auf Arbeitsband 1;
Starte Simulation von Orakelmaschine M_T^L auf Arbeitsband 1;
while $i > 0$ **do**
    Berechne einen Schritt der Simulation auf Arbeitsband i;
    if *Simulation auf Arbeitsband i ist im Zustand QUERY* **then**
        Inkrementiere i;
        Überschreibe Arbeitsband i mit dem Wort y vom Orakelband der Simulation auf Arbeitsband $i-1$;
        Starte Simulation von Orakelmaschine M_T^L auf Arbeitsband i;
    end
    if *Simulation auf Arbeitsband i endet mit Ausgabe 1* **then**
        Dekrementiere i;
        if *i == 0* **then**
            break;
        end
        Setze Simulation auf Arbeitsband i in Zustand YES;
    end
    if *Simulation auf Arbeitsband i endet mit Ausgabe 0* **then**
        Dekrementiere i;
        if *i == 0* **then**
            break;
        end
        Setze Simulation auf Arbeitsband i in Zustand NO;
    end
end
if *Simulation auf Arbeitsband 1 endet mit Ausgabe 1* **then**
    Schreibe 1 auf Ausgabeband
else
    Schreibe 0 auf Ausgabeband
end

Wort, was auf Arbeitsband 2 geschrieben wird, wird dann eine weitere Simulation von M_T^L gestartet, welche wiederum Wörter z erzeugt und auf das Orakelband schreibt Für diese gilt nun $z \prec y$. Da aber auch $y \prec x$ gilt, muss auch die $\prec$-Kette $z \prec y \prec x$ gelten. Das lässt sich mit jeder weiteren Simulation fortsetzen, solange die vorherigen noch nicht beendet wurden. Das bedeutet, dass die Wörter auf den nummerierten Arbeitsbändern implizit eine $\prec$-Kette erzeugen, wobei das Wort auf Arbeitsband 1 das "größte" Wort ist und das auf Arbeitsband i das "kleinste" Wort ist, wobei i der aktuelle Zähler ist.

Laut Definition 8 gilt für die Ordnung $\prec$, die von der Orakelmaschine M_T^L verwendet wird, dass die Länge jeder beliebigen $\prec$-Kette polynomiell über die Länge des "größten" Worts beschränkt ist. Da für alle Wörter y, die während der Berechnung erzeugt werden, entweder direkt oder über andere Wörter, dass y "kleiner" als x ist, ist x das "größte" Wort in jeder dieser $\prec$-Ketten, die während der Berechnung implizit erzeugt werden. Sei für diese $\prec$-Ketten k die maximal mögliche Länge. Es können also nur $\prec$-Ketten während der Berechnung erzeugt werden, die maximal k Elemente enthalten und dieses k ist polynomiell über die Länge von x beschränkt.

Wenn eine $\prec$-Kette maximaler Länge mit der Form $y_k \prec ..., y_1$ existiert, wobei $y_1 = x$ das "größte" Wort darin ist, so kann M_T^L kein Wort auf das Orakelband schreiben, wenn es y_k als Eingabe erhält. Laut Definition 9 schreibt die Orakelmaschine M_T^L nur Wörter auf das Orakelband, die laut der Ordnung $\prec$ "kleiner" als das Eingabewort sind. Würde M_T^L also ein Wort auf das Orakelband schreiben, welches nun y_{k+1} sein soll, würde $y_{k+1} \prec y_k$ gelten. Allerdings würde dann auch $y_{k+1} \prec y_k \prec ... \prec y_1$ gelten, wobei weiterhin $y_1 = x$. Diese $\prec$-Kette hätte dann eine Länge, die größer als k wäre, was nach Vorraussetzung aber nicht möglich ist. Spätestens also, wenn eine Simulation von M_T^L mit einem Wort als Eingabe gestartet wird, welches das "kleinste" Wort in so einer $\prec$-Kette ist, wird von dieser Simulation kein Wort auf das Orakelband geschrieben. Somit wird diese Simulation nicht unterbrochen und die Entscheidung wird in polynomieller Zeit berechnet.

Es können also nur maximal k Wörter auf den nummerierten Arbeitsbändern stehen, was bedeutet, dass auch nur maximal k dieser Arbeitsbänder verwendet werden. Da diese Arbeitsbänder immer wieder überschrieben werden, ist der Speicherbedarf auf diesen Arbeitsbändern beschränkt durch die maximal Länge der Wörter, die während der Berechnung auf diese geschrieben werden. Es wurde bereits erklärt, dass diese polynomiell lang sind.

Da nur polynomiell viele nummerierte Arbeitsbänder verwendet werden können, die jeweils nur polynomiell viel Speicher benötigen, folgt, dass M_T damit einen polynomiellen Speicherbedarf hat.

Da auch k polynomiell über die Länge von x beschränkt ist, folgt daraus, dass nur polynomiell viele nummerierte Arbeitsbänder verwendet werden. Damit ist der insgesamte Speicherbedarf von M_T polynomiell.

Es wurde zwar gezeigt, dass die Turingmaschine M_T einen polynomiellen Speicherbedarf hat, doch ist noch nicht sichergestellt, dass diese auch ein Ergebnis berechnen wird oder ob sie in eine Endlosschleife übergehen wird. Dies soll an dieser Stelle gezeigt werden.

Da jede Simulation von M_T^L nur maximal polynomiell viele Wörter auf das Orakelband schreibt, können diese Simulationen auch nur polynomiell oft unterbrochen werden, bevor sie eine Entscheidung berechnet haben. Auf jede Simulation können also nur polynomiell viele Simulationen direkt folgen, auf die wiederum auch polynomiell viele folgen können. Dies kann man in Form eines gerichteten Baumes darstellen, in dem jeder Knoten einer Simulation entspricht und die Kindknoten jeweils die Simulationen sind, welche die Simulation des Elternknotens unterbechen. Die Wurzel dieses Baumes ist die erste Simulation von M_T^L, die als Eingabe x bekam.

Jeder Pfad in diesem Baum kann als eine Folge von Simulationen angesehen werden, in

der jede davon die jeweils vorherige Simulation unterbricht und die letzte Simulation die aktuell laufende ist. Es wurde bereits erklärt, dass die Wörter, die für diese Simulationen als Eingabe dienen, eine $\prec$-Kette ergeben. Darüberhinaus wurde bereits erklärt, dass diese Ketten eine maximale Länge von k haben, somit haben auch alle Pfade in diesem Baum diese maximale Länge. Speziel die Pfade von der Wurzel zu jedem beliebigen Blattknoten können maximal die Länge k haben. Damit ist die Höhe dieses Baumes polynomiell beschränkt. Da jeder Knoten im Baum nur polynomiell viele Kindknoten haben kann, folgt, dass dieser Baum maximal exponentiell viele Knoten haben kann, also auch nur exponentiell viele Simulationen während der Berechnung durchgeführt werden können. Dies bedeutet, dass M_T nicht in eine Endlosschleife geraten kann und immer ein Ergebnis berechnen wird.

Zuletzt muss noch gezeigt werden, dass M_T die Sprache L korrekt entscheidet. Zunächst kann man feststellen, dass jede Simulation von M_T^L entweder Wörter auf das eigene Orakelband schreibt oder die Entscheidung ohne einmal in den Zustand QUERY zu gehen berechnet und auf das eigene Ausgabeband schreibt. Wenn M_T^L die Eingabe entscheidet ohne ein Wort auf das Orakelband zu schreiben, so gehört die Eingabe zur Sprache L, wenn die Ausgabe 1 ist, und so gehört die Eingabe nicht zur Sprache L, wenn die Ausgabe 0 ist.

