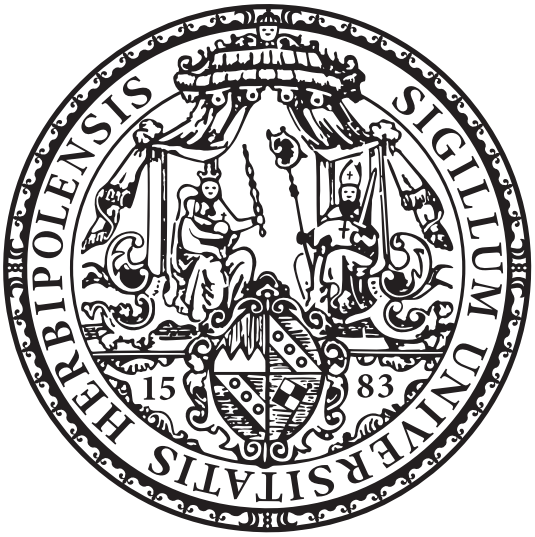
Bachelorarbeit

Komplexität der Berechnung alternativer Lösungen

Fynn Godau

Abgabedatum:	30. September 2024
Überarbeitet:	20. November 2024
Betreuer:	Prof. Dr. Christian Glaßer
	Fabian Egidy, M. Sc.



Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen und Komplexität

Kurzzusammenfassung

Wir unterscheiden für jede NP-Menge A folgende Aufgaben:

- A-Entscheidungsproblem: Entscheide bei Eingabe x, ob $x \in A$.
- A-Suchproblem: Suche bei Eingabe x einen Beweis für $x \in A$ in einem bestimmten Formalismus.

Beobachtung: Aussagen wie $x \in A$ können durch verschiedene Formalismen bewiesen werden. Jeder Formalismus führt zu einem A-Suchproblem, es gibt also verschiedene A-Suchprobleme.

Satz: Seien A eine NP-Menge und G eine beliebige Schwierigkeit.
Für jedes A-Suchproblem S existiert ein A-Suchproblem S' mit:

1. S und S' sind gleich schwer.
2. Bei S' hat die Suche nach Alternativbeweisen exakt die Schwierigkeit G.

Bedeutung: Die Schwierigkeit der Suche nach Alternativbeweisen hängt nicht von der Schwierigkeit von A, sondern ausschließlich vom verwendeten Formalismus für das A-Suchproblem ab.

- Beim A-Entscheidungsproblem ist die Antwort immer „ja“ oder „nein“, wohingegen beim A-Suchproblem im Fall $x \in A$ zusätzlich ein Beweis dafür gefordert ist.[1]
- Für den Beweisaufschrieb erlauben wir dabei alle Formalismen, in denen Beweise effizient überprüft werden können. Für jeden Formalismus ist das Suchproblem verschieden. Wir betrachten alle A-Suchprobleme, ohne uns auf ein bestimmtes festzulegen.
- Der Satz liefert konstruktiv ein A-Suchproblem S', was sehr ähnlich zum gegebenen Suchproblem S ist, jedoch für die Suche nach Alternativbeweisen einen frei gewählten Schwierigkeitsgrad hat. Genauer ist unsere Konstruktion so allgemeingültig, dass hier beliebige Schwierigkeiten von NP-Suchproblemen gewählt werden können.
- Suche nach Alternativbeweisen bedeutet, dass zu einem gegebenen Beweis für $x \in A$ ein anderer Beweis für $x \in A$ im selben Formalismus gesucht werden muss.

[1] Beim Suchproblem ist die Antwort entweder „ja“ mit Beweis in einem konkreten Formalismus, oder „nein“.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In diesem Kapitel arbeiten wir auf einen Überblick über die Arbeit und ihre Resultate hin.

1.1 Entscheidungs- und Funktionsprobleme

In der Komplexitätstheorie interessiert man sich oft nur für Entscheidungsprobleme, bei denen für jede Instanz nach einer booleschen Antwort (ja oder nein) gefragt ist. Auf diese Weise wird beispielsweise die Frage, ob eine boolesche Formel eine erfüllende Belegung hat, als Erfüllbarkeitsproblem formuliert. In der Praxis interessiert man sich hingegen meist für konkrete Lösungen und nicht nur für deren bloße Existenz;[1] im Beispiel wäre es naheliegend, nach einer konkreten erfüllenden Belegung zu fragen.

Für dieses Erfüllbarkeitsproblem ist diese bestimmte Art, einen Beweis für die Erfüllbarkeit zu notieren, sehr natürlich; schließlich besteht unsere Vorstellung von einem Lösungsalgorithmus für die Erfüllbarkeit boolescher Formeln im Wesentlichen daraus, dass viele verschiedene Belegungen getestet werden müssen. Es könnte aber sein, dass es eine nichtkonstruktive Art gibt, zu beweisen, dass eine Formel erfüllbar ist. Wer die Erfüllbarkeit von booleschen Formeln als Suchproblem definieren möchte, muss sich daher überlegen, welche genaue Art des Beweisaufschriebs zulässig sein soll. Folgendes Beispiel gibt ein konkretes Problem an, auf das dieser Fall zutrifft.

Beispiel 1.1 (Kubischer anderer Hamilton-Kreis, [Pap95])**.** Beim Problem „Hamilton-Kreis“ ist in einem Graphen ein Rundlauf über dessen Kanten gesucht, der jeden Knoten genau einmal durchläuft. „Anderer Hamilton-Kreis“ sucht in einem gegebenen Graphen mit einem ersten gegebenen Hamilton-Kreis nach einem Weiteren.[2]

Für die Untermenge der kubischen Graphen[3] ist bekannt, dass es zu einem ersten Hamilton-Kreis immer einen zweiten gibt; jedoch gelingt die Konstruktion durch keinen bekannten Polynomialzeitalgorithmus.[4] Gefragt nach einem Beweis, dass ein anderer Hamilton-Kreis existiert, könnte man daher entweder einen konkreten weiteren Hamilton-Kreis im Graphen benennen, oder man könnte – sofern zutreffend – beweisen, dass der Graph kubisch ist. Letzterer Beweis ließe sich mit heutigem Wissen nicht in einen Beweis der ersten Art überführen.

Aus diesen Betrachtungen heraus abstrahieren wir von einer auf natürlichen Lösungen basierten Vorstellung. Dafür brauchen wir eine möglichst allgemeine Möglichkeit, Beweis- oder Suchprobleme zu definieren. Dazu erinnern wir uns an die Definition von NP.

Definition 1.2. $A \in \mathrm{NP} \Leftrightarrow$ es ex. Polynom p und $B \in \mathrm{P}$ mit $(x \in A \Leftrightarrow \exists y : (|y| \leq p(|x|) \wedge (x, y) \in B)$

Der Variable y kommt dabei bereits die Bedeutung einer Lösung zu; B ist ein Polynomialzeit-Prädikat, das ausdrückt, welche y gültige Lösungen für ein gegebenes x sind. Da y als Lösung für das gewählte Prädikat B beweisen kann, dass $x \in A$ liegt, handelt es sich bei B um einen Beweisbegriff für das Problem A. Da die Definition sehr allgemein ist, gibt es viele verschiedene Möglichkeiten, den Beweisbegriff B zu

[1] Den Fokus komplexitätstheoretischer Untersuchungen auf Entscheidungsprobleme motiviert man mit der Möglichkeit, Funktionsprobleme durch Folgen von Fragen an Entscheidungsproblemen auszudrücken. Siehe [KUW88] für eine ausführliche Erklärung.

[2] Es ist bekannt, dass die beiden Probleme genau gleich schwer sind.

[3] Ein Graph ist kubisch, falls jeder Knoten genau drei Kanten besitzt.

[4] Siehe [Del+20] für einen Überblick über die aktuell besten Algorithmen.

wählen – dazu zählen eben auch solche, die auf den ersten Blick abwegig oder derzeit nicht bekannt sind. Also ist die Betrachtung der möglichen Prädikate B für ein NP-Problem A genau die gesuchte Art, Lösungen bzw. Beweise für „$x \in A$“ allgemein zu behandeln. In Abschnitt 2.2.2 folgt noch eine formale Definition. Lösungen und Beweise sind für uns in Bezug auf eine konkret und genau gestellte Frage fortan synonym.[5]

Bei der Lösung eines Entscheidungsproblems kommt es nicht primär darauf an, eine Lösung y mit $(x, y) \in B$ auszurechnen – ein Algorithmus kann bereits dann die Eingabe x akzeptieren, wenn er sicher weiß, dass es ein solches y gibt. Funktionsprobleme stellen hingegen genau die Frage, wie schwierig es ist, ein y anzugeben; diese Schwierigkeit hängt ganz maßgeblich mit davon ab, wie der Beweisbegriff B gewählt wurde. So kann man zu trivialen Entscheidungsproblemen, wie zum Beispiel Σ^*, trotzdem nichttriviale Beweise für die Zugehörigkeit von Instanzen zur Menge fordern. Solche Probleme definieren wir in Abschnitt 2.2.5.

1.2 Bisherige Forschung

In der Vergangenheit wurden Fragen nach einer alternativen Lösung vor allem als auf ein bestimmtes Problem zugeschnittene Frage aufgefasst. Oft wurde dabei auch nur mit einer – bei strenger Lesart unzureichender – eher vagen Definition des Problems gearbeitet. So entstammt Beispiel 1.1 dem Lehrbuch [Pap95]; bei näherer Betrachtung ist nicht genau beantwortet, welche Eingabe und welche Ausgabe ein Algorithmus zur Lösung der jeweiligen Probleme haben muss.

In der Arbeit [UN96] wird zwar allgemein das Prinzip „Alternative Lösung für das NP-Problem X“ auf verschiedene Probleme angewendet, dieser Schritt aber nicht präzise formalisiert. In derselben Arbeit wird auch die NP-Vollständigkeit des Entscheidungsproblems „Alternative Lösung für Nonogramm“ (ein bestimmter Rätseltyp) bewiesen.

Formalisiert wird die Frage nach der n-ten Lösung ganz allgemein und genau formuliert erst in [YS03], wie sie auch in Abschnitt 2.4 eingeführt wird. Sie kann für formal definierte Funktionsprobleme angewendet werden. Mit einem speziellen Reduktionsbegriff liefern die Autoren der Arbeit auch eine einfache Methode, Aussagen der Form „A und n-te Lösung von A sind NP-vollständig und genau gleich schwer“ zu zeigen. Dies wenden sie auf bekannte und neue NP-vollständige Probleme, darunter SAT und Sudoku,[6] an. Auf diese allgemeine, problemunabhängige Auffassung der Frage nach alternativen Lösungen fußt diese Arbeit.

1.3 Beispiele zur Berechnung weiterer Lösungen

Um eine Vorstellung davon zu erarbeiten, warum die Suche nach alternativen Lösungen interessant ist, analysieren wir beispielhaft das Problem der Suche nach Teilern, welches ein interessantes Muster bildet.

Beispiel 1.3 (Suche nach Teilern)**.** Wir untersuchen exemplarisch an einer großen Zahl $x = p \cdot q$ mit p, q prim, wie schwierig es ist, die Teiler von x zu finden.

Da $1|x$ und $x|x$ für alle $x > 0$ gilt, sind die ersten zwei Teiler sofort klar und daher trivial zu berechnen. Schwierig ist die Suche nach dem dritten Teiler, denn hierfür muss x in seine beiden Primfaktoren p, q zerlegt werden. Ist nun einer der beiden Primfaktoren bekannt, sagen wir p, so ergibt sich wiederum leicht der letzte Teiler $q = \frac{x}{p}$. Die erste, zweite, und vierte Lösung waren also leicht zu berechnen, die dritte hingegen schwer. Dies ist eine Aussage über die Schwierigkeit des Problems „nächster Teiler“.[7]

Nun betrachten wir ein weiteres Beispiel, bei dem die Schwierigkeit des Findens einer weiteren Lösung bei natürlichsprachlicher Definition unbeabsichtigt einfach ist, und es somit genau auf den gewählten Lösungsbegriff ankommt, wie schwer eine zweite Lösung zu berechnen ist. Daran wird klar, warum es sich lohnt, verschiedene Arten des Beweisaufschriebs zu betrachten.

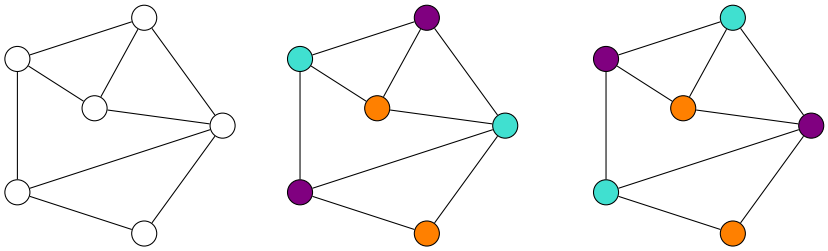
Beispiel 1.4 (3-Färbbarkeit)**.** Beim Suchproblem „3-Färbbarkeit“ wird ein Graph vorgelegt, dessen sämtliche Knoten mit drei zur Verfügung stehenden Farben so eingefärbt werden müssen, dass keine zwei

[5] Die intuitive Begründung lautet: Eine Lösung beweist, dass eine Lösung existiert. Ein Beweis ist die Lösung für die Frage nach einem Beweis.

[6] Auch wenn [Jay08] fälschlicherweise behauptet, dass die NP-Vollständigkeit von Sudoku dort tatsächlich nicht bewiesen worden sei.

[7] Das exemplarisch gefundene „Schwierigkeitsmuster“ des Problems kann auch allgemeingültig formal bewiesen werden.

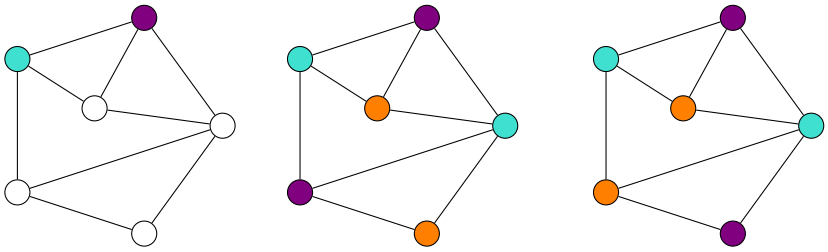
durch eine Kante verbundene Knoten mit derselben Farbe gefärbt sind. Wir betrachten eine konkrete Instanz und eine zugehörige erste und zweite Lösung.



zu färbender Graph erster gefärbter Graph zweiter gefärbter Graph

Bei genauer Betrachtung der ersten und zweiten Färbung fällt auf, dass lediglich die Farben türkis und violett vertauscht wurden. Zwar wurde formal alles richtig gemacht, doch könnte man kritisieren, dass ein bloßes Vertauschen der Farben keine echte neue Lösung darstellt. Dies spiegelt sich auch darin wider, dass sich eine derartige Permutation der Farben mit wenig Rechenaufwand erzeugen und anwenden lässt, und dass eine solche zweite Lösung immer existiert.

Will man erzwingen, dass eine zweite Lösung nicht auf diese Art zustande kommt, kann man das Problem so formulieren, dass im zu färbenden Graphen bereits zwei Farben entlang einer einzigen Kante vorgegeben werden. Mit obiger Instanz ergibt sich das folgende Beispiel:



zu färbender Graph erster gefärbter Graph zweiter gefärbter Graph

Mit dieser Formulierung[8] ist die Berechnung einer weiteren Färbung nichttrivial, liefert dafür aber garantiert eine völlig andere Lösung, falls eine existiert.

Als Alternative zur geänderten Formulierung des Problems käme noch ein geändertes Verständnis von „alternative Lösung“ an sich in Frage. Wir wünschen jedoch, allgemeine Aussagen über *alle* „2. Lösung“-Probleme zu treffen. Daher können wir es uns nicht leisten, für jedes Problem eine eigene Definition des „2. Lösung“-Problems anzusetzen, und müssen eine allgemeingültige Definition verwenden. Umso wichtiger ist es, genau auf die Formulierung des ursprünglichen Problems zu achten. Dass dies noch bei weiteren Problemen neben 3-Färbbarkeit zu beachten ist, zeigt das folgende Beispiel.

Beispiel 1.5 (Weiterer Hamilton-Kreis)**.** Wir haben in Abschnitt 1.2 bereits angesprochen, dass das Problem „Weiterer Hamilton-Kreis“ aus Beispiel 1.1 nur unzureichend präzise formuliert ist. Rein aus der Intuition heraus könnte man das zugrundeliegende Suchproblem „Hamilton-Kreis“ so formalisieren:

> Wir betrachten das Suchproblem mit der Eingabe eines gängig codierten Graphen mit den Knoten 1 bis n, für den nach einem Hamilton-Kreis gesucht wird. Die Ausgabe soll ein n-Tupel $(y_1, \ldots, y_n)$ sein, welches die Reihenfolge der Knoten im Hamilton-Kreis repräsentiert.

Unsere Interpretation als geschlossener Kreis besagt, dass $(y_1, \ldots, y_n)$ und $(y_2, \ldots, y_n, y_1)$ in Wirklichkeit der gleiche Kreis sind und letzteres daher nicht als alternative Lösung akzeptiert werden soll. Die Definition von „alternative Lösung“ soll aber allgemein bleiben und jede mögliche Antwort auf die ursprüngliche Frage berücksichtigen; sie darf keine eigenen, zusätzlichen Einschränkungen vornehmen. Wir beheben das Problem stattdessen, indem wir von vornherein nur noch sortierte Ausgaben zulassen, und fordern als Lösung ein n-Tupel $(y_1, \ldots, y_n)$ mit $y_1 = \min\{y_1, \ldots, y_n\}$. Nun ist eine alternative Lösung immer ein echt verschiedener Kreis; das zugrundeliegende Prinzip ist, dass Lösungen, die als gleich gelten, nur eine eindeutige Darstellung haben dürfen.

[8] Streng genommen haben wir auch die Formalisierung der Frage und nicht nur die Formalisierung der möglichen Lösungen geändert. Selbiges Ergebnis lässt sich codierungsabhängig auch bei unveränderter Frage erreichen; im nächsten Beispiel wird klar, nach welchem Prinzip man hier vorgehen kann.

1.4 Fragestellung und Überblick über die Ergebnisse

Die vorliegende Arbeit untersucht, wie groß der Einfluss der Formulierung des Problems auf die Komplexität alternativer Lösungen genau ist. Das Hauptresultat in Abschnitt 5.2 besagt, dass sie zumindest für die zweite Lösung *ausschließlich* von der Formulierung abhängt. Dies zeigen wir dadurch, dass wir zu jeden zwei gegebenen Funktionsproblemen A und B konstruktiv ein weiteres, ähnliches Funktionsproblem angeben, dessen Funktionswerte ebenfalls die Zugehörigkeit der Eingabe zu A beweisen, dessen Berechnung einer zweiten Lösung aber genau die Schwierigkeit von B annimmt. Dabei übernimmt B die Rolle eines beliebigen Schwierigkeitsgrades, denn jede mögliche Schwierigkeit eines Funktionsproblems wird durch mindestens ein Funktionsproblem verkörpert. Wir erlangen damit einen Einblick, welche Schwierigkeitskombinationen bei Funktionsproblemen in Bezug auf die Suche nach 1. und 2. Lösung möglich sind.

Auf dem Weg dorthin besprechen wir zuerst ausführlich die benötigten Grundlagen in Kapitel 2. Anschließend betrachten wir einen neuen Begriff für Mengen, der besagt, dass eine Menge eine leichte, indizierte Teilmenge hat. Diese Eigenschaft, die wir in Kapitel 3 einführen, bezeichnen wir als „1-paddable“. Um die gewünschte Konstruktion zu ermöglichen, werden wir 1-Paddability vom Problem A fordern; in Abschnitt 3.6 argumentieren wir, warum ein großer Teil des Resultats auch ohne die Forderung gilt.

Mit der Hilfe dieses neuen Begriffs stellen wir in Kapitel 4 zwei einfachere Konstruktionsvarianten auf, die auf den Hauptsatz hinarbeiten. In der ersten Variante in Abschnitt 4.2 wird das neu konstruierte Funktionsproblem zwar Beweise für die Menge A und auch die geforderte Schwierigkeit von B für die zweite Lösung liefern, aber noch den Kritikpunkt offenlassen, dass die Schwierigkeit der ersten Lösung von A abweichen könnte und daher das neue Problem eher unähnlich zu A ist. In der darauffolgenden zweiten Variante in Abschnitt 4.4 wird gezeigt, dass diese Kritik durch Hinzunahme einer zusätzlichen Voraussetzung ausgeräumt werden kann. Allerdings wird nicht klar, ob die neue Voraussetzung mit den Voraussetzungen aus der ersten Variante deckungsgleich ist.

Erst durch genaues Verständnis des Begriffs der 1-Paddability und der Varianten der Konstruktion können wir anschließend zum Hauptresultat gelangen. In Kapitel 5 wird die erste Konstruktionsvariante dahingehend erweitert, dass die entstehenden Probleme nun ohne die zusätzliche Voraussetzung aus der zweiten Variante bereits sehr ähnlich zu A sind. Die Beweisideen hierfür ergeben sich durch die Vorarbeiten.

Abschließend zeigen wir in Kapitel 6 die noch offenen Fragen auf, darunter die Frage, ob sich die Aussage der Konstruktion auch auf „3. Lösung“-Probleme übertragen lässt.

Kapitel 2

Definitionen

Um die Fragestellung formalisieren zu können, müssen wir zunächst einige Definitionen einführen.

2.1 Allgemeine Festlegungen

Als Berechnungsmodell für Algorithmen verwenden wir im Hintergrund allgemein Turing-Maschinen (wie üblich definiert), die normalerweise über dem festen Alphabet $\Sigma = \{0, 1\}$ arbeiten. Dies hat zur Folge, dass die Länge der Ausgabe einer Berechnung immer durch die Zeit, die benötigt wird, um sie zu erzeugen, beschränkt ist, da eine Turing-Maschine in jedem Schritt höchstens ein Zeichen Ausgabe schreiben kann.

Zu einer gegebenen, injektiven Funktion f werden wir von der Umkehrfunktion f^{-1} Gebrauch machen. Diese liefert die eindeutige Eingabe x, die in f zu einem gegebenen Funktionswert y führt. Gibt es solch ein x nicht, so fordern wir, dass die Umkehrfunktion dies erkennt und stattdessen die spezielle Ausgabe $\perp$ ausgibt. Mit dieser Wahl können andere Algorithmen, die f^{-1} verwenden, mit der Ausgabe in jedem Fall weiterarbeiten.

Definition 2.1 (Umkehrfunktion)**.** Sei $f : \Sigma^* \to \Sigma^*$ eine polynomialzeitberechenbare, injektive Funktion. Dann ist $f^{-1} : \Sigma^* \to \Sigma^* \cup \{\perp\}$ wie folgt definiert:

$$f^{-1}(y) = \begin{cases} x, & \text{falls ein } x \text{ mit } f(x) = y \text{ existiert} \\ \perp, & \text{falls } y \notin W_f \end{cases}$$

Falls f^{-1} polynomialzeitberechenbar ist, heißt f polynomialzeitinvertierbar.

Um trotz der Einschränkung auf ein festes Alphabet mit den gängigen Objekten „natürliche Zahl“ und „Liste natürlicher Zahlen“ arbeiten zu können, gebrauchen wir noch die folgenden beiden bekannten Funktionen zur Interpretation von Wörtern als natürliche Zahl und zur Kombination von mehreren natürlichen Zahlen in einem einzigen Wort in Form einer Listencodierung.

Definition 2.2 (Codierung natürlicher Zahlen)**.** Sei $dya : \mathbb{N}_0 \to \Sigma^*$ eine bijektive, polynomialzeitberechen- und -invertierbare Codierung der natürlichen Zahlen mit der Eigenschaft $x > y \Rightarrow |dya(x)| \geq |dya(y)|$.

Definition 2.3 (Listencodierung)**.** Sei $\langle x_1, \ldots, x_n \rangle$ eine totale, injektive, polynomialzeitberechen- und -invertierbare n-stellige Listencodierung für $x_i \geq 0$, also $\langle \cdot \rangle : \mathbb{N}_0^n \to \Sigma^*$. In der Listencodierung gelte $|\langle x_1, \ldots, x_n \rangle| \geq n$, d. h. jedes Element trägt mindestens ein Bit zur Länge bei.

2.2 Funktionsprobleme

Als Nächstes wollen wir die bereits in Abschnitt 1.1 erklärten Funktionsprobleme formalisieren und klären, wie wir sie intuitiv als Algorithmus notieren können.

2.2.1 Mehrwertige Funktionen

Im Kontext von deterministischen Berechnungen hängt das Ergebnis einer Berechnung immer nur vom Eingabewert ab; wird die Berechnung mehrfach durchgeführt, so ändert sich das Ergebnis nicht. Daher

modelliert man deterministische Berechnungen durch Funktionen (genauer: partielle Funktionen, denn sollte eine Berechnung niemals zum Ende kommen, so gilt das Ergebnis der Berechnung als nicht definiert).

Um den Anforderungen für Funktionsprobleme gerecht zu werden, führen wir den Begriff der *mehrwertigen Funktion* ein. Mehrwertige Funktionen sind Relationen, also Wertpaare. Wie auch bei einwertigen partiellen Funktionen wollen wir nicht fordern, dass jeder Eingabe mindestens ein Funktionswert zugeordnet ist; entsprechend einfach ist die Definition.

Definition 2.4. Ist $A \subseteq \Sigma^* \times \Sigma^*$, so ist A eine partielle, mehrwertige Funktion. Die Funktionswerte notiert man als $A(x) = \{y \mid (x, y) \in A\}$.

Wir nutzen ausschließlich mehrwertige Funktionen, die partiell sind, und verzichten im Folgenden daher auf die Angabe „partiell“. Wenn wir für alle möglichen Eingaben x eine Wertemenge $A(x)$ angeben, dann definiert dies die Menge A bereits eindeutig.

Mitunter benötigen wir zu einer mehrwertigen Funktion eine passende einwertige Funktion, die uns auf einen bestimmten aus allen verschiedenen Funktionswerten festlegt, falls solche existieren. Dies lässt sich so formalisieren:

Definition 2.5 (Einwertige Verfeinerung, [Sel94])**.** Sei $f \subseteq \Sigma^* \times \Sigma^*$ eine mehrwertige Funktion. Dann ist $f' : \Sigma^* \to \Sigma^* \cup \{\perp\}$ eine (einwertige) Verfeinerung von f, falls gilt:

- f' ist total
- $f'(x) = \perp \Leftrightarrow f(x) = \emptyset$
- $f'(x) = y \Rightarrow y \in f(x)$

Mehrwertige Funktionen sind automatisch bereits Funktionsprobleme, denen die Frage zugeordnet ist, zu einer gegebenen Eingabe eine der möglichen Ausgaben zu finden. Funktionsproblemen lässt sich immer ein Entscheidungsproblem zuordnen, welches die Frage stellt, ob eine Instanz x mindestens einen Funktionswert hat.

Definition 2.6. Sei f eine mehrwertige Funktion. Dann ist

$$dom(f) = \{\, x \mid \exists y : (x, y) \in f \,\}.$$

Damit ist $x \in dom(f) \Leftrightarrow \exists y \in f(x)$.

Wir veranschaulichen den Zusammenhang zwischen Funktions- und Entscheidungsproblemen an einem Beispiel.

Beispiel 2.7 (Definition von SAT)**.** Wir definieren das bereits angesprochene Problem der Erfüllbarkeit einer booleschen Formel als Funktionsproblem, auf die natürliche Art: Eingabe soll eine boolesche Formel und Ausgabe eine erfüllende Belegung sein.

Sei für boolesche Formeln $B(b_1, \ldots, b_n)$ über $\wedge, \vee, \neg$ eine geeignete Codierung als $\langle B \rangle$ mit $|\langle B \rangle| \geq n$ gegeben. Dann ist SAT wie folgt definiert:

$$\text{SAT} = \{\, (\langle B \rangle, b_1 \cdots b_n) \mid b_1, \ldots, b_n \in \{0, 1\} \wedge B(b_1, \ldots, b_n) = 1 \,\}$$

Mit dieser Definition ist $dom(\text{SAT})$ genau das gewohnte Entscheidungsproblem:

$$\begin{aligned} dom(\text{SAT}) &= \{\, x \mid \exists y : (x, y) \in \text{SAT} \,\} \\ &= \{\, \langle B \rangle \mid \exists\, b_1, \ldots, b_n \in \{0, 1\} : B(b_1, \ldots, b_n) = 1 \,\} \end{aligned}$$

Also enthält $dom(\text{SAT})$ solche Formeln, die erfüllende Belegungen haben.

2.2.2 Funktionskomplexitätsklassen

Wie auch bei Entscheidungsproblemen fasst man Funktionsprobleme in Komplexitätsklassen zusammen. Die Funktionskomplexitätsklasse FNP wird so definiert, dass sie genau wie in Abschnitt 1.1 skizziert zur Entscheidungsproblemklasse NP passt: Lösungen – bei uns also Funktionswerte – müssen durch ein Polynom beschränkt und außerdem leicht prüfbar sein; also muss sich leicht verifizieren lassen, ob ein gegebener Wert y tatsächlich ein Funktionswert von $f(x)$ ist. Anders formuliert muss die Zugehörigkeit eines gegebenen Paars (x, y) zum Funktionsproblem f in Polynomialzeit prüfbar sein; dies bedeutet genau $f \in \text{P}$.

Definition 2.8. Wir definieren FNP wie folgt:

$$f \in \text{FNP} \Leftrightarrow \text{es ex. Polynom } p \text{ mit } \forall (x,y) \in f : |y| \leq p(|x|) \text{ und } f \in \text{P}$$

Dass der vorgenannte Zusammenhang wirklich genau vorliegt, verdeutlicht folgender Satz.

Satz 2.9. *Sei $f \in$ FNP. Dann ist $dom(f) \in$ NP.*

Beweis. Wir zeigen, dass $dom(f)$ die in Definition 1.2 geforderte Struktur hat.
Aus $f \in$ FNP folgt schon $f \in$ P, womit gilt:

$$(x \in dom(f) \Leftrightarrow \exists y : (x,y) \in f)$$

Wir wissen ebenfalls aus $f \in$ FNP von der Existenz eines Polynoms p, was die Länge der Funktionswerte beschränkt, also $\forall (x,y) \in f : |y| \leq p(|x|)$. Daher ist ebenso wahr: Es existiert ein Polynom p und $f \in$ P mit

$$(x \in dom(f) \Leftrightarrow \exists y : (|y| \leq p(|x|) \wedge (x,y) \in f).$$

Ebendies ist laut Definition für $dom(f) \in$ NP gefordert. □

Beispiel 2.10 (SAT in FNP)**.** Das nach Beispiel 2.7 definierte SAT liegt in FNP:

- Die erfüllenden Belegungen sind als Funktionswerte nicht länger als die booleschen Formeln selbst, weil die Belegung jeder in der Formel vorkommenden Variable mit einem Bit codiert wird. Die Wahl der Codierung der booleschen Formel besagt, dass die Formel mindestens so lang ist wie die Anzahl ihrer Variablen. Daher gilt $\forall (x,y) \in \text{SAT} : |y| \leq p(|x|)$ für das Polynom $p(x) = x$.
- Es gilt SAT $\in$ P, denn es muss nur geprüft werden, ob das Paar (x,y) eine gültige boolesche Formel mit erfüllender Belegung darstellt; hierfür existiert ein Polynomialzeitalgorithmus.

Zusammen folgt SAT $\in$ FNP.