Wenn aber ein Wort während einer Simulation auf das Orakelband geschrieben wird, wird diese unterbrochen und später fortgesetzt. Es wird dann mit dem Wort, welches auf das Orakelband geschrieben wurde, eine neue Simulation gestartet. Wenn diese mit Ausgabe 1 endet, wird die ursprüngliche Simulation im Zustand YES fortgesetzt. Wenn diese mit Ausgabe 0 endet, wird die ursprüngliche Simulation im Zustand NO fortgesetzt. Dies entspricht der normalen Funktionsweise der Orakelmaschine M_T^L, da diese im nächsten Schritt in den Zustand YES wechselt, wenn das Wort auf dem Orakelband zur Sprache L gehört. Dies ist dann der Fall, wenn M_T^L mit diesem Wort als Eingabe die Ausgabe 1 berechnet. Analog dazu gilt das gleiche mit dem Zustand NO und der Ausgabe 0.

Das bedeutet, auch wenn eine Simulation unterbrochen wird, wird sie zu einem späteren Zeitpunkt im korrekten Zustand fortgesetzt, sodass das berechnete Ergebnis dann auch dem Ergebnis der reinen Orakelmaschine entspricht. Solange eine Simulation von M_T^L also beendet wird, wird durch diese Simulation für die Eingabe korrekt entschieden, ob diese zur Sprache L gehört.

Die Turingmaschine M_T schreibt genau dann eine 1 auf das Ausgabeband, wenn die Simulation von M_T^L auf Arbeitsband 1 mit Ausgabe 1 endet. Das bedeutet, die Entscheidung von M_T^L mit Eingabe x wird direkt als Ausgabe verwendet. Da jede Simulation von M_T^L die Sprache L entscheidet, gilt, dass auch M_T die Sprache L entscheidet.

Zusammenfassend wurde gezeigt, dass die Turingmaschine M_T einen polynomiellen Speicherbedarf hat, immer ein Ergebnis berechnen wird und die Sprache L korrekt entscheidet. Damit folgt schlussendlich die Aussage $L \in \text{PSPACE}$. □

Aufgrund von Satz 1 kann leicht die folgende Aussage bewiesen werden.

Korollar 3. *Gegeben sei ein Alphabet Σ und eine Sprache $L \subseteq \Sigma^*$. Wenn L tt-selbstreduzierbar ist, dann gilt, dass $L \in$ PSPACE.*

Selbstverständlich lässt sich mit den Korollaren 1 und 2 das gleiche auch über d-selbstreduzierbare und k-selbstreduzierbare Sprachen zeigen, dass auch diese Sprachen in PSPACE sind. Tatsächlich lassen sich aber noch andere obere Schranken bezüglich ihrer Komplexität finden, wie im weiteren Verlauf dieses Kapitels gezeigt wird.

4.2 d-selbstreduzierbare Sprachen

Bei T-Selbstreduzierbarkeit handelt es sich um die vermutlich allgemeinste Art der Selbstreduktion. Daher gilt vorraussichtlich vür alle selbstreduzierbaren Sprachen, dass sie in PSPACE sind. Bei den anderen Arten von Selbstreduktion in dieser Arbeit lassen sich aber noch genauere obere Schranken bezüglich ihrer Komplexität finden. Zunächst soll hierzu für d-selbstreduzierbare Sprachen gezeigt werden, dass sie in der Komplexitätsklasse NP sind, was durch den folgenden Satz formuliert wird.

Satz 4 ([4])**.** *Gegeben sei ein Alphabet* Σ *und eine Sprache* $L \in \Sigma^*$. Wenn L d-selbstreduzierbar ist, dann gilt, dass $L \in \mathrm{NP}$.

Beweis. Um diesen Satz zu beweisen, soll wieder gezeigt werden, dass sich für jede d-selbstreduzierbare Sprache L eine Turingmaschine konstruieren lässt, die L hier wiederum nichtdeterministisch in polynomieller Zeit entscheidet.

Sei L eine beliebige d-selbstreduzierbare Sprache. Laut Definition 11 existiert dann eine Reduktionsfunktion f, die in polynomieller Zeit berechenbar ist. Das bedeutet, dass sie deterministisch von einer Turingmaschine simuliert werden kann und diese Simulation nur einen polynomiellen Zeitbedarf aufweist. Darüberhinaus soll $\prec$ die polynomially related Ordnung sein, über die L selbstreduzierbar ist.

Es soll eine nichtdeterministische Turingmaschine M_d definiert werden, die unter Verwendung von f die Sprache L entscheidet und dabei nur einen polynomiellen Zeitbedarf hat. Diese Turingmaschine M_d soll durch den Algorithmus 3 beschrieben werden.

Zunächst soll gezeigt werden, dass die Turingmaschine M_d insgesamt einen polynomiellen Zeitbedarf hat. Wie bereits geschrieben, ist die Reduktionsfunktion f in polynomieller Zeit berechenbar. Somit ist auch jeder Schritt von M_d, welcher diese Funktion berechnet, in polynomieller Zeit möglich.

Da f in polynomieller Zeit berechenbar ist, kann das Ergebnis dieser Funktion nur polynomiellen Speicher benötigen, da ansonsten die Berechnung eine längere Zeit bräuchte. Daraus folgt, dass zum einen die Boolsche Funktion α als auch die Wörter $y_1, ..., y_k$ eine polynomielle Länge haben müssen. Damit ist auch das Schreiben dieser auf Arbeitsbänder in polynomieller Zeit möglich.

Darüberhinaus gilt, dass nur polynomiell viele Wörter $y_1, ..., y_k$ durch f erzeugt werden können, da sie zusammen bereits eine polynomielle Länge haben. Das k ist also auch polynomiell beschränkt und selbstverständlich haben die einzelnen Wörter dann auch eine polynomielle Länge.

Zu überprüfen ob ein Band leer ist, egal ob es sich dabei um ein Ausgabeband, Eingabeband oder Arbeitsband handelt, ist in konstanter Zeit möglich. Ein Arbeitsband zu leeren wiederum benötigt eine lineare Zeit in Abhängigkeit der Länge des Wortes

Algorithm 3: Turingmaschine M_d

Input: x

Simuliere f mit Eingabe x und schreibe die Wörter $y_1, ..., y_k$ auf Arbeitsband 1 und die Boolsche Funktion α auf Arbeitsband 2;
while *Ausgabeband ist leer* **do**
  if *Arbeitsband 1 ist leer* **then**
    Simuliere α mit leerer Eingabe und schreibe das Ergebnis auf das Ausgabeband;
    Halte an;
  end
  Leere Arbeitsband 2;
  Wähle nichtdeterministisch ein Wort y_i mit $i \in [1, k]$ von Arbeitsband 1;
  Schreibe y_i auf Arbeitsband 3;
  Leere Arbeitsband 1;
  Simuliere f mit Eingabe y_i und schreibe die Wörter $y_1, ..., y_k$ auf Arbeitsband 1 und schreibe α auf Arbeitsband 2;
  Leere Arbeitsband 3;
end

auf dem Arbeitsband. Auf Arbeitsband 1 werden während der Berechnung nur die Wörter $y_1, ..., y_k$ geschrieben, auf Arbeitsband 2 werden nur Boolsche Funktionen α geschrieben. Beide werden durch die Reduktionsfunktion f erzeugt und haben somit eine polynomielle Länge. Sie zu löschen ist damit auch in polynomieller Zeit möglich. Da auf Arbeitsband 3 nur einzelne Wörter von $y_1, ..., y_k$ geschrieben werden, ist auch der Inhalts dieses Bands polynomiell beschränkt und das Löschen auf Arbeitsband 3 ist folglich auch in polynomieller Zeit möglich.