2.2.3 Funktionsprobleme als Suchprobleme zu nichtdeterministischen Maschinen

Funktionsprobleme sind auch sehr nützlich, um die Suche nach akzeptierenden Rechenwegen in nichtdeterministischen Entscheidungsmaschinen zu beschreiben. Die Motivation hierfür ist, dass wir anschließend Funktionsprobleme und nichtdeterministische Maschinen wegen der herauszuarbeitenden Ähnlichkeit zwar formal als verschieden, aber auf intuitiver Ebene als synonym behandeln können. Dafür definieren wir uns ein passendes Funktionsproblem zu jeder nichtdeterministischen Maschine:

Definition 2.11. Sei M eine nichtdeterministische Entscheidungsmaschine. Dann ist

$$f_M = \{\, (x,r) \mid r \text{ ist akzeptierender Rechenweg von } M(x) \,\}.$$

Da in Polynomialzeit durch Ablaufen geprüft werden kann, ob ein gegebener Rechenweg tatsächlich akzeptierend ist, gilt $f_M \in$ P. Falls die Rechenwege von M polynomiell in der Länge beschränkt sind, so ist zusätzlich $f_M \in$ FNP. Somit haben wir zu jedem NP-Akzeptor bereits ein geeignetes FNP-Funktionsproblem, das die Suche nach akzeptierenden Rechenwegen für diesen speziellen Akzeptor ausdrückt.

Auch umgekehrt findet man zu jedem FNP-Problem f sofort eine nichtdeterministische Maschine, deren Rechenwege auf Eingabe x jeweils eine der möglichen Lösungen für $f(x)$ testen.

Definition 2.12. Sei $f \in$ FNP, dessen Lösungen durch das Polynom p beschränkt sind. Dann sei M_f die nichtdeterministische Polynomialzeit-Entscheidungsmaschine mit folgendem Algorithmus auf Eingabe x:

1. Erzeuge nichtdeterministisch ein $y \in \Sigma^*$ mit $|y| \leq p(|x|)$
2. Akzeptiere genau dann, wenn $(x,y) \in f$

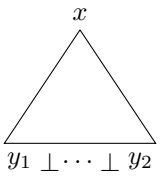
M_f entscheidet $dom(f)$, da jedes der möglichen $y \in f(x)$ auf einem Rechenweg geraten wird; M_f akzeptiert genau dann, falls ein Paar (x, y) existiert, wie in der Definition von $dom(f)$ gefordert.

Zu beachten ist, dass die Rechenwege r von M_f durch die zusätzlichen, deterministischen Rechenschritte im zweiten Schritt dieses Algorithmus nicht genau der geratenen Lösung y entsprechen; daher gilt $f_{M_g} \neq g$. Aus einem Rechenweg r in M_g lässt sich die geratene Lösung y aber leicht ablesen, und umgekehrt kann man die Rechnung in der Maschine M_g leicht ab dem zweiten Schritt simulieren, um von einer gegebenen Lösung y zum eindeutigen, akzeptierenden Rechenweg r in M_g zu kommen. An diesem informalen Argument erkennt man, dass f_{M_g} und g zwar verschieden, aber doch (mit polynomieller Unschärfe) gleich schwer sind.

2.2.4 Funktionsprobleme als nichtdeterministische Berechnung

Bislang betrachteten wir Funktionsprobleme nur im Sinne eines Suchproblems mit prüfbarer Lösung. Bei dieser Sichtweise ist jedoch nicht erkennbar, wie die entsprechende Lösung entsteht. Wir ziehen daher in Anbetracht der Überlegungen aus dem vorangegangenen Abschnitt nun der Vorstellung halber eine nichtdeterministische Maschine mit Ausgabe als Berechnungsmodell für Funktionsprobleme heran.

Da bei nichtdeterministischen Berechnungen mehrere unabhängige Rechenwege existieren können, kann es dort sein, dass verschiedene Rechenwege verschiedene Ausgaben liefern. Das passt auch genau zu der Anwendung, dass Funktionsprobleme oft mehrere Lösungen haben; so kann es beispielsweise zu einer booleschen Formel mehrere verschiedene erfüllende Belegungen geben, die alle gleichwertig als Lösung für das Problem angesehen werden. Für den Fall, dass ein Rechenweg keine Lösung gefunden hat, erlauben wir der Maschine die spezielle Ausgabe $\perp$; hat das Problem gar keine Lösung, so geben also alle Rechenwege $\perp$ aus. Bei FNP-Problemen sind die Rechenwege polynomiell in der Länge beschränkt.



Ähnlich zum Algorithmus in Definition 2.12 kann man schnell klären, wie man von einem beliebigen FNP-Problem f zu einer Berechnungsvorschrift in diesem Rechenmodell kommt:

1. Erzeuge nichtdeterministisch ein $y \in \Sigma^*$ mit $|y| \leq p(|x|)$
2. Falls $(x, y) \in f$, gib y aus.
3. Sonst, gib $\perp$ aus.

Umgekehrt werden alle getätigten Ausgaben einer polynomialzeitbeschränkten Berechnung in diesem Modell durch ein FNP-Problem beschrieben.

Wenn wir per Funktionsdefinition die Arbeitsweise eines Algorithmus mit eindeutiger Ausgabe angeben, können wir ebenfalls informal $\perp$ als Funktionswert anstatt von „*n.d.*“ nutzen, um zu betonen, dass der Algorithmus nach polynomieller Zeit ohne Ergebnis die Berechnung beendet. Mit dem Wert $\perp$ können andere Algorithmen in der Folge weiterrechnen; dabei gilt $x \cdot \perp = \perp \cdot x = \perp$ für alle x.

2.2.5 Totale Funktionsprobleme

Eine besondere Untergruppe von Funktionsproblemen sind totale Funktionsprobleme. Analog zur Definition von totalen, einwertigen Funktionen haben totale Funktionsprobleme immer mindestens einen Funktionswert. Ausgehend von der Definition von FNP-Funktionsproblemen definiert man die Klasse TFNP:

Definition 2.13. $\text{TFNP} = \{ f \in \text{FNP} \mid dom(f) = \Sigma^* \}$

TFNP-Probleme bilden in gewisser Hinsicht einen Extremwert in der Frage, wie stark sich die Komplexität eines Funktionsproblems vom zugehörigen Entscheidungsproblem unterscheiden kann: Für $f \in \text{TFNP}$ ist $dom(f) = \Sigma^*$ maximal einfach, und dennoch existieren in TFNP Funktionen, zu denen kein Polynomialzeitalgorithmus bekannt ist, der verlässlich für jede Eingabe einen Funktionswert benennen kann. Eine solche Funktion illustriert folgendes Beispiel.

Beispiel 2.14 (Primfaktorzerlegung in TFNP)**.** Ähnlich zum in Beispiel 1.3 informal vorgestellten Problem der Suche nach Teilern definieren wir das Funktionsproblem Pr als Suche nach Primfaktoren:

$$Pr(x) = \begin{cases} \{\, dya(p) \mid p \text{ ist prim und } p | dya^{-1}(x) \,\}, & \text{falls } x \neq dya(0) \\ \{\, dya(0) \,\} & \text{falls } x = dya(0) \end{cases}$$

Abgesehen von der 0, die als Sonderfall behandelt wird, sind alle primen Teiler von x höchstens so groß – und daher mit dya codiert auch höchstens so lang – wie x; somit sind die Funktionswerte durch das Polynom $p(l) = l$ in der Länge beschränkt. Wegen der Existenz eines Polynomialzeittests für Primzahlen können die Funktionswerte auch effizient geprüft werden. Aus beidem zusammen folgt $Pr \in \text{FNP}$. Aus der Existenz der Primfaktorzerlegung folgt ferner, dass die Funktion immer mindestens einen Funktionswert hat (die Zahl selbst, falls sie prim ist, und ansonsten jeden ihrer Primfaktoren einmal). Damit gilt $Pr \in$ TFNP.

Die Frage, wie schwierig es genau ist, Lösungen für Pr zu berechnen, ist von hoher praktischer Bedeutung. Man geht davon aus, dass Pr nicht effizient lösbar ist; die Sicherheit des weit verbreiteten RSA-Kryptosystems hängt davon ab. Andererseits geht man davon aus, dass weder Pr noch alle anderen TFNP-Probleme bis an die Schwierigkeit der schwersten NP-Probleme heranreichen.[1]

2.3 Reduktionsbegriffe

Die angesprochene „Schwierigkeit“ eines FNP-Problems müssen wir noch formalisieren. Dazu nutzen wir Reduktionsbegriffe, mit denen wir analog zu den bekannten Begriffen für NP-Vollständigkeit arbeiten können.

2.3.1 Reduktionen für Entscheidungsprobleme und NP-Vollständigkeit

Der bekannte Begriff der Polynomialzeitreduktion wird genutzt, um die Schwierigkeit von Entscheidungsproblemen zu vergleichen. Ist ein Problem A reduzierbar auf ein anderes Problem B, so hat das die Aussagekraft, dass A höchstens so schwer ist wie ein anderes Problem B, weil sich alle Fragen, die man mithilfe von A beantworten kann – also alle Fragen, ob $x \in A$ –, mit einer leicht zu berechnenden Funktion f in Fragen an B übersetzen kann. Die Antwort auf $x \in A$ lässt sich dann an $f(x) \in B$ ablesen. Man versieht den Begriff noch mit dem Zusatz „many-one“, um darauf hinzuweisen, dass die Reduktionsfunktion f nicht injektiv zu sein braucht. Lassen sich die Fragen von zwei Problemen auch durch das jeweils andere Problem ausdrücken, gelten die Probleme als (bis auf polynomielle Unschärfe) gleich schwer.

Definition 2.15 (polynomial-time many-one reduction)**.**

- $A \leq_m^p B \Leftrightarrow$ es ex. totales, polynomialzeitberechenbares $f : \Sigma^* \to \Sigma^*$ mit $x \in A \Leftrightarrow f(x) \in B$
- $A \equiv_m^p B \Leftrightarrow A \leq_m^p B$ und $B \leq_m^p A$

Will man für eine Komplexitätsklasse definieren, welche Probleme darin am schwersten zu lösen sind, so schaut man auf diejenigen Probleme, die die Fragen aller anderen Probleme in der Klasse ebenfalls ausdrücken können, also mindestens so schwer sind wie jedes andere Problem in der Klasse. Für die Klasse NP gelangt man so zum bekannten Begriff der NP-Vollständigkeit:

Definition 2.16 (NP-Vollständigkeit)**.** A ist NP-vollständig $\Leftrightarrow A \in$ NP und für alle $B \in$ NP gilt $B \leq_m^p A$

Bei Polynomialzeit-Reduktion muss allgemein beachtet werden, dass Σ^* keine geeignete Übersetzung für negative Instanzen bereithält. Daher gilt der folgende Spezialfall:

Satz 2.17. *Es gilt* $B = \Sigma^* \wedge A \leq_m^p B \Rightarrow A = \Sigma^*$.

Beweis. Wegen $A \leq_m^p B$ existiert eine totale Funktion f mit $x \in A \Leftrightarrow f(x) \in B$. Da $f(x) \in B = \Sigma^*$ für alle x gilt, folgt auch $x \in A$ und damit $A = \Sigma^*$. □

Die analoge Aussage für $B = \emptyset$ zeigt sich ebenso leicht. Somit sind $A \leq_m^p \Sigma^*$ und $A \leq_m^p \emptyset$ beide eine sehr starke Aussage, an der sich nicht nur Schwierigkeit, sondern auch genauer Inhalt von A ablesen lassen.

[1] „[. . . T]he existence of an FNP-complete problem in TFNP[] would imply that NP = coNP.“ [Pap94] Vgl. außerdem [Din22] für eine Diskussion über die mögliche NP-Härte von TFNP-Problemen.

2.3.2 Reduktionen für Funktionsprobleme und FNP-Vollständigkeit

Für Funktionsprobleme braucht man andere Reduktionsbegriffe – schließlich kann man für verschiedene, aber intuitiv gleich schwere Probleme f und g nicht erwarten, dass es eine Übersetzung der Frage nach einer Lösung für $f(x)$ in der Art gibt, dass für ein $z \in g(y)$ immer auch $z \in f(x)$ gilt. Stattdessen müssen wir zulassen, dass ein Algorithmus zwischengeschaltet wird, der die Frage $f(x)$ zu einer Frage $g(y)$ und die Antwort z zu einer Antwort $z' \in f(x)$ verarbeiten kann. Der Algorithmus soll also folgende Eigenschaften haben:

- Eingabe x – Algo muss Frage $f(x)$ beantworten
- Möglichkeit des Zugriffs auf $g(y)$ für jedes y – liefert Algo beliebiges $z \in g(y)$
- Ausgabe eines $z' \in f(x)$, falls ein solches existiert, andernfalls Ausgabe $\perp$
- Polynomiell beschränkte Laufzeit

Den erwähnten Zugriffsmechanismus auf g bezeichnet man als „Orakel“, den Algorithmus, der ein Orakel befragen kann, als „Orakelalgorithmus“. Das Orakel ist ein Teil der Berechnung des Algorithmus, dessen Schwierigkeit außer Acht gelassen werden soll. So kann man durch Wahl von g als Orakel bewirken, dass ein Zugriff auf jedes der Rechenergebnisse, die g liefern kann, durch „Orakelfragen“ möglich ist, ohne, dass Rechenzeit für die Beantwortung der Frage beansprucht wird. Weil die Schwierigkeit von g auf diese Art ausgeblendet wird, kann man damit ausdrücken, dass jemand, der g lösen kann, automatisch auch f lösen kann.

In vorheriger Betrachtung ist allerdings vernachlässigt, dass g bei Funktionsproblemen eine mehrwertige Funktion ist. Falls es mehrere mögliche Orakelantworten gibt, welche soll der Orakelalgorithmus dann erhalten? Wir wollen verlangen, dass der Orakelalgorithmus mit jeder der möglichen Orakelantworten korrekt arbeitet. Er muss daher mit einer beliebigen Verfeinerung (nach Definition 2.5) von g zurechtkommen. Welches Orakel O ein Orakelalgorithmus M zur Verfügung hat, notiert man mit M^O.

Definition 2.18 (Reduktionsbegriff für Funktionen, [Sel96])**.**

- $f \leq_T^p g \Leftrightarrow$ es ex. deterministischer Orakelalgorithmus M, sodass für jede Verfeinerung g' von g : $M^{g'}$ berechnet eine Verfeinerung von f
- $f \equiv_T^p g \Leftrightarrow f \leq_T^p g$ und $g \leq_T^p f$

Somit bedeutet $f \leq_T^p g$, dass $f(x)$ unter Hinzunahme polynomiell weniger Fragen[2] an g in Polynomialzeit berechenbar ist. Daneben bedeutet $f \equiv_T^p g$, dass sich f und g durch das jeweils andere Problem lösen lassen und damit gleich schwer sind.

In der Praxis weisen wir diese Reduktion meist durch Angabe einer Funktion oder eines Orakelalgorithmus nach, aus dem klar ersichtlich ist, dass sie in Polynomialzeit von einer solchen Maschine gegen alle passenden Orakel berechnet werden kann. Dabei verwenden wir $g(x)$ informell als Rückgabewert der beliebigen Verfeinerung $g'(x)$.

Analog zur NP-Vollständigkeit bezeichnet man auch in FNP die schwersten Probleme bezüglich „$\leq_T^p$“ als vollständig:

Definition 2.19 (FNP-Vollständigkeit)**.** f ist FNP-vollständig $\Leftrightarrow f \in$ FNP und für alle $g \in$ FNP gilt $g \leq_T^p f$

Beispielsweise ist SAT FNP-vollständig.[3]

2.4 Problem der Suche nach weiteren Lösungen

Zuletzt definieren wir noch einen Formalismus, der die Suche nach einer n-ten Lösung beschreibt. Dazu betrachten wir, gegeben eine Instanz und n Lösungen, die Frage nach einer $n+1$-ten Lösung als Funktionsproblem. Die bisherigen n Lösungen werden dabei zusammen mit der Instanz in einer Liste codiert. Eine gültige Frage enthält n verschiedene, gültige Lösungen; eine gültige Antwort besteht aus einer weiteren, neuen Lösung.