Von den Wörtern $y_1, ..., y_k$ auf Arbeitsband 1 eines nichtdeterministisch zu wählen benötigt auch nur polynomielle Zeit. Man könnte nacheinander über diese Wörter laufen und bei jedem Wort nichtdeterministisch entscheiden, ob man dieses auswählt. Sobald eins ausgewählt ist, kann der nächste Schritt des Algorithmus durchgeführt werden. Im Schlimmsten Fall müsste man dann nur alle Wörter durchgehen, da dies aber polynomiell viele mit polynomieller Länge sind, ist dies in polynomieller Zeit möglich.

Der letzte Schritt der fehlt ist das Simulieren der Boolschen Funktion α. Es ist bekannt, dass sich eine Boolsche Funktion in polynomieller Zeit berechnen lässt in Abhängigkeit der Länge der Eingabe. Solange die Eingabe also polynomiell beschränkt ist, ist auch das Simulieren dieser Funktion in polynomieller Zeit möglich. In diesem Algorithmus wird α immer nur mit leerer Eingabe simuliert. Da α keine Eingabe erhält, kann es nur direkt eine 1 oder eine 0 ausgeben, womit der Zeitbedarf für diese Simulation sogar konstant ist. Das Ergebnis dieser Funktion auf das Ausgabeband zu schreiben ist offensichtlich auch in konstanter Zeit möglich.

Damit sind alle Schritte des Algorithmus in maximal polynomieller Zeit möglich. Es fehlt zu zeigen, dass die Schleife nur polynomiell oft durchlaufen wird.

Wenn die Ausgabe der Reduktionsfunktion f mit dem Eingabewort x als Eingabe keine Wörter $y_1, ..., y_k$ enthält, also $k = 0$ gilt, dann ist Arbeitsband 1 leer und es wird direkt das Ergebnis mit α berechnet. Dies trifft auch für jede andere Eingabe zu, für die f keine Wörter $y_1, ..., y_k$ sondern nur eine Boolsche Funktion α ausgibt. Die Schleife wird also nur solange durchgeführt, wie f eine Ausgabe mit $k > 0$ berechnet.

Laut Definition 10 gilt für alle Wörter y_i von $y_1, ..., y_k$, dass $y_i \prec x$ gilt, wenn x als Eingabe für f verwendet wird. Wenn nun für ein beliebiges Wort y_i wieder die Ausgabe von f berechnet wird, soll die Ausgabe $(\alpha, y_{i1}, ..., y_{il})$ mit $l > 0$ sein. Wieder gilt, wäre $l = 0$, dann würde das Endergebnis berechnet werden und die Berechnung wird beendet. Für alle diese Wörter y_{ij} gilt nun $y_{ij} \prec y_i$. Darüberhinaus gilt aber auch $y_{ij} \prec y_i \prec x$. Dies gilt für alle $1 \leq i \leq k$ und $1 \leq j \leq l$.

Dies lässt sich so weiter fortsetzen und es entsteht ein Baum, in dem jeder Knoten dem Eingabewort x oder einem Wort y_i entspricht, welches durch eine Berechnung von f erzeugt wurde. In diesem Baum gilt für alle Kindknoten, dass sie laut der Ordnung $\prec$ "kleiner" als der Elternknoten sind. Zuletzt ist der Wurzelknoten das Eingabewort x.

Jeder Pfad in diesem Baum entspricht nun einer $\prec$-Kette und jeder diese Pfade kann durch die Turingmaschine M_d durchlaufen werden, je nachdem welche Wörter nichtdeterministisch ausgewählt werden. Laut Definition 8 gilt nun allerdings, dass die Länge dieser $\prec$-Ketten polynomiell über die Länge des "größten" Worts beschränkt ist. Da das Eingabewort x der Wurzelknoten des Baums ist, ist x auch das "größte" Wort in den längsten Pfaden. Somit ist die Länge dieser Pfade polynomiell über die Länge von x beschränkt. Sei diese Länge n.

Sei nun $y_{n-1} \prec ... \prec y_1 \prec x$ eine beliebige $\prec$-Kette maximaler Länge. Nun gilt, dass keine Wörter y_i mit $y_i \prec y_{n-1}$ existieren. Denn wäre das der Fall, so könnte man die $\prec$-Kette maximaler Länge um ein weiteres Element erweitern und hätte eine $\prec$-Kette, deren Länge größer als die maximale Länge wäre.

Jede nichtdeterministische Entscheidung folgt implizit einem der Pfade in diesem Baum. In jedem Schleifendurchlauf wird also ein Wort nichtdeterministisch ausgewählt und die Ausgabe der Reduktionsfunktion f mit diesem Wort als Eingabe berechnet. Dies wird solange fortgesetzt bis das letzte Wort eines Pfades erreicht wird. Denn da keine Wörter existieren, die "kleiner" als dieses sind, muss die Ausgabe von f nur eine Boolsche Funktion α enthalten. Arbeitsband 1 ist im nächsten Schleifendurchlauf somit leer und die Berechnung wird beendet.

Die Schleife kann also nur polynomiell oft durchlaufen werden, nämlich maximal so oft wie die maximale Länge der $\prec$-Ketten. Damit ist der gesamte Zeitbedarf von M_d polynomiell.

Nun muss noch gezeigt werden, dass die Turingmaschine M_d auch tatsächlich die Sprache L entscheidet. Es gibt zwei mögliche Fälle, zum einen kann die Ausgabe der Reduktionsfunktion f keine Wörter $y_1, ..., y_k$ enthalten, die Ausgabe ist also nur eine einzelne Boolsche Funktion α. Im zweiten Fall gilt $k > 0$, es wird also mindestens ein Wort zusätzlich zu α ausgegeben.

Zunächst gilt für den ersten Fall, dass α eine 0-stellige Boolsche Funktion ist. Diese gibt bei leerer Eingabe entweder eine 1 oder eine 0, abhänig davon ob x zur Sprache L gehört oder nicht. Dies wird auch direkt von M_d als Ausgabe berechnet, womit diese Turingmaschine für diesen Fall L korrekt entscheidet.

Für den zweiten Fall soll nun $f(x) = (\alpha, y_1, ..., y_k)$ gelten. $x \notin L$ gilt genau dann, wenn für alle Wörter $y_i \notin L$ gilt. Das heißt, sobald eines dieser Wörter zur Sprache L gehört, gilt auch $x \in L$.