[2] Tatsächlich nutzen wir in dieser Arbeit immer nur eine Orakelfrage. Man kann die Definition des Reduktionsbegriffs daher entsprechend verschärfen.

[3] Es folgt, dass FP = FNP $\Leftrightarrow$ P = NP. Vgl. [Pap95] Satz 10.2.

Definition 2.20 (Weitere Lösung, [YS03])**.** Sei $f \in \text{FNP}$ und $n \geq 1$. Nun ist

$$f_{[n]} = \{ (\langle x, l_1, \ldots, l_n \rangle, l_{n+1}) \mid \forall i \in \{1, \ldots, n+1\} : [(\forall j < i : l_i \neq l_j) \wedge (x, l_i) \in f] \}.$$

Ferner sei

$$f_{[n]t} = f_{[n]} \cup \{(x, \varepsilon) \mid \underbrace{\nexists z, l_1, \ldots, l_n : x = \langle z, l_1, \ldots, l_n \rangle \wedge \forall i \in \{1, \ldots, n\} : [(\forall j < i : l_i \neq l_j) \wedge (z, l_i) \in f]}_{x \text{ kein Code für } n \text{ gültige Lösungen}}\}.$$

Durch die zweite Definition gibt es zu Eingaben, die keine gültigen, verschiedenen, gegebenen Lösungen codieren, immer eine leichte Lösung, nämlich ε. Mit dieser Wahl ist die Suche nach einer weiteren Lösung immer dann ein totales Funktionsproblem, wenn garantiert ist, dass eine solche nächste Lösung existiert.

Kapitel 3

1-Paddability

3.1 Definition

Für die anstehende Konstruktion werden wir einen relativ großen Bereich in einer der beiden vorkommenden Mengen brauchen, für den wir Instanzen speziell behandeln können. Damit wir dadurch zu den von uns gewünschten Eigenschaften gelangen können, müssen wir die Werte aus diesem Bereich leicht finden und erkennen können. Der Bereich soll daher eine unendliche, polynomialzeitentscheidbare Teilmenge sein, deren Einträge zusätzlich anhand von Indizes leicht erzeugt werden und deren Indizes auch leicht berechnet werden können. Mengen, die solche Teilmengen haben, nennen wir *1-paddable*. In Abschnitt 3.2 werden wir diese Benennung näher begründen.

Definition 3.1 (1-Paddability)**.** Sei A eine Menge.

- A heißt *1-paddable via* p, falls $p : \Sigma^* \to \Sigma^*$ total und injektiv, p, p^{-1} polynomialzeitberechenbar, und $W_p \subseteq A$.
- A heißt *1-paddable*, falls ein p existiert, sodass A 1-paddable via p ist.

Die gesuchte unendliche Teilmenge von A ist also W_p; insbesondere ist $p(x) \in A$ für alle x. Der Index eines $y \in W_p$ ist gegeben via dem polynomialzeitberechenbaren Ausdruck $x = p^{-1}(y)$. Vom Index x gelangt man zurück zu y via $p(x)$. Weil man an der Ausgabe von $p^{-1}(x)$ erkennen kann, ob $x \in W_p$ ist, gilt $W_p \in \mathrm{P}$.

Beispielhaft zeigen wir 1-Paddability für eine konkrete Menge. In Abschnitt 3.2 werden wir sehen, dass der Begriff auch auf viele weitere bekannte und praktisch relevante Mengen zutrifft.

Beispiel 3.2 (1-Paddability von SAT)**.** Wir beweisen die 1-Paddability des Entscheidungsproblems $dom(\mathrm{SAT})$ durch explizite Angabe einer 1-Padding-Funktion p mit $x = x_1 \cdots x_n$ als Eingabebits für Eingaben mit Länge $n \geq 1$.

$$p(x) = p(x_1 \cdots x_n) = \langle \left\{ \begin{array}{ll} b_1, & \text{falls } x_1 = 1 \\ \neg b_1, & \text{sonst} \end{array} \right\} \wedge \cdots \wedge \left\{ \begin{array}{ll} b_n, & \text{falls } x_n = 1 \\ \neg b_n, & \text{sonst} \end{array} \right\} \rangle$$

Derartige Funktionswerte stellen erfüllbare Formeln dar; konkret kann die Formel erfüllt werden, indem $b_i = x_i$ gesetzt wird (insgesamt also genau durch die Belegung x). Ferner setzen wir $p(\varepsilon) = \langle b_0 \vee \neg b_0 \rangle$, was durch die Belegung $b_0 = 0$ erfüllt wird. Also $p(x) \in dom(\mathrm{SAT})$ für alle $x \in \Sigma^*$.

Injektivität, Totalität und Polynomialzeitberechenbarkeit sind alle direkt klar. Auch ist p polynomialzeitinvertierbar, indem eine gegebene Instanz B mit den Variablen $b_1, \ldots, b_n$ auf den festen Wert $p(\varepsilon)$ und anschließend auf das Muster „$\bigwedge_{i \leq n} c_i$“ mit $c_i = b_i$ oder $c_i = \neg b_i$ getestet wird. Liegt das Muster vor, so lässt sich der ursprüngliche Wert mithilfe der Konstruktionsvorschrift ablesen.

Damit ist $dom(\mathrm{SAT})$ 1-paddable via p; also ist $dom(\mathrm{SAT})$ 1-paddable.

1-Paddability als Form der sehr leichten Teilmenge einer Menge ist an sich bereits ein interessanter Begriff, der bislang noch nicht bekannt war. Wir verbringen daher den Rest dieses Kapitels damit, die Eigenschaften des Begriffs herauszuarbeiten.

3.2 Zusammenhang zur Paddability

Um nicht für jede Menge einzeln die 1-Paddability zeigen zu müssen, wollen wir uns bekannte Resultate für den Begriff der *Paddability* zunutze machen. Paddability ist wie folgt definiert:

Definition 3.3 (Paddability [BH77; DK14])**.** Sei A eine Menge. A heißt *paddable*, falls ein totales, injektives $pad : \Sigma^* \times \Sigma^* \to \Sigma^*$ mit pad, pad^{-1} polynomialzeitberechenbar existiert, sodass für alle x, y gilt:

$$x \in A \Leftrightarrow pad(x, y) \in A.$$

Dann ist pad eine Padding-Funktion für A.

Die Intuition des Begriffs ist, dass mittels pad in eine Instanz x ein Wert y eincodiert werden kann, sodass $pad(x, y)$ immer dieselbe Zugehörigkeit zur Menge A hat wie x. Später können x und y aus dem Funktionswert wieder extrahiert werden.

Man weiß, dass fast alle bekannten, natürlichen, NP-vollständigen Probleme paddable sind [DK14]. Aber was nützt dieses Wissen für die 1-Paddability? Ziemlich viel, denn Paddability ist der allgemeinere Begriff, der 1-Paddability impliziert. Somit wissen wir bereits für all diese bekannten NP-vollständigen Probleme (darunter Entscheidungsprobleme wie TSP, CLIQUE, und BP), dass sie auch 1-paddable sind.

Satz 3.4. *Es gilt: $A \neq \emptyset \wedge A$ paddable $\Rightarrow A$ 1-paddable.*

Beweis. Sei pad eine Padding-Funktion für A und sei $a \in A$. Wir definieren $p(x) = pad(a, x)$. Aus $a \in A$ und der Eigenschaft der Padding-Funktion, dass $a \in A \Leftrightarrow pad(a, x) \in A$, folgt dann $p(x) \in A$ für alle x. Totalität, Injektivität, Polynomialzeitberechenbar- sowie -invertierbarkeit übertragen sich alle direkt von pad. □

Der Beweis legt eine Analogie zwischen „paddable“ und „1-paddable“ nahe, was auch die Benennung letzterens begründet:

Paddability	jedes Wort ist paddable
1-Paddability	mindestens eine positive Instanz ist paddable

Die umgekehrte Implikation der eben gezeigten Aussage gilt jedoch nicht. Dafür zeigen wir als Zwischenschritt, dass „A paddable“ und „$\overline{A}$ paddable“ dieselbe Aussage sind, bevor wir ein Gegenbeispiel angeben.

Lemma 3.5. *Sei A eine Menge. A paddable $\Leftrightarrow \overline{A}$ paddable.*

Beweis. Sei A paddable via pad. Dann gilt

$$(x \in A \Leftrightarrow pad(x, y) \in A) \Leftrightarrow (x \notin A \Leftrightarrow pad(x, y) \notin A)\,.$$

Damit ist pad auch eine Padding-Funktion für $\overline{A}$. □

Satz 3.6. *Es gibt Mengen, die nicht paddable, aber 1-paddable sind.*

Beweis. Sei $A = \Sigma^* \setminus \{\varepsilon\}$.

- Angenommen, A ist paddable. Dann ist nach Lemma 3.5 auch $\overline{A}$ paddable, also nach Satz 3.4 1-paddable via p'. Wir wissen, dass p' injektiv ist, also $p'(0)$ und $p'(1)$ unterschiedliche Werte ergeben müssen; ein Widerspruch zu $\overline{A} = \{\varepsilon\}$. Also war die Annahme falsch und A ist nicht paddable.
- Wir zeigen, dass A 1-paddable via $p(x) = 0x$ ist; damit gilt $W_p = 0 \cdot \Sigma^*$. Das leere Wort ε ist das einzige Element, was nicht in A liegt; da ε nach Definition auch nicht in W_p liegt, gilt $W_p \subseteq A$ und somit ist p eine 1-Padding-Funktion für A.

Zusammengenommen ist A 1-paddable, aber nicht paddable. □

Bei Betrachtung des vorgestellten Beispiels kann man sich fragen, ob 1-Paddability nicht eher eine Art „halbe Paddability“ ist, in dem Sinne, dass „A 1-paddable und $\overline{A}$ 1-paddable $\Leftrightarrow A$ paddable“ gelten würde. Allerdings ist die Vermutung unplausibel, da völlig unklar ist, wie aus den beiden 1-Padding-Funktionen p, p' eine Padding-Funktion $pad(x, y)$ gebaut werden sollte, ohne über das Wissen zu verfügen, ob $x \in A$ gilt. Insbesondere sind die Wertebereiche der 1-Padding-Funktionen leicht zu entscheiden; man darf daher nicht damit rechnen, dass $W_{pad} \subseteq W_p \cup W_{p'}$ ist, denn dann wäre ganz A leicht zu entscheiden, indem man die Padding-Funktion anwendet und anschließend prüft, ob die Ausgabe in W_p oder $W_{p'}$ liegt.

3.3 Dichteeigenschaften

Ein einfaches Kriterium, um Mengen grundlegend voneinander unterscheiden zu können, ist die Anzahl an Elementen. Diese misst man durch eine Dichtefunktion $C_A(n) : \mathbb{N}_0 \to \mathbb{N}_0$, die angibt, wie viele Elemente bis zur Eingabelänge n in A enthalten sind, also $C_A(n) = |\{x \in A \mid |x| \leq n\}|$. Eine Menge heißt *dünn*, wenn diese Zählfunktion durch ein Polynom beschränkt ist.

Definition 3.7 (Dünne Mengen, [BH77; DK14])**.** A ist dünn $\Leftrightarrow$ es ex. Polynom p mit $C_A(n) \leq p(n)$ für alle n.

Dünne Mengen enthalten also für jede Länge nur wenige Elemente. Wir wollen zeigen, dass Mengen, die 1-paddable sind, niemals dünn sind. Verantwortlich dafür ist der Wertebereich der 1-Padding-Funktion, der mit wenigen Elementen nicht auskommt.

Satz 3.8. *Sei A 1-paddable via p. Dann sind A und W_p beide nicht dünn.*[1]

Beweis. Angenommen, W_p sei dünn und C_{W_p} durch das Polynom q begrenzt. Da p polynomialzeitberechenbar ist, ist dessen Ausgabelänge durch ein Polynom r beschränkt, also $|p(x)| \leq r(|x|)$.

Wir halten eine Eingabelänge n fest; die 2^n verschiedenen Wörter dieser Länge bezeichnen wir mit Σ^n. Weil p injektiv ist, führt jedes dieser Eingaben zu einer unterschiedlichen Ausgabe in der Menge $p(\Sigma^n) = \{\, p(x) \mid x \in \Sigma^n \,\}$, also ebenfalls $|p(\Sigma^n)| = 2^n$.

Da $r(n)$ die längstmögliche Ausgabe von $p(x)$ mit $|x| = n$ ist, sind die Funktionswerte aus $p(\Sigma^n)$ bereits in der Anzahl $C_{W_p}(r(n))$ enthalten. Daher gilt unter Verwendung der oberen Schranke q für C_{W_p}:

$$2^n = |p(\Sigma^n)| \leq C_{W_p}(r(n)) \leq q(r(n))$$

Folglich muss gelten, dass

$$\frac{2^n}{q(r(n))} \leq 1.$$

Geht n gegen unendlich, ist die Aussage widersprüchlich, denn es ist bekannt, dass 2^n stärker wächst als jedes Polynom. Also ist W_p nicht dünn. Aus $W_p \subseteq A$ folgt $C_{W_p}(n) \leq C_A(n)$, woraus wiederum folgt, dass A ebenfalls nicht dünn ist. □

Nun wissen wir also, dass eine Menge, die 1-paddable ist, nicht nur aus wenigen Elementen besteht. Da Mengen, die paddable sind, nach Satz 3.4 auch 1-paddable sind, können auch Mengen mit Paddability-Eigenschaft nicht dünn sein.

3.4 Abgrenzung zu anderen Begriffen

In diesem Abschnitt führen wir zwei weitere Begriffe ein, die gewisse Ähnlichkeiten zu 1-Paddability haben, um sie anschließend mit 1-Paddability zu vergleichen und sie von unserem Begriff abzugrenzen.

3.4.1 P-Printable

Der Begriff *P-Printable* wird für Mengen verwendet, deren sämtliche Elemente bis zu einer Länge n sich in Polynomialzeit abhängig von n auflisten lassen.

Definition 3.9 (P-Printable, [HY84; HIS85; AR88])**.** Sei A eine Menge. Wir setzen $print_A : \{1\}^* \to \Sigma^*$ auf $print_A(1^n) = \langle x_1, \ldots, x_k \rangle$ für $\{x_1, \ldots, x_k\} = \{\, x \in A \mid |x| \leq n \,\}$.[2] Dann ist A P-Printable, falls $print_A$ polynomialzeitberechenbar ist.

Für eine Menge A, die 1-paddable via p ist, lassen sich zumindest die Elemente aus der Teilmenge W_p effizient generieren; wir geben dabei jedoch bereits polynomiell viel Zeit für die Benennung eines einzelnen Elements. Bei P-Printable muss die polynomielle Zeitschranke ausreichen, um *alle* Elemente bis zu einer gewissen Länge auf einmal zu erzeugen. Es folgt, dass eine Menge, die P-Printable ist, nur wenige Elemente haben kann. Hingegen hatten wir bereits gesehen, dass für 1-Paddability wenige Elemente nicht ausreichen.

[1] Vgl. analog [DK14] Satz 7.15.

[2] Die Definition hat die Lücke, dass keine Reihenfolge für die $x_1, \ldots, x_k$ festgelegt ist, und die angegebene Definition von $print_A$ daher nicht eindeutig ist. Die Sortierung spielt jedoch keine Rolle: Gegeben einen Polynomialzeitalgorithmus, der die Elemente unsortiert ausgibt, können wir die Elemente nachträglich sortieren und erhalten so einen neuen Polynomialzeitalgorithmus, der eine Sortierung nach Wahl liefert (zum Beispiel lexikographisch).

Satz 3.10. *Es gibt keine Menge A, die sowohl 1-paddable als auch P-Printable ist.*

Beweis. Sei A P-Printable. Wir zeigen, dass A dann dünn ist.

Gegeben sei $n \in \mathbb{N}$. Sei $\{x_1, \ldots, x_k\} = \{\, x \in A \mid |x| \leq n \,\}$. Unter Hinzunahme der Eigenschaft der Listencodierung, dass jedes Element mindestens ein Bit zur Länge beiträgt, gilt

$$C_A(n) = |\{x_1, \ldots, x_k\}| = k \leq |\langle x_1, \ldots, x_k \rangle| = |print_A(1^n)|.$$

Die Funktion $print_A$ ist nach Voraussetzung polynomialzeitberechenbar, die Funktionswerte sind also durch ein Polynom p beschränkt; es gilt damit $|print_A(1^n)| \leq p(|1^n|) = p(n)$. Zusammen gilt $C_A(n) \leq p(n)$, also ist p die gesuchte polynomielle Schranke für die Anzahl an Elementen.

Angenommen, A wäre gleichzeitig auch 1-paddable. Nach Satz 3.8 ist dann jedoch A nicht dünn, ein Widerspruch. Es gilt also „A P-Printable $\Rightarrow$ A nicht 1-paddable“, was die Aussage zeigt. □

Wir haben die beiden Begriffe damit vollständig voneinander getrennt.

3.4.2 p-immun

Mit *p-immun* bezeichnet man Mengen, die keine unendlichen, leicht entscheidbaren Teilmengen haben:

Definition 3.11 (p-immun, [HH03])**.** Sei A eine Menge. Dann heißt A p-immun, falls kein unendliches $B \subseteq A$ mit $B \in \mathrm{P}$ existiert.

Gerade solche leicht entscheidbaren Teilmengen sind von 1-Paddability jedoch gefordert. Man erkennt daher schnell, dass 1-Paddability die p-Immunität ausschließt.

Satz 3.12. *Sei A eine Menge mit A ist 1-paddable. Dann ist A nicht p-immun.*

Beweis. Sei q eine 1-Padding-Funktion für A. Wir wissen bereits $W_q \subseteq A$ und $W_q \in \mathrm{P}$. W_q ist auch unendlich, da q injektiv und total ist. Also ist A nicht p-immun. □

Dies wirft die Frage auf, ob die Rückrichtung „A nicht p-immun $\Rightarrow$ A 1-paddable“ gilt. „Nicht p-immun“ bedeutet bereits, dass eine unendliche, polynomialzeitentscheidbare Teilmenge existiert. Wir können jedoch zeigen, dass „1-paddable“ eine echt stärkere Eigenschaft ist als „nicht p-immun“.

Satz 3.13. *Es gibt Mengen, die nicht p-immun und nicht 1-paddable sind.*

Beweis. Sei $A = \{1^n\}$. Dann ist A nicht p-immun, denn es existiert die unendliche Teilmenge $A \subseteq A$ mit $A \in \mathrm{P}$. Wäre A gleichzeitig auch 1-paddable, dann wäre A nach Satz 3.8 nicht dünn. Tatsächlich ist A wegen $C_A(n) = n + 1$ jedoch dünn, ein Widerspruch. A ist also weder p-immun noch 1-paddable. □

Die Begriffe sind damit gegenseitig eingeordnet.

3.5 Beweisfunktion zur 1-Padding-Funktion

Bislang haben wir mit 1-Paddability einen Begriff für Entscheidungsprobleme definiert, bei denen allein die Zugehörigkeit zur Menge entscheidend ist. Bei einem Funktionsproblem f hilft eine gegebene 1-Padding-Funktion p für $dom(f)$ nur bedingt weiter, denn zwar kann man so Werte $x \in W_p$ ermitteln, für die es eine Lösung $(x, y) \in f$ geben muss; die Lösung y selbst bleibt jedoch unbekannt. Daher definieren wir uns *beweisbar 1-paddable* als weiteren Begriff, der bedeutet, dass genau diese Lösungen erzeugt werden können.

Definition 3.14. Sei $f \in \mathrm{FNP}$.

- f heißt *beweisbar 1-paddable via* p, q, falls $dom(f)$ 1-paddable via p, und falls $q : W_p \to \Sigma^*$ total und polynomialzeitberechenbar ist und für alle x gilt:

  $$q(p(x)) = y \text{ mit } (p(x), y) \in f.$$

  (q liefert Beweise der Zugehörigkeit zu $dom(f)$ für Instanzen aus W_p)

- f heißt *beweisbar 1-paddable*, falls p, q existieren, sodass f beweisbar 1-paddable via p, q ist.

Für natürliche Funktionsprobleme mit natürlichen 1-Padding-Funktionen ergibt sich oft die Lösung aus der Konstruktion; so auch bei der von uns zuvor aufgestellten 1-Padding-Funktion zu SAT.

Beispiel 3.15. Wir schreiben Beispiel 3.2 fort und beweisen, dass SAT beweisbar 1-paddable ist. Dazu geben wir die Beweisfunktion q explizit an. Wir können verwenden, dass die Funktion $q : W_p \to \Sigma^*$ nur Eingaben erhält, für die $p^{-1}(x) \neq \perp$ ist.

$$q(x) = \begin{cases} p^{-1}(x), & \text{falls } |p^{-1}(x)| \geq 1 \\ 0, & \text{falls } p^{-1}(x) = \varepsilon \end{cases}$$

Mit der Funktion p aus Beispiel 3.2 gilt:

- $x = \varepsilon$: Dann ist $(p(\varepsilon), q(p(\varepsilon)) = (\langle b_0 \vee \neg b_0 \rangle, 0) \in \text{SAT}$.
- $|x| \geq 1$: Dann ist $(p(x), q(p(x)) = (p(x), p^{-1}(p(x))) = (p(x), x)$. Wir haben schon in Beispiel 3.2 argumentiert, dass x eine erfüllende Belegung für $p(x)$ ist. Daher gilt $(p(x), x) \in \text{SAT}$.

Da p polynomialzeitinvertierbar ist, ist q polynomialzeitberechenbar. Folglich ist SAT beweisbar 1-paddable via p, q, was zu zeigen war.

Man darf jedoch nicht damit rechnen, zu jedem $dom(f)$ und jeder zugehörigen, konkreten 1-Padding-Funktion p eine passende Beweisfunktion zu finden. Dies entspricht folgender Aussage:

$$A \in \text{FNP und } dom(A) \text{ 1-paddable via } p \Rightarrow \text{es ex. } q \text{ mit } A \text{ beweisbar 1-paddable via } p, q$$

Um die Schwierigkeit zu erkennen, betrachten wir ein $f \in \text{TFNP}$. Da dann $dom(f) = \Sigma^*$, ist die Identitätsfunktion $p(x) = x$ eine 1-Padding-Funktion für f. Gäbe es nun ein q, sodass f beweisbar 1-paddable via p, q ist, so würde dies bedeuten, dass das polynomialzeitberechenbare q Lösungen für alle Werte in $W_p = \Sigma^*$ liefert; damit wäre bereits das gesamte, beliebige TFNP-Problem f leicht lösbar.

Auf die allgemeinere Aussage „$A \in \text{FNP}$ und $dom(A)$ 1-paddable $\Rightarrow$ A beweisbar 1-paddable“, bei der keine bestimmte 1-Padding-Funktion festgehalten wird, gehen wir in Abschnitt 6.4 als offene Frage ein. Für die folgende, dem entgegenstehende Aussage wollen wir uns noch überlegen, warum man nicht mit einem Beweis rechnen darf:

$$A \in \text{FNP und } dom(A) \text{ 1-paddable} \not\Rightarrow A \text{ beweisbar 1-paddable.}$$

Denn existiert ein $A \in \text{FNP}$ mit $dom(A)$ 1-paddable via p, sodass jedoch A nicht beweisbar 1-paddable ist, so gilt: W_p ist eine Teilmenge von $dom(A)$, zu der kein Polynomialzeitalgorithmus für alle Instanzen x einen Funktionswert $A(x)$ angeben kann. Es gäbe also FNP-Funktionen, die in keiner Verfeinerung polynomialzeitberechenbar sind, was eine sehr schwer zu zeigende Aussage ist.[3]

Da „1-paddable“ ein Begriff für Mengen und „beweisbar 1-paddable“ ein Begriff für Funktionsprobleme ist, Funktionsprobleme aber selbst wieder Mengen sind, besteht in Bezug auf ein Funktionsproblem f eine gewisse Verwechslungsgefahr. Leider sind die Aussagen „f 1-paddable“ (wobei f als Menge aufgefasst wird) und „f beweisbar 1-paddable“ nicht gleichbedeutend und dürfen daher nicht vertauscht werden, da „f beweisbar 1-paddable“ die stärkere Aussage ist.

Satz 3.16. *Sei $f \in \text{FNP}$. Dann gilt:*

1. *f beweisbar 1-paddable $\Rightarrow$ f 1-paddable*
2. *f 1-paddable $\not\Rightarrow$ f beweisbar 1-paddable*

Beweis. Wir zeigen beide Aussagen:

1. Sei f beweisbar 1-paddable via p, q. Dann ist f 1-paddable via $z(x) = (p(x), q(p(x))$.
2. Wir setzen
$$f = \{\, (1^n, y) \mid |y| = n \text{ und } n \in \mathbb{N} \,\}.$$
Dann ist f paddable via $p(x) = (1^{|x|}, x)$, aber $dom(f)$ ist dünn, denn $C_{dom(f)}(n) = n + 1$. Also ist $dom(f)$ nach Satz 3.8 nicht 1-paddable, nach Definition also f nicht beweisbar 1-paddable.

Die Ausdrücke sind damit voneinander abgegrenzt. □

[3] Wegen FP = FNP $\Leftrightarrow$ P = NP.

3.6 1-Paddability zu jeder FNP-Äquivalenzklasse

Zwar wissen wir bereits, dass nicht jedes Entscheidungsproblem 1-paddable sein kann (wegen Satz 3.6), also auch nicht zu jedem Funktionsproblem $f \in \text{FNP}$ schon $dom(f)$ 1-paddable sein kann. Jedoch wollen wir demonstrieren, dass es zu jedem Funktionsproblem f immerhin ein recht ähnliches, da genau gleich schweres Problem gibt, dessen Entscheidungsproblem die gewünschte Eigenschaft besitzt. Dies bedeutet: hält man eine gegebene Schwierigkeit in Form eines FNP-Problems f fest, so gibt es sicher ein f' mit $dom(f')$ 1-paddable, was die festgehaltene Schwierigkeit besitzt.

Satz 3.17. *Zu jedem $f \in \text{FNP}$ existiert ein $f' \in \text{FNP}$ mit $f \equiv_T^p f'$ und $dom(f')$ 1-paddable.*

Beweis. Sei $f \in \text{FNP}$. Wir definieren f' via:

$$\begin{aligned} f'(0x) &= f(x) \\ f'(1x) &= \varepsilon \\ f'(\varepsilon) &= \varepsilon \end{aligned}$$

An der Definition ist sofort ersichtlich, dass $f' \in \text{FNP}$ und $dom(f')$ 1-paddable ist via $p(x) = 1x$. Wir zeigen noch beide Richtungen der Äquivalenz $f \equiv_T^p f'$:

- $f \leq_T^p f'$ gilt via $r(x) = f'(0x)$
- $f' \leq_T^p f$ via

$$s(x) = \begin{cases} f(x_1 \cdots x_n), & \text{falls } x_0 = 0 \\ \varepsilon, & \text{sonst} \end{cases}$$

  mit $x_0 \cdots x_n$ als Zerlegung der Eingabe in ihre Buchstaben.

Die Korrektheit beider Funktionen ergibt sich direkt aus der Definition. □

Also kommen in jeder Äquivalenzklasse bezüglich „$\equiv_T^p$“ innerhalb von FNP[4] Funktionsprobleme vor, welche 1-paddable sind.

[4] Man bezeichnet diese Äquivalenzklassen als FNP-Grade.

Kapitel 4

Konstruktion eines Funktionsproblems mit gewählter Schwierigkeit für die 2. Lösung

Auf Basis der bislang besprochenen Begriffe können wir nun, wie in Abschnitt 1.4 skizziert, als Zwischenziel eine erste Konstruktion durchführen, die aus zwei Problemen $A \in \text{FNP}$ mit A 1-paddable und $B \in \text{FNP}$ ein neues Problem konstruiert, das ähnlich zu A ist, aber dessen 2. Lösung-Problem die Schwierigkeit von B annimmt. „Schwierigkeit“ können wir dabei sowohl auf das Funktionsproblem als auch auf die zugehörigen Entscheidungsprobleme beziehen. Wie formulieren diese Ziele wie folgt aus:

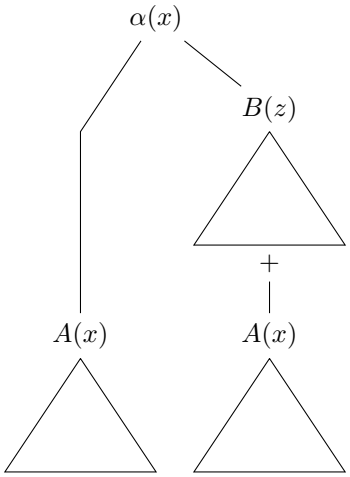
1. Das konstruierte Problem hat genau dann eine Lösung, wenn A eine Lösung besitzt.
2. Die Frage, ob das konstruierte Problem eine zweite Lösung besitzt, lässt sich als Entscheidungsfrage an das zu B zugehörige Entscheidungsproblem formulieren, und umgekehrt.
3. Die Suche nach einer zweiten Lösung im konstruierten Problem ist genauso schwer wie die Suche nach einer Lösung in B.

4.1 Idee

Wir starten mit den zwei Problemen $A, B \in \text{FNP}$; dabei müssen wir die Voraussetzung hinzunehmen, dass $dom(A)$ via p 1-paddable ist und wir so in W_p einen Arbeitsbereich mit einfach entscheidbaren Instanzen zur Verfügung haben. Um die Konstruktion besser zu verstehen, arbeiten wir nicht direkt auf Funktionsproblemen, sondern, wie in Abschnitt 2.2.4 besprochen, mit zugehörigen nichtdeterministischen Maschinen mit Ausgaben, die auf jedem Rechenweg eine mögliche Lösung für das Funktionsproblem erraten. Auf diese Weise behandeln wir A und B fortan als Maschine. Arbeitsergebnis wird die konstruierte Maschine α sein, deren zugehöriges Funktionsproblem die gewünschten Eigenschaften hat.

Wir verfolgen die folgende grundlegende Konstruktionsidee: da α genau die Menge $dom(A)$ akzeptieren soll, sagt uns die Intuition, dass der Test $x \in^? dom(A)$ auf jedem Rechenweg geleistet werden muss. Die Berechnung dieses Tests durch die Maschine $A(x)$ liefert einen Beweis $(x, y) \in A$. Dass wir wollen, dass die Komplexität von B als Schwierigkeit der 2. Lösung getroffen wird, führt uns zur Idee, einem Lösungsalgorithmus zu erlauben, den Beweis $(x, y) \in A$ für die Berechnung der 2. Lösung wiederzuverwenden, wenn er zusätzlich noch eine Instanz des Problems B löst.

Den vorläufigen Ansatz realisieren wir durch eine neue Maschine α auf Eingabe x, die zu Beginn zweigeteilt wird. Auf der linken Seite besteht sie aus der Maschine für $A(x)$; auf der rechten Seite kommt ebenfalls die Maschine für $A(x)$ vor, aber ihr ist die Maschine für $B(z)$ vorgeschaltet für ein aus x gewonnenes z.



Anhand des Aufbaus ist wie gewünscht klar, dass jedes der zwei Vorkommnisse von $A(x)$ den Test $x \in^? dom(A)$ leistet und daher $\alpha(x)$ nur dann einen akzeptierenden Rechenweg hat, wenn auch $A(x)$ einen besitzt. Es ist allerdings noch unklar, inwiefern ein Algorithmus für die Berechnung einer 2. Lösung tatsächlich eine Instanz von B lösen muss.