Die Turingmaschine M_d wählt nichtdeterministisch eines dieser Wörter aus. Es soll der Fall betrachtet werden, in dem $y_i \in L$ für das ausgewählte Wort y_i gilt. Mindestens ein Wort, nämlich y_i, ist somit in der Sprache L. Dann wird aus diesem Wort y_i mit f wieder neue Wörter erzeugt, von dem mindestens eines zur Sprache L gehören muss, da bereits y_i zu L gehört. Dies wird solange wiederholt, bis in der Ausgabe von f nur die Boolsche Funktion α ist. Bei jeder Auswahl soll ein Wort ausgewählt werden, welches zur Sprache L gehört.

Für dieses Wort wird dann das Ergebnis von α berechnet und als Ausgabe verwendet. Da die Ausgabe hiervon 1 ist, gilt zum einen, dass dieses Wort zur Sprache L gehört. Rückwirkend gilt dies auch für das vorherige Wort und dem Wort davor. Dies kann dann bis zum Eingabewort x weitergeführt werden, für das nun auch $x \in L$ gilt. Somit existiert mindestens ein Pfad im Berechnungsbaum von M_d, welcher die Eingabe akzeptiert, womit x von M_d akzeptiert wird.

Gilt nun allerdings, dass keines der Wörter, die durch f mit Eingabe x ausgegeben werden, in der Sprache L ist, so existiert auch kein Berechnungspfad, der die Eingabe akzpetiert. Gäbe es so einen Pfad, gäbe es eine Folge von Wörtern, die alle zur Sprache L gehören, wobei eines davon in der Ausgabe von f mit Eingabe x ist. Da diese aber alle nicht zur Sprache L gehören, wäre dies ein Widerspruch, womit so ein akzeptierender Pfad nicht existieren kann.

Folglich wird in diesem Fall das Eingabewort x in jedem Berechnungspfad abgelehnt, womit M_d insgesamt x ablehnt. Da die Entscheidung von M_d in beiden Fällen korrekt ist, wird die Sprache L von M_d korrekt entschieden.

Zusammenfassend bedeutet dies, dass M_d eine nichtdeterministische Turingmaschine mit polynomiellen Zeitbedarf ist, welche die Sprache L entscheidet. Damit gilt $L \in$ NP. □

4.3 k-selbstreduzierbare Sprachen

Wie zu sehen ist, ist eine Sprache L, die d-selbstreduzierbar, nicht nur mit einem polynomiellen Speicherbedarf sondern sogar mit nur einem polynomiellen Zeitbedarf entschieden werden kann, solange eine nichtdeterministische Turingmaschine verwendet wird. Für k-Selbstreduzierbarkeit wiederum gilt die folgende Aussage.

Satz 5 ([4])**.** *Gegeben sei ein Alphabet* Σ *und eine Sprache* $L \in \Sigma^*$. Wenn L k-selbstreduzierbar ist, dann gilt, dass $L \in$ coNP.

Beweis. Dieser Satz lässt sich einfach beweisen. Hierfür lassen sich in dieser Arbeit bereits gezeigte Sätze nutzen. Sei L eine beliebige k-selbstreduzierbare Sprache.

Zunächst kann man mit Satz 2 feststellen, dass $\overline{L}$ eine d-selbstreduzierbare Sprache sein muss. Denn es wurde gezeigt, dass das Komplement einer d-selbstreduzierbaren Sprache k-selbstreduzierbar sein muss und umgekehrt.

Satz 4 wiederum sagt aus, dass jede d-selbstreduzierbare Sprache in der Klasse NP ist. Somit folgt, dass $\overline{L} \in \text{NP}$ gelten muss.

In Abschnitt 2.1 wurde geschrieben, dass eine Sprache zur Klasse coNP gehört, wenn ihr Komplement zur Klasse NP gehört. Da hier bereits gezeigt wurde, dass das Komplement der k-selbstreduzierbaren Sprache L in NP ist, folgt unmittelbar, dass $L \in \text{coNP}$ gilt. □

5 Selbstreduktion und Vollständigkeit

Es existieren weitere Eigenschaften von Sprachen, die sich auf deren Komplexität auswirken können. Tatsächlich können diese, wenn sie mit Selbstreduzierbarkeit zusammenfallen, die Komplexität einer Sprache noch weiter verringern. Dies hat alleine noch nicht viel Auswirkung, allerdings existieren besonders wichtige Sprachen, die selbstreduzierbar sind. Speziell existieren NP-vollständige und PSPACE-vollständige Sprachen, die jeweils d-selbstreduzierbar beziehungsweise T-selbstreduzierbar sind.

Manche Eigenschaften sorgen zusammen mit Selbstreduzierbarkeit, dass die entsprechende Sprache in P ist. Sollte dies für eine der NP-vollständigen oder PSPACE-vollständigen Sprachen gelten, so hätte das große Auswirkung auf die Informatik.

In dieser Arbeit soll als Beispiel gezeigt werden, dass die bekannte Sprache SAT d-selbstreduzierbar ist. Darüberhinaus wird gezeigt, was die Komplexität einer d-selbstreduzierbaren Sprache ist, wenn sie P-selektiv ist.

5.1 SAT ist d-selbstreduzierbar

SAT ist die Sprache der erfüllbaren aussagenlogischen Formeln. Eine aussagenlogische Formel ist ein Wort, welches eine beliebige Anzahl an Variablen hat, die beliebig oft vorkommen können und durch Konnektoren miteinander verbunden sind. So eine Formel ist erfüllbar, wenn eine Belegung existiert, welche diese erfüllt. Eine Belegung ist eine Zuweisung der Variablen mit den Werten 0 und 1. So eine Belegung erfüllt eine Formel, wenn diese nach Einsetzen dieser Werte für die entsprechenden Variablen semantisch äquivalent zu 1 ist.

Für diese Arbeit sollen die Variablen einer aussagenlogischen Formel durchgehend nummeriert sein. Wenn eine Formel n Variablen hat, so soll für jede Zahl i mit $1 \leq i \leq n$ eine Variable x_i in der Formel vorkommen. Ist dies nicht der Fall, so können die Variablen einfach umbenannt werden. Man kann einmal durch die Formel laufen und jeder Variablen eine neue Bezeichung zuweisen.

Die Syntax lässt sich durch Induktion definieren. Hierbei sollen x eine Variable, 0 und 1 Konstanten, und ϕ und ψ aussagenlogische Formeln sein. Dann können aussagenlogische Formeln wie folgt geschrieben werden.

- 0
- 1
- x
- $\neg\phi$

- $(\phi \wedge \psi)$
- $(\phi \vee \psi)$
- $(\phi \rightarrow \psi)$
- $(\phi \leftrightarrow \psi)$

Um die Selbstreduzierbarkeit von SAT zu zeigen, müssen aus einer gegebenen Formel neue Formeln erzeugt werden, für die eine Variable durch eine Konstante ersetzt wird. Dies soll mit $\phi[x_i/0]$ beziehungsweise $\phi[x_i/1]$ beschrieben werden, wobei ϕ eine aussagenlogische Formel ist. Dies sind dann die Formeln, die durch Ersetzen der Variable x_i in ϕ mit 0 beziehungsweise 1 entstehen.

Als erstes muss für die Selbstreduzierbarkeit von SAT eine polynomially related Ordnung $\prec$ definiert werden. Hierfür werden die eben beschriebenen Formeln verwendet.

Definition 13. *Gegeben seien zwei aussagenlogische Formeln ϕ und ψ. Dann gilt* $\psi \prec \phi \iff \exists i \in \mathbb{N}\colon \psi = \phi[x_i/0] \vee \psi = \phi[x_i/1]$.