Präziser gibt es die folgenden zwei Schwierigkeiten: Bei der Reduktion „$B \leq$ 2. Lösung in α“ müssen wir das Problem $B(z)$ unter Zugriff auf einen effizienten Algorithmus zur Suche nach einer 2. Lösung in unserer Konstruktion lösen. Letzteres bezeichnen wir mit der Notation zur Suche nach alternativen Lösungen als $\alpha_{[1]}$. Also brauchen wir eine Möglichkeit, eine geeignete Frage als $\alpha_{[1]}$-Problem zu stellen, deren Antwort für die Beantwortung einer B-Frage nützt. Um im Rahmen von $\alpha(x)$ nach einer 2. Lösung fragen zu können, müssen wir aber zunächst selbst eine 1. Lösung angeben, können jedoch dabei $A(x)$ nicht selbst lösen. Selbst wenn wir eine solche Frage stellen könnten, wäre noch nicht sichergestellt, dass die Antwort tatsächlich aus rechten Teilbaum der Zeichnung stammt, also eine Antwort auf die Frage $B(z)$ enthält, denn $A(x)$ könnte mehrere Lösungen haben.

Aus diesen beiden Gründen müssen wir unsere Idee weiterentwickeln. Dafür machen wir uns die 1-Padding-Funktion p zunutze, die $dom(A)$ in leicht zu entscheidende Instanzen W_p und einen Rest $dom(A) \setminus W_p$, der alle schweren Instanzen enthält, aufteilt. Erstere werden wir nutzen, um mit $\alpha_{[1]}$ die Komplexität von B zu treffen.

4.1.1 Wie treffen wir die Komplexität von B mit $\alpha_{[1]}$?

Für Instanzen $x \in W_p$ ist es für den Test $x \in^? dom(A)$ nicht nötig, dass wir die Maschine $A(x)$ benutzen, denn wir wissen bereits sicher, dass $x \in dom(A)$ gilt. Wir verwenden dieses Wissen, indem wir diesen Instanzen eine einfache, eindeutige 1. Lösung in α verschaffen. Parallel zu dieser einfachen Lösung simulieren wir die Rechnung von B auf einer Instanz z. Durch die Eindeutigkeit der einfachen Lösung wird sichergestellt, dass jede 2. Lösung tatsächlich ein Problem aus B lösen muss.

Welche Instanz z soll dort gelöst werden? Am einfachsten wäre $z = x$. Es könnte intuitiverweise jedoch sein, dass B sehr ähnlich zu A, und daher ebenfalls gerade auf den Instanzen W_p besonders einfach ist. Formal bedeutet das Problem, dass wir dann während der Reduktion „$B \leq$ 2. Lösung in α“ nicht jede B-Frage beantworten könnten, sondern nur solche, die in W_p liegen.

Da p eine injektive, totale und polynomialzeitinvertierbare Funktion ist, gilt $p^{-1}(W_p) = \Sigma^*$. Demzufolge ist $z = p^{-1}(x)$ die bessere Wahl, da es dann für jede B-Frage z eine α-Instanz $x = p(z)$ gibt, die diese Frage codiert und die, gemäß obiger Konstruktion, wenn nicht schon im ersten, so spätestens im zweiten gefundenen Rechenweg beantwortet wird.

Für „$B \leq$ 2. Lösung in α“ haben wir nun also argumentiert. „2. Lösung in $\alpha \leq B$“ gilt ebenfalls bereits per Konstruktion . Zusammengenommen haben wir also mit $\alpha_{[1]}$ genau die Schwierigkeit von B getroffen.

4.1.2 Wie behandeln wir die schwierigen Instanzen aus A?

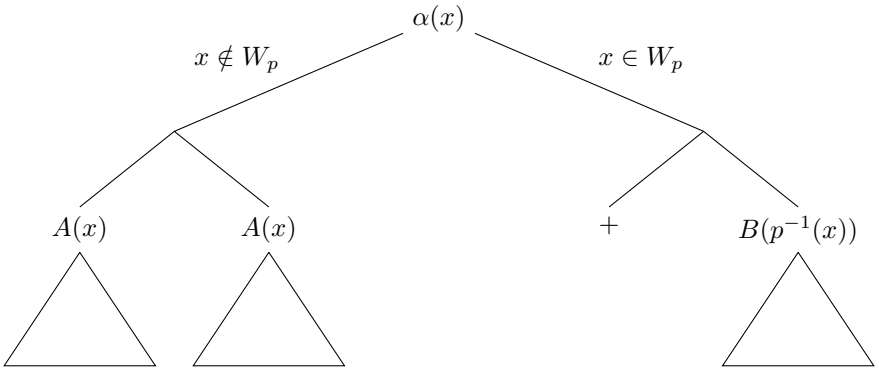
Eine andere Behandlung brauchen die Instanzen $x \notin W_p$, denn hier wissen wir die Antwort auf die Frage $x \in^? dom(A)$ nicht von vornherein, sondern müssen die Maschine $A(x)$ simulieren. Dies allein ist jedoch

noch nicht ausreichend, denn so wird die eben erreichte Reduktionsrichtung „2. Lösung in $\alpha \leq B$“ wieder zunichtegemacht: Ein Lösungsalgorithmus für $\alpha_{[1]}$ könnte dann nicht für alle Eingaben nur durch Fragen an B eine 2. Lösung finden, denn er müsste sie im Fall $x \notin W_p$ in der Maschine A suchen.

Um dies zu vermeiden, könnte man auf die Idee kommen, wieder die Maschine $B(z')$ parallel zu $A(x)$ laufen zu lassen. Dies führt jedoch zum selben Problem, denn unabhängig von der Eingabe z' würde die Antwort, dass es keinen akzeptierenden Rechenweg in $B(z')$ gibt, noch nicht bedeuten, dass es insgesamt keinen 2. Rechenweg in $\alpha(x)$ gibt: Es wäre nicht auszuschließen, dass dieser sich in $A(x)$ versteckt.

Viel einfacher lässt sich das Problem lösen, indem $A(x)$ zweimal parallel ausgeführt wird. So ist für den Fall $x \notin W_p$ die Suche nach der 2. Lösung unintuitiverweise immer einfach, was aber eben nur bedeutet, dass die Reduktionsaufgabe „2. Lösung in $\alpha \leq B$“ dann sogar ohne B gelöst werden kann. Da die Rückrichtung dank der Konstruktion für W_p unbeeinflusst bleibt, bleibt die Schwierigkeit für $\alpha_{[1]}$ insgesamt gleich B.

In der Gesamtschau ergibt sich das folgende Bild:



4.2 Konstruktion

Im Folgenden geben wir die besprochene Konstruktion nicht als Maschine, sondern direkt als Funktionsproblem an.

Definition 4.1. Sei $A, B \in \text{FNP}$ mit A 1-paddable via p. Wir definieren α via:

$$\alpha(A,B)(x) = \begin{cases} \{0,1\} \cdot A(x), & \text{falls } x \notin W_p \\ \{0\} \cup \{1\} \cdot B(p^{-1}(x)), & \text{falls } x \in W_p \end{cases}$$

Wir formalisieren die oben gestellten Forderungen an die Konstruktion, beweisen ihre Gültigkeit hier jedoch noch nicht. Der Beweis bildet sich aus einer gekürzten Version des entsprechenden Beweises zur erweiterten Konstruktion aus Kapitel 5; an dieser Stelle wäre ein formaler Aufschrieb daher eine unnötige Dopplung. Aus Symmetrie zu Satz 4.3 nutzen wir $dom(\alpha(A,B)_{[1]t})$ statt $dom(\alpha(A,B)_{[1]})$, um Aussagen über die Entscheidungsvariante zu treffen. Für nichttriviale B sind $dom(\alpha(A,B)_{[1]t})$ und $dom(\alpha(A,B)_{[1]})$ bezüglich $\leq_m^p$ äquivalent.

Satz 4.2. *Seien $A, B \in \text{FNP}$ und A 1-paddable via p. Dann gilt:*

1. $dom(\alpha(A,B)) = dom(A)$
2. $B \neq \emptyset \Rightarrow dom(\alpha(A,B)_{[1]t}) \equiv_m^p dom(B)$
3. $\alpha(A,B)_{[1]} \equiv_T^p B$

Beweis. Siehe Satz 5.3. □

Ein interessanter Sonderfall tritt auf, falls $B \in \text{TFNP}$ gegeben ist. Dann hat $\alpha(A,B)$ immer dann, wenn es eine erste Lösung für eine Eingabe gibt, auch eine zweite. Formal können wir zeigen, dass es sich bei $\alpha(A,B)_{[1]}$ bis auf ungültige Eingaben um ein TFNP-Problem handelt; dazu nutzen wir die Variante $\alpha(A,B)_{[1]t}$, bei der die ungültigen Eingaben alle akzeptiert werden. Auch diese Eigenschaft wird erst später bewiesen.

Satz 4.3. *Für $A \in$ FNP 1-paddable via p und $B \in$ TFNP gilt*

$$dom(\alpha(A,B)_{[1]t}) = dom(B) = \Sigma^*.$$

Beweis. Siehe Satz 5.5.3. □

4.3 Wie ähnlich sind A und $\alpha(A,B)$?

An der Konstruktion $\alpha(A,B)$ lässt sich kritisieren, dass das entstandene Funktionsproblem sich zu stark von A unterscheidet, als dass man von einer „Umformulierung“ sprechen könnte. Der Grund hierfür ist, dass unklar ist, ob A und $\alpha(A,B)$ als Funktionsprobleme gleich schwer sind oder ob $\alpha(A,B)$ in Wirklichkeit einfacher geworden ist.

Dieser Kritik ließe sich dadurch begegnen, dass man $\alpha(A,B) \equiv_T^p A$ zeigt. Es lässt sich schnell sehen, dass die Reduktionsrichtung $\alpha(A,B) \leq_T^p A$ gilt (und $\alpha(A,B)$ somit immerhin nicht schwerer als A geworden ist):

Lemma 4.4. *Sei $A \in$ FNP mit A 1-paddable via p. Dann gilt $\alpha(A,B) \leq_T^p A$.*

Beweis. Es gilt $\alpha(A,B) \leq_T^p A$ via folgender Funktion, die durch einen Orakelalgorithmus in Polynomialzeit berechnet werden kann:

$$r(x) = \begin{cases} 0 \cdot A(x), & \text{falls } x \notin W_p \\ 0 & \text{falls } x \in W_p \end{cases}$$

Im Fall $x \in W_{p_0}$ ist laut Definition von $\alpha(x)$ immer 0 ein Funktionswert. Falls im anderen Fall $x \notin W_p$ die Orakelfrage $A(x)$ den Wert $\perp$ ausgibt, so gibt auch r den Wert $\perp$ aus, da wir $0 \cdot \perp = \perp$ definiert hatten. Die Antwort ist auch korrekt, da eine solche Orakelantwort bedeutet, dass $A(x) = \emptyset$, also auch $\{0,1\} \cdot A(x) = \alpha(A,B)(x) = \emptyset$. Ansonsten wird der Antwort der Buchstabe 0 vorangestellt, sodass die Ausgabe wie gefordert in $\{0,1\} \cdot A(x)$ liegt. □

Beim Beweis für die andere Richtung, also $A \leq_T^p \alpha(A,B)$, tut sich hingegen eine Schwierigkeit auf: Durch die Sonderbehandlung der Werte aus W_p werden keine Beweise für Instanzen aus W_p mehr gefordert, sondern es gibt für sie die triviale Lösung 0. Das bedeutet, dass es sein könnte, dass das konstruierte Problem im Vergleich zu A echt einfacher geworden ist. Diese Schwierigkeit illustriert das folgende Beispiel genauer, dessen Idee ähnlich zu einem Argument in Abschnitt 3.5 lautet.

Beispiel 4.5. Wir betrachten die Menge $Pr \in$ TFNP aus Beispiel 2.14. Als TFNP-Menge hat Pr für jede Eingabe einen Funktionswert, also gilt $dom(Pr) = \Sigma^*$. Folglich ist $dom(Pr)$ 1-paddable via p für $p(x) = x$.

Nun betrachten wir $\alpha(Pr,B)$ für $B = \emptyset$ mit dieser konkreten Funktion p. Da $W_p = \Sigma^* = dom(Pr) = dom(\alpha(Pr,B))$, löst nach Konstruktion von α derjenige Algorithmus, der auf jeder Eingabe 0 ausgibt, bereits $\alpha(Pr,B)$. Da $\alpha(Pr,B)$ also einen Polynomialzeitalgorithmus besitzt, würde $Pr \leq_T^p \alpha(Pr,B)$ bedeuten, dass es auch für Pr einen Polynomialzeitalgorithmus gibt.[1] Ein solcher ist jedoch nicht bekannt; daher wäre $Pr \leq_T^p \alpha(Pr,B)$ eine erstaunliche Erkenntnis, mit der man nicht rechnen darf.

Das Beispiel zeigt, dass die 1-Padding-Funktion für ein $dom(A)$ zwar nur solche Werte ausgibt, für die das Entscheidungsproblem $dom(A)$ einfach zu lösen ist, aber dass für dieselben Werte trotzdem noch das Funktionsproblem A kompliziert zu lösen sein kann. Für die konkret genutzte Menge Pr lässt sich $Pr \leq_T^p \alpha(Pr,B)$ durch eine verbesserte Wahl von p erreichen; im Allgemeinen ist jedoch nicht klar, ob das auf diese Weise klappt.

4.4 Eigenschaften mit gegebener Beweisfunktion zur 1-Paddability von A

Bevor wir die erarbeitete Konstruktion modifizieren, arbeiten wir zunächst noch heraus, dass wir durch Hinzunahme einer zusätzlichen Voraussetzung die gewünschte Eigenschaft erreichen können.

[1] Warum das gilt, versteht man am schnellsten, wenn man „$\leq$“ als „höchstens so schwer wie“ auffasst. Formal kann man begründen, dass der Orakelalgorithmus für Pr aus der Reduktion $Pr \leq_T^p \alpha(Pr,B)$ nur Zugriff auf ein Orakel hat, was auf alle Eingaben den Wert 0 ausgibt. Diese Orakelantworten kann der Algorithmus sich auch selbst erzeugen, statt das Orakel zu befragen. So kommt man vom Orakel-Polynomialzeitalgorithmus zu einem gewöhnlichen Polynomialzeitalgorithmus.

Wir haben in Abschnitt 3.5 bereits den Begriff „beweisbar 1-paddable“ eingeführt, der genau zu dem oben skizzierten Problem passt. Indem wir „A beweisbar 1-paddable via p, q“ fordern, stellen wir sicher, dass W_p nur solche Instanzen erhält, die auch einfache Lösungen im Funktionsproblem A haben. Auf diese Lösungen können wir mit der polynomialzeitberechenbaren Funktion q zugreifen.

Durch die neue Voraussetzung erreichen wir das folgende Gesamtpaket an Folgerungen.

Satz 4.6. *Sei $A \in$ FNP mit A beweisbar 1-paddable via p, q, und sei $B \in$ FNP. Dann gilt:*

1. $dom(\alpha(A, B)) = dom(A)$
2. $\alpha(A, B) \equiv_T^p A$
3. $B \neq \emptyset \Rightarrow dom(\alpha(A, B)_{[1]t}) \equiv_m^p dom(B)$
4. $\alpha(A, B)_{[1]} \equiv_T^p B$

Beweis. Dank der zusätzlich gegebene Beweisfunktion q gelingt die Reduktionsrichtung $A \leq_T^p \alpha(A, B)$.

1. Folgt direkt aus Satz 4.2.1.
2. Wir beweisen $\alpha(A, B) \equiv_T^p A$:
   - $\alpha(A, B) \leq_T^p A$ folgt bereits aus Lemma 4.4.
   - $A \leq_T^p \alpha(A, B)$ mit dem folgenden Orakelalgorithmus auf Eingabe x:
     (a) Falls $x \in W_p$, gib $q(x)$ aus.
     (b) Frage die Orakelfrage $\alpha(A, B)(x)$. Falls $\perp$ ausgegeben wird, gib $\perp$ aus. Sonst bezeichne y die Antwort.
     (c) Sei $y = y_0 y_1 \cdots y_n$ in Buchstaben zerlegt für $n \geq 0$. Gib $y_1 \cdots y_n$ aus.

     Im Fall (a) gibt es keine geeignete Orakelfrage, doch die Beweisfunktion q schafft Abhilfe und kann den fehlenden Funktionswert für $A(x)$ liefern, weil $x \in W_p$.

     Im anderen Fall (b) liefert das Orakel ein Ergebnis $y_0 y_1 \cdots y_n$ mit $y_0 \in \{0, 1\}$ beliebig und $y_1 \cdots y_n \in A(x)$, weswegen ebendies in der Zeile (c) die korrekte Ausgabe ist, und der Fall $|y| = 0$ dort nie eintritt.

   Zusammengenommen folgt aus beiden Resultaten $\alpha(A, B) \equiv_T^p A$.
3. Folgt direkt aus Satz 4.2.2.
4. Folgt direkt aus Satz 4.2.3.

Damit konnten wir eine neue Folgerung hinzunehmen. □

Die Interpretation des Ergebnisses ist, dass A und $\alpha(A, B)$ wegen 1.+2. sehr ähnliche Probleme sind, aber bei $\alpha(A, B)$ die Schwierigkeit der 2. Lösung jedoch beliebig gewählt werden kann. Nach den Überlegungen aus Abschnitt 3.5 haben wir jedoch kein Argument dafür, dass jedes Funktionsproblem A, für das $dom(A)$ 1-paddable ist, bereits beweisbar 1-paddable ist.

Kapitel 5

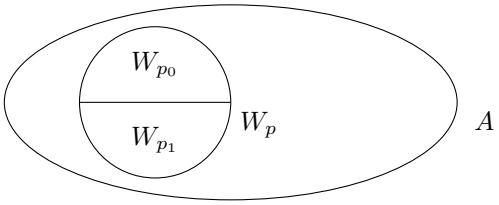
Erweiterte Konstruktion eines Funktionsproblems mit gewählter Schwierigkeit für die 2. Lösung

Unser letztendliches Ziel ist, dass gleichzeitig einerseits das neu konstruierte Problem sehr ähnlich zum ursprünglichen Problem ist (wie in Abschnitt 4.4), andererseits aber auch neben der 1-Paddability keine zusätzlichen Voraussetzungen benötigt werden (anders als in Abschnitt 4.4). Dafür erweitern wir die Konstruktion α, die in Abschnitt 4.1 erarbeitet wurde, zur Konstruktion β. Dort wird $\beta(A, B) \equiv_T^p A$ gelten, unter der einzigen Voraussetzung, dass A 1-paddable ist.

5.1 Idee

Die Idee zu β basiert auf der Idee zu α (siehe Abschnitt 4.1). In Abschnitt 4.3 hatten wir herausgearbeitet, dass $A \leq_T^p \alpha(A, B)$ die kritische Richtung für $\alpha(A, B) \equiv_T^p A$ ist. Wir müssen daher besonders auf $A \leq_T^p \beta(A, B)$ hinarbeiten, da wir $\beta(A, B) \equiv_T^p A$ erreichen wollen.

Bei der Konstruktion α hatten wir uns die Instanzen aus W_p zunutze gemacht, zu denen bereits sicher ist, dass unsere Maschine α einen akzeptierenden Rechenweg besitzen muss, um die Schwierigkeit der zweiten Lösung einzustellen. Einen ähnlichen Trick nutzen wir hier nochmals; dazu brauchen wir einen neuen Bereich an Werten, um dort auf $A \leq_T^p \beta(A, B)$ hinzuarbeiten. Einen solchen Bereich würde uns eine zusätzliche 1-Padding-Funktion für $dom(A)$ mit disjunktem Wertebereich zur ersten 1-Padding-Funktion verschaffen. Dies gelingt unter Wahrung der bestehenden Voraussetzungen, indem wir die bestehende 1-Padding-Funktion p in zwei unendlich große, disjunkte Wertebereiche W_{p_0} und W_{p_1} aufteilen, was die gewünschten zwei 1-Padding-Funktionen p_0 und p_1 liefert. Der Bereich W_{p_0} übernimmt die bisherigen Aufgaben von W_p aus α, sodass wir mit W_{p_1} nun einen freien Bereich geschaffen haben.



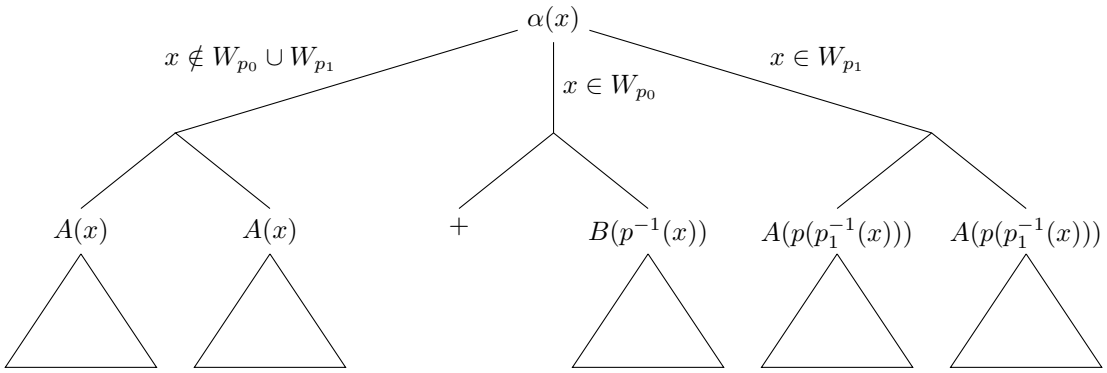
Wie können wir diesen Bereich zugunsten des Ziels $A \leq_T^p \beta(A, B)$ nutzen? Wir wollen, dass man A-Fragen beantworten kann, indem man sich Rechenwege für Instanzen aus W_{p_1} erzeugen lässt. Die Eigenschaft des Bereichs $W_{p_1} \subseteq dom(A)$, dass jede Instanz einen akzeptierenden Rechenweg haben muss, erscheint zunächst nachteilig, denn somit können wir die Maschine $A(x)$ nur dann in unserer Konstruktion integrieren, wenn wir entweder bereits wissen, dass $A(x)$ einen akzeptierenden Rechenweg hat, oder wenn parallel dazu noch ein anderer akzeptierender Rechenweg besteht. Letzteres kommt nicht in Frage, da dann nicht sichergestellt ist, dass Orakelfragen an die Konstruktion tatsächlich $A(x)$ beantworten. Ersteres können wir nicht allgemein garantieren.

Dennoch gelingt es unter Hinzunahme einer weiteren Idee: Ein Orakelalgorithmus, der $A(x)$ für ein $x \notin W_p$ lösen muss, kann bereits $\alpha(A, B)(x)$ befragen, da die Konstruktion auf solchen Fragen genau

$A(x)$ aufruft. Wir müssen uns für β also nur noch um Instanzen $x' \in W_p$ kümmern. Wegen $W_p \subseteq dom(A)$ wissen wir, dass $A(x')$ auf jeden Fall einen akzeptierenden Rechenweg besitzen muss und wir daher solche Fragen in Instanzen aus W_{p_1} codieren können.

Technisch erreichen wir diese Codierung, indem wir für Eingaben x aus W_{p_1} via p_1^{-1} den Index bezüglich p_1 berechnen. Anschließend interpretieren wir diesen Index als Index in W_p und simulieren somit die Maschine $A(p(p_1^{-1}(x))$. Es ist leicht zu sehen, dass wir damit jeden Wert aus W_p als Eingabe genau einmal treffen, denn $p_1^{-1}(W_{p_1}) = \Sigma^*$ und $p(\Sigma^*) = W_p$, also $p(p_1^{-1}(W_{p_1})) = W_p$. Wie auch für Eingaben außerhalb von W_p führen wir die Berechnung von A dabei zweimal parallel durch, um die Schwierigkeit der 2. Lösung nicht anzutasten.

Für unsere Gesamtkonstruktion β ergibt sich das folgende Bild:



5.2 Konstruktion

Die in der Idee genutzte Unterteilung von W_p geben wir explizit an, behandeln der Übersichtlichkeit halber $p(\varepsilon)$ jedoch gesondert.

Definition 5.1. Sei A 1-paddable via p. Dann ist

$$p_0(x) = p(0x)$$
$$p_1(x) = p(1x)$$

Mit dieser Definition ist $W_p = W_{p_0} \cup W_{p_1} \cup \{p(\varepsilon)\}$ eine disjunkte Zerlegung. Damit können wir die in der Idee vorgestellte Konstruktion umsetzen.

Definition 5.2. Sei $A, B \in$ FNP mit A 1-paddable via p. Wir definieren β via:

$$\beta(A,B)(x) = \begin{cases} \{0,1\} \cdot A(x), & \text{falls } x \notin W_p \vee x = p(\varepsilon) \\ \{0\} \cup \{1\} \cdot B(p_0^{-1}(x)), & \text{falls } x \in W_{p_0} \\ \{0,1\} \cdot A(p(p_1^{-1}(x))), & \text{falls } x \in W_{p_1} \end{cases}$$

5.3 Eigenschaften der Konstruktion

Wir zeigen, dass das konstruierte Problem allgemein genau dann eine Lösung hat, wenn A eine Lösung hat. Ferner zeigen wir, dass das Finden einer ersten Lösung gleich schwer ist wie das Finden einer Lösung in A; außerdem ist das Entscheiden der Existenz bzw. das Finden einer zweiten Lösung gleich schwer wie das Entscheiden bzw. Finden einer Lösung in B.

Für die Existenzvariante nutzen wir $\beta(A,B)_{[1]t}$, um im Fall $B \in$ TFNP direkt $dom(B) = \Sigma^*$ zu erzielen; dies behandeln wir in Abschnitt 5.4.[1]

Satz 5.3. *Seien $A, B \in$ FNP und $dom(A)$ 1-paddable via p. Dann gilt:*

1. $dom(\beta(A,B)) = dom(A)$
2. $\beta(A,B) \equiv_T^p A$

[1] Für nichttriviale B gilt $dom(\beta(A,B)_{[1]t}) \equiv_m^p dom(\beta(A,B)_{[1]})$.

3. $B \neq \emptyset \Rightarrow dom(\beta(A,B)_{[1]t}) \equiv_m^p dom(B)$

4. $\beta(A,B)_{[1]} \equiv_T^p B$

Beweis. Der Übersichtlichkeit halber schreiben wir β, wenn wir $\beta(A,B)$ meinen. Wann immer ein Funktionswert von β in seine Buchstaben zerlegt wird, machen wir uns zunutze, dass laut Definition jeder solche Funktionswert mindestens Länge 1 hat.

1. Wir unterscheiden die Fälle aus der Definition von β und wollen zeigen, dass $\beta(x)$ genau dann Funktionswerte hat, wenn $A(x)$ ebenfalls Funktionswerte hat.

   - 1. Fall: $x \notin W_p \vee x = p(\varepsilon)$, dann ist $\beta(x) = \{0,1\} \cdot A(x)$. Falls $A(x) = \emptyset$, so ist $\beta(x) = \{0,1\} \cdot \emptyset = \emptyset = A(x)$. Falls $A(x)$ nichtleer ist, gibt es ein $y \in A(x)$; dann ist $0y \in \beta(x)$. In beiden Fällen ist $A(x) \neq \emptyset \Leftrightarrow \beta(x) \neq \emptyset$, also $x \in dom(A) \Leftrightarrow x \in dom(\beta)$.
   - 2. Fall: $x \in W_{p_0}$, dann ist $\beta(x) = \{0\} \cup \{1\} \cdot B(p_0^{-1}(x))$. Da immer $x \in W_{p_0} \subset W_p \subseteq dom(A)$ gilt, erwarten wir, dass $\beta(x)$ immer einen Funktionswert hat. Nach der Definition gilt $0 \in \beta(x)$, die Konstruktion verhält sich also korrekt.
   - 3. Fall: $x \in W_{p_1}$, dann ist $\beta(x) = \{0,1\} \cdot A(p(p_1^{-1}(x)))$. Aus $x \in W_{p_1}$ folgt, dass der Wert $x' = p_1^{-1}(x)$ existiert. Nach den Eigenschaften der 1-Padding-Funktion p gilt $p(x') \in dom(A)$, daher existiert ein $y \in A(p(x')) = A(p(p_1^{-1}(x)))$. Nach Definition von β ist nun $0y \in \beta(x)$; da $x \in W_{p_1} \subset W_p \subseteq dom(A)$ nach Voraussetzung gilt, ist dieses Verhalten korrekt.

2. Wir beweisen $\beta \equiv_T^p A$ mittels Reduktion in beide Richtungen.

   - Es gilt $\beta \leq_T^p A$ via folgender Funktion, die durch einen Orakelalgorithmus in Polynomialzeit berechnet werden kann:

     $$r(x) = \begin{cases} 0 \cdot A(x), & \text{falls } x \notin W_p \vee x = p(\varepsilon) \\ 0 & \text{falls } x \in W_{p_0} \\ 0 \cdot A(p(p_1^{-1}(x))) & \text{falls } x \in W_{p_1} \end{cases}$$

     Im Fall $x \in W_{p_0}$ ist laut Definition von $\beta(x)$ immer 0 ein Funktionswert. Die beiden anderen Fälle $x \notin W_p \vee x = p(\varepsilon)$[2] und $x \in W_{p_1}$[3] verhalten sich genau analog zur Definition von $\beta(x)$.
   - $A \leq_T^p \beta$ via folgendem Algorithmus auf Eingabe x:
     (a) Falls $x \notin W_p$, frage Orakel β nach $\beta(x)$. Liefert es $\perp$, gib $\perp$ aus. Sonst bezeichne y die Antwort.
     (b) Falls $x \in W_p$, frage Orakel β nach $\beta(p_1(p^{-1}(x)))$. Bezeichne y die Antwort.
     (c) Sei $y = y_0 y_1 \cdots y_n$ für $n \geq 0$. Gib $y_1 \cdots y_n$ aus.

     Im Fall (a) gilt $\beta(x) = \{0,1\} \cdot A(x)$. Ist $A(x)$ leer, liefert das Orakel $\perp$ und der Algorithmus verhält sich durch Ausgabe von $\perp$ korrekt. Andernfalls liefert das Orakel ein Ergebnis $y_0 y_1 \cdots y_n$ mit $y_0 \in \{0,1\}$ und $y_1 \cdots y_n \in A(x)$, weswegen ebendies in der Zeile (c) die korrekte Ausgabe ist, und der Fall $|y| = 0$ dort nie eintritt.
     Im Fall (b) gilt für $x' = p_1(p^{-1}(x))$, also $x' \in W_{p_1}$, dass $\beta(x') = \{0,1\} \cdot A(p(p_1^{-1}(x')) = \{0,1\} \cdot A(p(p_1^{-1}(p_1(p^{-1}(x))))) = \{0,1\} \cdot A(x)$. Wegen des Falls, dass $x \in W_p$, liefert dabei $A(x)$ immer eine Ausgabe. Wie im Fall (a) wird in der Zeile (c) das erste Zeichen abgetrennt, was zum korrekten Ergebnis führt.

4. Wir beweisen zuerst die Äquivalenz $\beta_{[1]} \equiv_T^p B$, indem wir in beide Richtungen die Reduktion mithilfe von Orakelalgorithmen zeigen. Die Aussage 3. zeigen wir im Anschluss.

   - $\beta_{[1]} \leq_T^p B$ via folgendem Orakelalgorithmus auf Eingabe x':
     (a) Falls $x' \neq \langle x, l \rangle$, gib $\perp$ aus. Im Folgenden gilt $x' = \langle x, l \rangle$.
     (b) Falls $(x,l) \notin \beta$, gib $\perp$ aus.
     (c) Falls $x \notin W_{p_0}$, sei $l = y_0 y_1 \cdots y_n$ die Zerlegung in Buchstaben für $n \geq 0$. Gib dann $(1 - y_0) y_1 \cdots y_n$ aus.