Außerdem wird eine Reduktionsfunktion f benötigt, die eine Anzahl von Formeln, die laut $\prec$ "kleiner" als die Eingabe sind, und eine Boolsche Funktion α erzeugt, die disjunktiv ist.

Definition 14. *Gegeben sei eine aussagenlogische Formel ϕ mit $k \in \mathbb{N}$ Variablen. Dann ist*

$$f(\phi) = \left\{ \begin{array}{ll} (\phi[x_1/0], \phi[x_1/1], \alpha_2), & k > 0 \\ (\alpha_0), & k = 0 \end{array} \right\},$$

wobei α_i eine i-stellige disjunktive Boolsche Funktion ist.

Es ist offensichtlich, dass die von f erzeugten Wörter "kleiner" als die Eingabe sind. Mit diesen Definitionen soll nun folgender Satz bewiesen werden.

Satz 6. *SAT ist d-selbstreduzierbar mit der Ordnung $\prec$ und der Reduktionsfunktion f.*

Beweis. Zunächst muss gezeigt werden, dass $\prec$ die Eigenschaften von Definition 8 erfüllt. Die erste Eigenschaft besagt, dass man in polynomieller Zeit bestimmen kann, ob eine Formel "kleiner" als eine andere ist. Dafür soll hier eine Turingmaschine beschrieben werden, die das berechnet.

Diese Turingmaschine soll zwei aussagenlogische Formeln ψ und ϕ als Eingabe bekommen und eine 1 als Ausgabe haben, wenn die erste Formel "kleiner" als die zweite ist, also wenn $\psi \prec \phi$ gilt. Ansonsten soll 0 ausgegeben werden.

Am Anfang bestimmt die Turingmaschine die Anzahl der Variablen in ϕ und speichert diese auf einem Arbeitsband. Diese Anzahl soll k sein. Danach soll für alle Zahlen i von 1 bis k auf zwei weiteren Arbeitsbändern je $\phi[x_i/0]$ und $\phi[x_i/1]$ geschrieben werden. Dann soll ψ mit diesen beiden Formeln verglichen werden. Wenn ψ gleich einer dieser Formeln ist, soll direkt eine 1 auf das Ausgabeband geschrieben werden und die Berechnung wird beendet. Ansonsten wird mit der nächsten Zahl fortgesetzt.

Sobald die Vergleiche für alle Zahlen i durchgeführt wurde und die Berechnung noch nicht beendet wurde, soll die Turingmaschine eine 0 auf das Ausgabeband schreiben und die Berechnung beenden.

Die Anzahl der Variablen ist polynomiell über die Länge der Formeln beschränkt. Daher werden nur polynomiell viele Formeln aus ϕ erzeugt und mit ψ verglichen. Der Vergleich von zwei Formeln ist in linearer Zeit möglich, da sie nur Symbol für Symbol verglichen werden müssen.

Die Formeln $\phi[x_i/0]$ und $\phi[x_i/1]$ lassen sich ebenfalls in polynomieller Zeit erzeugen. Hierfür muss jeweils nur einmal über die Formel gelaufen werden und zeichenweise kopiert werden, außer wenn die Turingmaschine auf die Variable x_i trifft. Dann soll stattdessen jeweils 0 beziehungsweise 1 geschrieben werden. Damit lässt sich in polynomieller Zeit bestimmen, ob eine Formel "kleiner" als eine andere ist.

Die zweite Eigenschaft besagt, dass die Länge der "kleineren" Formel polynomiell über die Länge der anderen Formel beschränkt ist. In diesem Fall kann die Länge der "kleineren" Formel nie größer sein, als die Länge der anderen Formel, da die Vorkommen der Variablen durch Konstanten ersetzt werden. Mit einer sinnvollen Kodierung können sie nie kürzer als die Konstanten 0 und 1 sein. Damit ist auch die zweite Eigenschaft erfüllt.

Die dritte Eigenschaft besagt, dass jede $\prec$-Kette polynomiell über die Länge der "größten" Formel beschränkt ist. Hierfür sei ϕ_1 eine aussagenlogische Formel, die k Variablen hat. Jede Formel ϕ_2, für die $\phi_2 \prec \phi_1$ gilt, muss dann $k-1$ Variablen haben, da genau eine Variable von ϕ_1 durch eine Konstante ersetzt wird. Für jede Formel ϕ_3 gilt wiederum, dass sie $k-2$ Variablen hat, wenn $\phi_3 \prec \phi_2$ gilt. Es gilt dann aber auch $\phi_3 \prec \phi_2 \prec \phi_1$.

Dies lässt sich weiter fortsetzen bis irgendwann eine beliebige Kette $\phi_k \prec ... \prec \phi_1$ erzeugt wird. Für die Formel ϕ_k gilt nun, dass sie 0 Variablen haben muss. Wenn ϕ_k keine Variablen hat, so können keine Formeln durch Ersetzen von Variablen durch Konstanten erzeugt werden. Damit existieren keine Formeln ϕ_{k+1}, sodass $\phi_{k+1} \prec \phi_k$ gilt. Die Länge so einer Kette ist also maximal k. Dies entspricht der Anzahl an Variablen der "größten" Formel in der Kette und ist polynomiell beschränkt. Damit ist auch die Länge jedes beliebigen $\prec$-Kette polynomiell beschränkt. Die dritte Eigenschaft ist auch erfüllt.

Damit wurde gezeigt, dass alle drei Eigenschaften aus Definition 8 durch $\prec$ erfüllt werden. Es fehlt noch zu zeigen, dass die Reduktionsfunktion f in polynomieller Zeit berechenbar ist. Auch diese soll durch eine Turingmaschine berechnet werden, die folgend beschrieben wird.

Als Eingabe bekommt diese Turingmaschine eine aussagenlogische Formel ϕ. Wenn ϕ keine Variablen hat, so ist diese Formel für jede beliebige Belegung semantisch äquivalent zu einer konstanten 1 oder 0. Dies lässt sich in polynomieller Zeit bestimmen. Es lässt sich dann eine 0-stellige Boolsche Funktion konstruieren, die bei leerer Eingabe den enstprechenden Wert ausgibt. Für diesen Fall ist die Reduktionfunktion f in polynomieller Zeit berechenbar.

Wenn ϕ mindestens eine Variable hat, hat sie somit auch mindestens die Variable x_1. Es werden die Formeln $\phi[x_1/0]$ und $\phi[x_1/1]$ erzeugt und auf das Ausgabeband geschrieben. Wie bereits gezeigt wurde, ist dies für jede beliebige Variable in polynomieller

Zeit möglich und somit auch für die Variable x_1.

Es wird dann eine 2-stellige Boolsche Funktion α konstruiert, die bei 1 ausgibt, sobald eine der Eingaben 1 ist. Wenn alle Eingaben 0 sind, so soll die Ausgabe 0 sein. Damit ist α disjunktiv. Diese Funktion ist unabhängig von der Länge des Eingabewortes, da es immer die gleiche Funktion ist. Dies benötigt somit nur konstante Zeit.

Daraus folgt, dass sowohl $\prec$ polynomially related ist als auch dass f in polynomieller Zeit berechenbar ist. Es muss noch gezeigt werden, dass die Ausgabe von α immer dann 1 ist, wenn die als Eingabe verwendete Formel erfüllbar ist, und immer dann 0 ist, wenn diese Formel unerfüllbar ist.

Der Fall $k = 0$ ist hierbei trivial, da die übergebene Formel keine Variablen enthält und somit semantisch äquivalent zu einer Konstanten ist. Dies wird in der Reduktionsfunktion bestimmt und es wird eine Boolsche Funktion ausgegeben, die bei leerer Eingabe die entsprechende Konstante ausgibt.

Für den Fall $k > 0$ wird die Boolsche Funktion α eine 2-stellige Funktion sein. Sie gibt dann eine 1 aus, wenn die charakteristische Funktion entweder mit $\phi[x_1/0]$ oder $\phi[x_1/1]$ als Eingabe eine 1 ausgibt. Wenn die charakteristische Funktion für beide 0 ausgibt, dann gibt auch α eine 0 aus. Das heißt, die Ausgabe von α ist dann 1, wenn $\phi[x_1/0]$ oder $\phi[x_1/1]$ erfüllbar sind.

Wenn $\phi[x1_/0]$ oder $\phi[x_1/1]$ erfüllbar sind, so existiert eine erfüllende Belegung, die mindestens eine dieser Formeln erfüllt. Diese Belegung kann man dann mit der Zuweisung einer neuen Variable erweitern, welche die neue Variable x_1 sein soll. Dieser neuen Variable wird dann 0 oder 1 zugewiesen, entsprechend der Formel, welche durch die ursprüngliche Belegung erfüllt wird. Die daraus erzeugte Belegung erfüllt dann ϕ, da das Ersetzen der Variable x_1 in ϕ entweder $\phi[x_1/0]$ oder $\phi[x_1/1]$ entsteht und die restliche Belegung die entsprechende Formel erfüllt.

Wenn beide Formeln $\phi[x1_/0]$ und $\phi[x_1/1]$ unerfüllbar sind, so muss auch ϕ unerfüllbar sein. Wäre ϕ erfüllbar, gäbe es eine Belegung, die x_1 entweder eine 0 oder eine 1 zuweist. Der Rest der Belegung müsste dann auch entweder $\phi[x1_/0]$ oder $\phi[x_1/1]$ erfüllen. Dann wären diese aber nicht unerfüllbar. Dies ist ein Widerspruch.

Damit ist die Ausgabe von α genau dann 1, wenn $\phi \in \mathrm{SAT}$ gilt. □

5.2 d-selbstreduzierbare Sprachen und P-Selektivität

Es existieren einige Eigenschaften, die zusammen mit Selbstreduzierbarkeit die Komplexität einer Sprache verändern. Als Beispiel soll hier mit P-Selektivität gezeigt werden, wie sich dies auf die Komplexität einer d-selbstreduzierbaren Sprache auswirkt. P-Selektivität wird wie folgt definiert.

Definition 15 ([4]). *Gegeben sei ein Alphabet Σ, eine Sprache $L \subseteq \Sigma^*$ und zwei beliebige Wörter $x, y \in \Sigma^*$. L* ist P-selektiv*, wenn eine Funktion f existiert, sodass f in polynomieller Zeit berechenbar ist und folgende Eigenschaften gelten:*

1. $f(x, y) = x$ *oder* $f(x, y) = y$
2. $x \in L \vee y \in L \implies f(x, y) \in L$

Eine Sprache ist somit P-selektiv, wenn eine Funktion existiert, die bei Eingabe von zwei Wörtern das Wort ausgibt, welches am ehesten zur Sprache gehört. Wenn mindestens eines der zwei Wörter in der Sprache ist, wird die Ausgabe der Funktion garantiert auch in der Sprache sein.

Nun kann man sich überlegen, ob eine Sprache existieren kann, die sowohl d-selbstreduzierbar als auch P-selektiv ist, beziehungsweise welche Auswirkungen dies auf so eine Sprache hätte. Hierzu soll der folgende Satz bewiesen werden.

Satz 7. *Gegeben sein ein Alphabet* Σ *und eine Sprache* $L \subseteq \Sigma^*$. Wenn L d-selbstreduzierbar und P-selektiv ist, dann gilt $L \in \mathrm{P}$.

Beweis. Um diesen Satz zu beweisen soll gezeigt werden, dass für jede Sprache, die diese Eigenschaften hat, eine deterministische Turingmaschine existiert, welche diese Sprache in polynomieller Zeit entscheiden.

Sei $L \subseteq \Sigma^*$ eine beliebige d-selbstreduzierbare und P-selektive Sprache. Laut Definition 11 existiert eine Reduktionsfunktion f, die bei Eingabe x die Ausgabe $(\alpha, y_1, ..., y_k)$ hat, wobei α eine disjunktive k-stellige Boolsche Funktion ist und $x, y_1, ..., y_k \in \Sigma^*$ gilt. Weiterhin existiert laut Definition 15 eine Funktion g, die zwei Wörter als Eingabe bekommt und das Wort ausgibt, welches am ehesten zur Sprache L gehört.

Für diese Sprache L wird die Turingmaschine M_P konstruiert, welche durch Algorithmus 4 beschrieben wird.

Als erstes soll gezeigt werden, dass M_P einen polynomiellen Zeitbedarf hat. Zunächst kann man wieder feststellen, dass die Ausgabe der Reduktionsfunktion f nur polynomiell lang sein kann, da f polynomielle Zeit für die Berechnung braucht. Für die Funktion g gilt ebenfalls, dass die Ausgabe nur polynomielle Länge haben kann, da diese gleich einer der beiden Eingaben ist.

Wenn Arbeitsband 2 leer ist, bedeutet das, dass die letzte Ausgabe von f nur die Boolsche Funktion α enthielt. Da für diese dann gilt, dass sie 0-stellig ist, kann sie mit leerer Eingabe simuliert werden. Hierfür wird nur polynomielle Zeit benötigt, da α selbst eine polynomielle Länge haben muss und eine 0-stellige Boolsche Funktion zu berechnen in polynomieller Zeit möglich ist.

Auf Arbeitsband 1 werden nur die Boolschen Funktionen, die durch f erzeugt werden, geschrieben. Auf Arbeitsband 2 werden die restlichen Wörter $y_1, ..., y_k$ der Ausgabe geschrieben. Auf Arbeitsband 3 schlussendlich wird eines dieser Wörter geschrieben. Damit haben alle diese Arbeitsbänder maximal polynomiell lange Bandinhalte, die dementsprechend in polynomieller Zeit löschbar sind. Das Schreiben von polynomiell langen Wörter auf Arbeitsbänder benötigt offensichtlich nur polynomielle Zeit.

Das Wort auf Arbeitsband 3 wird innerhalb eines Schleifendurchlaufs maximal k-mal überschrieben. Davor wurde bereits ein polynomiell langes Wort auf dieses Band geschrieben. Dieses wird dann wiederholt mit der Ausgabe von g überschrieben, wobei bereits festgestellt wurde, dass g in polynomieller Zeit berechenbar ist. Damit ist auch das wiederholte Überschreiben von Arbeitsband 3 in polynomieller Zeit möglich, da es nur polynomiell oft mit polynomiell langen Wörtern überschrieben wird.

Bereits im Beweis von Satz 4 wurde gezeigt, dass irgendwann die Ausgabe der Reduktionsfunktion f nur eine Boolsche Funktion enthält, solange f immer wieder auf Wörter angewendet wird, die "kleiner" als die vorherige Eingabe sind. Auch in diesem

Algorithm 4: Turingmaschine M_P

Input: x

```
1  Simuliere f mit Eingabe x und schreibe α auf Arbeitsband 1 und y_1, ..., y_k auf
     Arbeitsband 2;
2  while Ausgabeband ist leer do
3      if Arbeitsband 2 ist leer then
4          Simuliere α mit leerer Eingabe und schreibe das Ergebnis auf das
             Ausgabeband;
5          Halte an;
6      end
7      if k == 1 then
8          Schreibe y_1 auf Arbeitsband 3 und nenne es y;
9          Leere Arbeitsband 1;
10         Leere Arbeitsband 2;
11         Simuliere f mit Eingabe y und schreibe α auf Arbeitsband 1 und
             y_1, ..., y_k auf Arbeitsband 2;
12     else
13         Schreibe y_1 auf Arbeitsband 3 und nenne es y;
14         foreach y_i ∈ y_1, ..., y_k do
15             Simuliere g mit Eingabe (y, y_i) und überschreibe Arbeitsband 3 mit
                 der Ausgabe;
16         end
17         Leere Arbeitsband 1;
18         Leere Arbeitsband 2;
19         Simuliere f mit Eingabe y und schreibe α auf Arbeitsband 1 und
             y_1, ..., y_k auf Arbeitsband 2;
20     end
21     Leere Arbeitsband 3;
22 end
```

Algorithmus wird f immer nur mit Wörtern, die im vorherigen Schleifendurchlauf von f selbst erzeugt wurden, oder mit dem Eingabewort x als Eingabe simuliert. Damit wird auch hier die Schleife maximal polynomiell oft durchgeführt.

Da alle Schritte polynomielle Zeit benötigen und die Schleife polynomiell oft durchgeführt wird, folgt, dass M_P einen polynomiellen Zeitbedarf hat. Darüberhinaus sind alle Berechnungsschritte deterministisch, womit M_P auch eine deterministische Turingmaschine ist.

Nun muss noch gezeigt werden, dass M_P die Sprache L entscheidet. Ähnlich wie bei M_d mit Algorithmus 3 wird in der Schleife immer ein von der Reduktionsfunktion f erzeugtes Wort ausgewählt. Irgendwann muss die Ausgabe von f nur eine Boolsche Funktion α enthalten, da die Folge der ausgewählten Wörter eine $\prec$-Kette ergibt, wobei $\prec$ die polynomially related Ordnung ist, über die L selbstreduzierbar ist.

Diese $\prec$-Kette hat dabei das Eingabewort x als "größtes" Wort, womit die Länge dieser Kette laut Definition 8 polynomiell über die Länge von x beschränkt ist. Spätestens wenn diese $\prec$-Kette die maximale Länge erreicht hat, muss die Ausgabe von f mit dem "kleinsten" Wort als Eingabe nur eine Boolsche Funktion enthalten, welche 0-stellig ist. Die Ausgabe von dieser ist dann die Entscheidung für dieses "kleinste" Wort.

Da L d-selbstreduzierbar ist, lässt sich die Entscheidung für das "kleinste" Wort einer $\prec$-Kette rückwirkend auf alle anderen Wörter in der $\prec$-Kette anwenden. Solange also die richtigen Wörter ausgewählt werden, ist das Endergebnis korrekt.

Die Turingmaschine M_P muss nur im Fall, dass $k > 1$ gilt, ein Wort auswählen. Ansonsten gibt es entweder keine Wörter und es kann direkt das Endergebnis berechnet werden, oder es gibt nur ein Wort in der Ausgabe von f, welches dementsprechend ausgewählt wird.

Gilt nun $k > 1$, existieren mindestens zwei Wörter, von denen eines ausgewählt werden soll. Initial wird das erste Wort vorerst ausgewählt und auf Arbeitsband 3 gespeichert. Das Wort, dass auf Arbeitsband 3 ist, wird nun mit allen Wörtern $y_1, ..., y_k$ verglichen unter Anwendung der Funktion g und immer mit der Ausgabe überschrieben.

Die Ausgabe von g ist das Wort von den Eingaben, welches am ehesten zur Sprache L gehört. Wenn also mindestens eines der Eingaben zur Sprache L gehört, wird dies auch für die Ausgabe gelten. Wenn keines der Eingaben zur Sprache L gehört, so gilt dies auch für die Ausgabe.

Im ersten Durchlauf wird $g(y_1, y_1)$ berechnet. Die Ausgabe hiervon ist offensichtlich y_1. Im zweiten Durchlauf wird $g(y_1, y_2)$ und im dritten Durchlauf $g(g(y_1, y_2), y_3)$ berechnet. Die im vorherigen Durchlauf berechnete Ausgabe wird also im folgenden Durchlauf immer mit dem nächsten Wort als Eingabe für g verwendet.

Das erste Eingabewort der Funktion g ist somit immer das Wort, was im letzten Durchlauf von g ausgegeben wurde. Wenn für dieses bereits galt, dass es zur Sprache L gehört, so wird auch die nächste Ausgabe zu L gehören. Dies muss nicht das gleiche Wort sein, zum Beispiel wenn beide Eingaben zu L gehören, dann kann auch das zweite Wort ausgegeben werden.

Sobald also das erste Eingabewort zu L gehört, wird in jedem weiteren Durchlauf immer ein Wort ausgegeben, welches zu L gehört. Wenn im letzten Durchlauf das erste Eingabewort nicht zu L gehört, kann die Ausgabe nur zu L gehören, wenn dies für das zweite Eingabewort gilt. Dieses wird dann auch die Ausgabe sein und schließlich ausgewählt werden. Wenn bereits im ersten Durchlauf das Wort y_1 zu L gehört, so wird ebenfalls für jeden weiteren Durchlauf ein Wort aus L ausgegeben.

Solange eines der Wörter $y_1, ..., y_k$ zur Sprache L gehört, wird duchr wiederholte Anwendung von g garantiert eines dieser ausgewählt und auf Arbeitsband 3 geschrieben. Nur wenn keines der Wörter zu L gehört, wird die letzte Ausgabe von g und somit auch das Wort auf Arbeitsband 3 nicht zu L gehören.

Damit wird von M_P das korrekte Ergebnis berechnet und sie entscheidet die Sprache L. Da für M_P darüberhinaus gilt, dass sie einen polynomiellen Zeitbedarf hat, folgt schlussendlich $L \in \mathrm{P}$. □

Wie man hier sieht, ist eine Sprache in polynomieller Zeit entscheidbar, wenn sie sowohl d-selbstreduzierbar als auch P-selektiv ist. Wenn man dies mit SAT und der

Frage P = NP? betrachtet, können leicht zwei Aussagen getroffen werden.

Korollar 4. *SAT ist P-selektiv* $\implies$ $P = NP$

Korollar 5. $P \neq NP$ $\implies$ *SAT ist nicht P-selektiv*

In Kapitel 2 wurde gezeigt, dass jede Sprache, die in polynomieller Zeit reduzierbar auf eine Sprache aus P ist, selbst auch in P liegt. Da SAT NP-vollständig ist, ist jede Sprache aus NP in polynomieller Zeit auf diese reduzierbar. Wenn nun SAT P-selektiv wäre, dann wäre jede Sprache aus NP auf eine Sprache in P in polynomieller Zeit reduzierbar. Das würde bedeuten, dass P = NP gilt.

Gilt wiederum P $\neq$ NP, so kann SAT nicht P-selektiv sein. Denn sonst wäre jede Sprache aus NP auf eine Sprache aus P in polynomieller Zeit reduzierbar, was aber zu einem Widerspruch führen würde.

6 Schluss

In dieser Arbeit wurden die Grundlagen von Selbstreduzierbarkeit vorgestellt und gezeigt, wie verschiedene Arten von Selbstreduzierbarkeit sich auf die Komplexität einer Sprache auswirken. Darüberhinaus wurde ein Beispiel gezeigt, wie man mit Selbstreduzierbarkeit in Verbindung mit anderen Eigenschaften von Sprachen genauere Beziehungen von Komplexitätsklassen aufzeigen könnte.

Es gibt aber noch weitere Möglichkeiten, wie man Selbstreduzierbarkeit definieren kann. Zum einen kann man andere Ordnungen verwenden anstelle von polynomially related Ordnungen. So existieren zum Beispiel length-decreasing Ordnungen, für die ein "kleineres" Wort auch echt kürzer als das "größere" Wort ist [2]. Dies ist eine Erweiterung von polynomially related Ordnungen, jede length-decreasing Ordnung ist auch polynomially related.

Eine andere Ordnung, die auch verwendet werden kann, ist die word-decreasing Ordnung. Bei dieser ist das "kleinere" Wort entweder echt kürzer als das "größere" Wort oder gleich lang aber lexikographisch kleiner [2]. Dies kann allerding zu exponentiell langen Ordnungsketten führen, womit word-decreasing Ordnungen nicht polynomially related sind. Dennoch lässt sich damit eine andere Art von Selbstreduzierbarkeit definieren, die andere Erkenntnisse bezüglich Komplexität bringt.

Selbstreduzierbarkeit in dieser Arbeit verwendet die Klasse P, um den Zeitbedarf für die Selbstreduktion zu beschränken. Dies sorgt allerdings dafür, dass alle Sprachen in P implizit selbstreduzierbar sind. Um diese Komplexitätsklasse und weitere Teilklassen zu untersuchen, müsste die Komplexität der Selbstreduktion anders beschränkt werden, so dass man auch diese Klassen genauer untersuchen kann. Beispielsweise gibt es eine Variante von Selbstreduzierbarkeit, die logarithmischen Speicher verwendet [2].

Da diese Arbeit nur die Grundlagen von Selbstreduzierbarkeit zeigen soll, wurden nur grundlegende Komplexitätsklassen untersucht sowie grundlegene Reduktionen verwendet. In der Arbeit von Mundhenk wurde zum Beispiel auch speziellere Klasen untersucht, wie eine Klasse von Sprachen, die beinahe in P sind [6].

In dieser Arbeit wurde sich nur mit Entscheidungsproblemen befasst, doch existieren selbstverständlich noch weitere Arten von Problemen. Man könnte zum Beispiel untersuchen, wie sich Selbstreduzierbarkeit auf Suchprobleme oder Zählprobleme auswirkt. Speziell könnte es interessant sein, welche Komplexität die Suchvariante beziehungsweise die Zählvariante eines selbstreduzierbaren Entscheidungsproblem hat.

Es wurde in dieser Arbeit gezeigt, dass wenn SAT, eine d-selbstreduzierbare und NP-vollständige Sprache, auch P-selektiv ist, dass daraus P = NP folgt. Solche Folgerungen lassen sich noch mit weiteren Eigenschaften zeigen. So lassen sich mit Tally Sets und Sparse Sets zeigen, dass wenn eine d-selbstreduzierbare Sprache auf ein solches reduzierbar über $\leq_m^P$ ist, dass diese in P ist [3].

Etwas ähnliches lässt sich mit T-selbstreduzierbaren Sprachen zeigen. Wenn für so eine Sprache un ihr Komplement gilt, dass beide auf jeweils auf ein Sparse Set über $\leq_m^P$ reduzierbar sind, dann ist auch diese Sprache in P [4]. Darüberhinaus, da auch PSPACE-vollständige Sprachen existieren, die T-selbstreduzierbar sind, kann sogar P = PSPACE folgen.

Literatur

[1] Klaus Ambos-Spies und Jürgen Kämper. "On disjunctive self-reducibility". In: *CSL ŝ8, Lecture Notes in Computer Science*. Bd. 385. Berlin, Heidelberg: Springer Berlin Heidelberg; 1989, S. 1–13. ISBN: 978-3-540-51659-0, 978-3-540-46736-6. DOI: 10.1007/BFb0026292,10.1007/BFb0026291. URL: https://www.tib.eu/de/suchen/id/springer%3Adoi%7E10.1007%252FBFb0026292.

[2] José L. Balcázar. "Self-reducibility, Journal of Computer and System Sciences". In: *Journal of Computer and System Sciences*. Journal of Computer and System Sciences 41.3 (1989), S. 367–388. ISSN: 0022-0000. DOI: 10.1016/0022-0000(90)90025-G. URL: https://www.tib.eu/de/suchen/id/elsevier%3Adoi%7E10.1016%252F0022-0000%252890%252990025-G.

[3] Lane A. Hemaspaandra. "The Power of Self-Reducibility: Selectivity, Information, and Approximation". In: *Lecture Notes in Computer Science* (2020), S. 19–47. ISSN: 1611-3349. DOI: 10.1007/978-3-030-41672-0_3. URL: http://dx.doi.org/10.1007/978-3-030-41672-0_3.

[4] Ker-I Ko. "On self-reducibility and weak P-selectivity, Journal of Computer and System Sciences". In: *Journal of Computer and System Sciences*. Journal of Computer and System Sciences 26.2 (1982), S. 209–221. ISSN: 0022-0000. DOI: 10.1016/0022-0000(83)90013-2. URL: https://www.tib.eu/de/suchen/id/elsevier%3Adoi%7E10.1016%252F0022-0000%252883%252990013-2.

[5] R.E. Ladner, N.A. Lynch und A.L. Selman. *A comparison of polynomial time reducibilities*. 1974. DOI: 10.1016/0304-3975(75)90016-X. URL: https://www.tib.eu/de/suchen/id/elsevier%3Adoi%7E10.1016%252F0304-3975%252875%252990016-X.

[6] Martin Mundhenk. "On self-reducible sets of low information content, Extended abstract". In: *Lecture Notes in Computer Science, Algorithms and Complexity*. Bd. 778. Berlin, Heidelberg: Springer Berlin Heidelberg; 1994, S. 203–212. ISBN: 978-3-540-57811-6, 978-3-540-48337-3. DOI: 10.1007/3-540-57811-0_17,10.1007/3-540-57811-0. URL: https://www.tib.eu/de/suchen/id/springer%3Adoi%7E10.1007%252F3-540-57811-0_17.

[7] Mitsunori Ogihara und Lane A. Hemaspaandra. *The complexity theory companion, Texts in theoretical computer science, EATCS series*. Berlin: Springer; 2002. ISBN: 3540674195. URL: https://www.tib.eu/de/suchen/id/TIBKAT%3A32427727X.