[2] Falls in diesem Fall eine Orakelfrage $A(x)$ den Wert $\perp$ ausgibt, so gibt auch r korrekterweise den Wert $\perp$ aus, da wir $0 \cdot \perp = \perp$ definiert haben. Da eine solche Orakelantwort bedeutet, dass $A(x) = \emptyset$, gilt auch $\{0,1\} \cdot A(x) = \emptyset$.

[3] In diesem Fall liefert $A(p(p_1^{-1}(x)))$ immer einen Funktionswert. Siehe 1., 3. Fall.

(d) Falls $x \in W_{p_0}$ und $l = 1l'$, gib 0 aus.

(e) Falls $x \in W_{p_0}$ und $l = 0$, gib $1 \cdot B(p_0^{-1}(x))$ aus.

Wir begründen die Korrektheit des Algorithmus wie folgt:

- In den Zeilen (a) und (b) wird geprüft, ob die Eingabe x' die richtige Form für ein 2. Lösung-Problem hat. Wenn nein, liegt sie sicher nicht in der Menge $\beta_{[1]}$ und der Orakelalgorithmus darf keinen Funktionswert ausgeben. Wir wissen ab nun wegen $(x, l) \in \beta$, dass l eine korrekte 1. Lösung für $\beta(x)$ ist.
- Laut Definition von β gelangt man in allen Fällen außer $x \in W_{p_0}$ von einer gegebenen Lösung zu einer anderen Lösung, indem man das erste Bit umkehrt. Daher leistet Zeile (c) ebendies.
- Nun verbleibt noch der Fall $x \in W_{p_0}$. Die vorgelegte Lösung l stammt daher aus der Menge $\beta(x) = \{0\} \cup \{1\} \cdot B(p_0^{-1}(x))$; dort muss nach der zweiten Lösung gesucht werden. In Zeile (d) wird der Fall behandelt, dass sie aus dem rechten Teil dieser Vereinigung stammt, indem dann 0 ausgegeben wird.
- Zeile (e) behandelt den gegenteiligen Fall, dass die Lösung l aus dem linken Teil dieser Vereinigung stammt und somit $l = 0$ gilt. In diesem Fall gibt es nur dann eine zweite Lösung, wenn $B(p_0^{-1}(x))$ nicht leer ist. Ist es leer, gibt die Orakelfrage $\perp$ und der Algorithmus somit $1 \cdot \perp = \perp$ aus. Andernfalls gibt der Algorithmus eine korrekte zweite Lösung aus.

Andere Fälle können nicht vorkommen. In jedem Fall verhält der Algorithmus sich korrekt.

- $B \leq_T^p \beta_{[1]}$ lässt sich mit folgendem Orakelalgorithmus auf Eingabe x zeigen:

  (a) Sei y die Antwort auf die Orakelfrage $\beta_{[1]}(\langle p_0(x), 0\rangle)$. Falls $y = \perp$, gib $\perp$ aus.

  (b) Sonst ist $y = 1y_1 \cdots y_n$. Gib $y_1 \cdots y_n$ aus.

  Die Orakelfrage $\beta_{[1]}(\langle p_0(x), 0\rangle)$ hat die Bedeutung, dass nach einer zweiten Lösung in $\beta_{[1]}(p_0(x))$ gefragt ist, gegeben die erste Lösung 0. Wegen $p_0(x) \in W_{p_0}$ ist

  $$\beta(p_0(x)) = \{0\} \cup \{1\} \cdot B(p_0^{-1}(p_0(x))) = \{0\} \cup \{1\} \cdot B(x).$$

  Da die Antwort auf die Orakelfrage nach Definition von $\beta_{[1]}$ ungleich der bereits gegebenen Lösung 0 sein muss, hat eine etwaige Antwort die Form $1 \cdot B(x)$. Daher ist diese Annahme in Zeile (b), sowie das Verhalten, das erste Zeichen abzutrennen, korrekt. Gibt es hingegen keine Funktionswerte in $B(x)$, so ist die Orakelantwort $\perp$. In diesem Fall ist es korrekt, dass der Algorithmus in Zeile (a) ebenfalls $\perp$ ausgibt.

3. Nun zeigen wir $dom(\beta(A, B)_{[1]t}) \equiv_m^p dom(B)$.

   - Sei $b \in dom(B)$. Dann ist $dom(\beta_{[1]t}) \leq_m^p dom(B)$ via folgender polynomialzeitberechenbarer Funktion:

     $$r(x') = \begin{cases} p_0^{-1}(x), & \text{falls } x' = \langle x, l\rangle, (x, l) \in \beta, x \in W_{p_0} \text{ und } l = 0 \\ b, & \text{sonst} \end{cases}$$

     Die Korrektheit der Funktion folgt im Wesentlichen aus dem unter 4. gezeigten Algorithmus für $\beta_{[1]} \leq_T^p B$. Dortiger Algorithmus liefert allerdings für Eingaben, die keine gültige Instanz mit 1. Lösung codieren, die Ausgabe $\perp$, wohingegen r eine Reduktion von $dom(\beta_{[1]t})$ aus leistet. In $dom(\beta_{[1]t})$ sind gerade solche Eingaben laut Definition enthalten; aus diesem Grund wird für $x' \neq \langle x, l\rangle$ oder $(x, l) \notin \beta$ die Konstante $r(x') = b \in B$ ausgegeben. In den Zeilen (c) und (d) liefert obiger Algorithmus Beweise, dass $x' \in dom(\beta_{[1]})$, was für diese Fälle $r(x') = b \in dom(B)$ begründet. Aus der Korrektheit der dortigen Zeile (e) folgt auch die Korrektheit des nichttrivialen Falls in r.

   - $dom(B) \leq_m^p dom(\beta_{[1]t})$ via folgender polynomialzeitberechenbarer Funktion:

     $$s(x) = \beta_{[1]}(\langle p_0(x), 0\rangle)$$

     Die Begründung der Korrektheit dieser Reduktionsfunktion folgt aus der Argumentation für den Algorithmus in 4. zu $B \leq_T^p \beta_{[1]}$.

Damit haben wir alle Aussagen gezeigt. □

Hiermit ist der potenzielle Kritikpunkt, das konstruierte Problem sei zu unähnlich zu A, beseitigt. Wir konnten also das gewünschte Resultat zeigen, dass die Schwierigkeit der 2. Lösung von Suchproblemen unabhängig von der Schwierigkeit der 1. Lösung ist.

Bemerkung 5.4 (Interpretation als Maschinen). Wir können die Eigenschaften durch Verwendung von Definition 2.12 auch als Resultate über die Suche nach Rechenwegen in NP-Maschinen interpretieren.

Gegeben sei $A \in \mathrm{NP}$ mit A 1-paddable und eine beliebige Polynomialzeitmaschine M_1, die A akzeptiert. Ebenso sei gegeben eine beliebige nichtdeterministische Polynomialzeitmaschine M_2, deren akzeptierte Sprache wir mit B bezeichnen. Dann gibt es eine Maschine M mit den folgenden Eigenschaften:

1. M entscheidet A und somit dieselbe Sprache wie M_1.
2. Die Suche nach einem akzeptierenden Rechenweg in M ist genauso schwer wie die Suche nach einem akzeptierenden Rechenweg in M_1.
3. Der Test, ob M einen zweiten akzeptierenden Rechenweg hat, ist genauso schwer wie der Test, ob M_2 akzeptiert.
4. Die Suche nach einem zweiten akzeptierenden Rechenweg in M ist genauso schwer wie die Suche nach einem akzeptierenden Rechenweg in M_2.

Beweis. Wir wissen aus Abschnitt 2.2.3, dass $M_{f_{M_1}} \equiv_m^p M_1$ und $M_{f_{M_2}} \equiv_m^p M_2$. Die gesuchte Maschine M ist die Maschine $M_{\alpha(f_{M_1}, f_{M_2})}$. Die Eigenschaften folgen direkt aus Satz 5.3. □

5.4 Eigenschaften der Konstruktion für $B \in \mathrm{TFNP}$

In Satz 4.3 hatten wir bereits angesprochen, dass sich im Fall $B \in \mathrm{TFNP}$ als interessante Zusatzeigenschaft ergibt, dass die Suche nach einem zweiten akzeptierenden Rechenweg, gegeben eine gültige erste Eingabe, selbst ein totales Suchproblem ist. Dies gilt auch für die neue Konstruktion β, mit der wir letztlich zu diesem Gesamtergebnis kommen:

Satz 5.5. *Für $A \in \mathrm{FNP}$ mit A 1-paddable und $B \in \mathrm{TFNP}$ gilt:*

1. $dom(\beta(A, B)) = dom(A)$
2. $\beta(A, B) \equiv_T^p A$
3. $dom(\beta(A, B)_{[1]t}) = dom(B) = \Sigma^*$
4. $\beta(A, B)_{[1]} \equiv_T^p B$

Beweis. Wir tragen bereits bewiesene Aussagen zusammen.

1. Folgt direkt aus Satz 5.3.1.
2. Folgt direkt aus Satz 5.3.2.
3. Nach Satz 5.3.3 gilt
$$B \neq \emptyset \Rightarrow dom(\beta(A, B)_{[1]t}) \equiv_m^p dom(B).$$
Da $B \in \mathrm{TFNP}$, ist $dom(B) = \Sigma^*$. Aus Satz 2.17 folgt bereits $dom(\beta(A, B)_{[1]t}) = \Sigma^*$, was zu zeigen war.
4. Folgt direkt aus Satz 5.3.4.

Die vier Eigenschaften sind damit schon gezeigt. □

Mit den Voraussetzungen gilt also $\beta_{[1]}(A, B) \in \mathrm{TFNP}$.

Bemerkung 5.6 (Interpretation als Maschinen). Wiederum interpretieren wir Satz 5.5 über die Suche nach Rechenwegen in NP-Maschinen.

Gegeben sei $A \in \mathrm{NP}$ mit A 1-paddable und eine beliebige Maschine M_1, die A akzeptiert. Ebenso sei gegeben eine beliebige nichtdeterministische Polynomialzeitmaschine M_2, die immer auf mindestens einem Rechenweg akzeptiert; ihre akzeptierte Sprache ist also Σ^*. Dann gibt es eine Maschine M mit den folgenden Eigenschaften:

1. M entscheidet A und somit dieselbe Sprache wie M_1.
2. Die Suche nach einem akzeptierenden Rechenweg in M ist genauso schwer wie die Suche nach einem akzeptierenden Rechenweg in M_1.
3. M hat immer dann, wenn M akzeptiert, auch einen zweiten akzeptierenden Rechenweg.
4. Die Suche nach einem zweiten akzeptierenden Rechenweg in M ist genauso schwer wie die Suche nach einem akzeptierenden Rechenweg in M_2.

Beweis. Vergleiche Bemerkung 5.4 zusammen mit Satz 5.5. □

Hiermit ist auch für den Sonderfall derjenigen Maschinen, die immer einen zweiten Rechenweg haben, gezeigt, dass die Suche nach einem solchen zweiten Rechenweg trotzdem noch jede beliebige Schwierigkeit aus allen Schwierigkeiten totaler Funktionsprobleme, also TFNP, annehmen kann.

5.5 Eigenschaften der Konstruktion ohne $dom(A)$ 1-paddable

Wie kann man die Resultate aus Satz 5.3 anwenden, wenn nur Funktionsprobleme A und B gegeben sind, wobei $dom(A)$ nicht 1-paddable ist? Unter Anwendung von Satz 3.17 kommt man immerhin noch zu einer abgeschwächten Aussage.

Satz 5.7. *Für $A, B \in$ FNP existiert ein $C \in$ FNP mit:*

1. $C \equiv_T^p A$
2. $B \neq \emptyset \Rightarrow dom(C_{[1]t}) \equiv_m^p dom(B)$
3. $C_{[1]} \equiv_T^p B$

Beweis. Sei $A' \in$ FNP mit $A \equiv_T^p A'$ und $dom(A')$ 1-paddable nach Satz 3.17. Das gesuchte Problem C ist dann $\beta(A', B)$, denn mit Satz 5.3.2, 5.3.3 und 5.3.4 gilt:

1. $\beta(A', B) \equiv_T^p A' \equiv_T^p A$
2. $B \neq \emptyset \Rightarrow dom(\beta(A', B)_{[1]t}) \equiv_m^p dom(B)$
3. $\beta(A', B)_{[1]} \equiv_T^p B$

Dies sind genau die gesuchten Eigenschaften. □

Wir müssen also darauf verzichten, dass das Funktionsproblem C genau für solche Instanzen Lösungen hat, für die auch das Problem A Lösungen hat, verlieren also die Kontrolle über $dom(C)$. Wir erhalten allerdings trotzdem noch ein Problem, dessen Schwierigkeit der 1. und 2. Lösung wir völlig frei einstellen können. Dies bedeutet, dass jede Kombination an Schwierigkeiten für 1. und 2. Lösung bei Funktionsproblemen vorkommen kann; keine Möglichkeit kann von vornherein ausgeschlossen werden.

Satz 5.5 lässt sich auf die gleiche Weise übertragen.

Kapitel 6

Offene Fragen

6.1 Erweiterungen der Konstruktion

Unsere Konstruktion fokussiert sich bislang nur auf die erste und zweite Lösung. Hier stellen sich direkt Fragen nach einer möglichen Erweiterung:

- Kann man statt der Schwierigkeit der 1. und 2. Lösung stattdessen auch die Schwierigkeit der n- und $n+1$-ten Lösung per Konstruktion wählen?
- Kann man die Konstruktion erweitern, um zusätzlich die Schwierigkeit der 3. Lösung auf eine weitere Menge C festzulegen?
- Gibt es eine Iteration der Konstruktion, die beliebig die Schwierigkeiten von der 1. bis zur n-ten Lösung festlegt?

Eine positive Antwort auf die letzte Frage würde die ersten beiden Fragen ebenfalls mitbeantworten. Wir zeigen im Folgenden einen Ansatz für eine Antwort auf die zweite Frage; er ließe sich jedoch beliebig oft iterieren.

Eine vielversprechende Idee ist es, den Wertebereich der 1-Padding-Funktion W_p weiter zu unterteilen und damit einen neuen Bereich W_{p_2} zu schaffen, in dem in der Konstruktion auf die Schwierigkeit der 3. Lösung hingearbeitet wird. Dazu müssen in diesem Bereich sowohl 1. als auch 2. Lösung triviale Beweise sein; die dritte Lösung soll sich durch $C(p_2^{-1}(x))$ ergeben. Die Komplexität der Berechnung der 1. und 2. Lösung insgesamt bleibt damit unverändert. Bei Eingabe $p_2(x')$ ist so garantiert, dass eine Antwort $C(p_2^{-1}(p_2(x'))) = C(x)$ spätestens in der dritten Lösung vorkommt. Damit die Schwierigkeit für die dritte Lösung genau ge- und nicht übertroffen wird, muss in jedem anderen Bereich sichergestellt werden, dass dort jeweils entweder keine, einer oder genau drei Rechenwege existieren, ähnlich wie bei β genau kein oder zwei Rechenwege im Bereich W_{p_1} existieren.

Wie man der Skizze bereits ansieht, ist die Konstruktion nunmehr noch technischer geworden. Allerdings ist die Idee im Grunde nichts neues, sondern nur eine Fortführung der Idee für β. Man könnte die Idee weiter ausformulieren, ihre Iteration notieren und versuchen, ihre Korrektheit zu beweisen, um so beliebige „Muster“ an Schwierigkeiten (im Sinne einer vorgegebenen, endlichen Folge an Schwierigkeiten für die k-te Lösung) zu erzeugen und die Resultate zu verallgemeinern.

6.2 Schärfe der Voraussetzung

Wir konnten zwar in Abschnitt 5.5 einen Großteil des Resultats übertragen auf die Situation, dass $dom(A)$ nicht 1-paddable ist. Dennoch stellt sich die Frage, ob wir in Satz 5.3 nicht zu viel gefordert haben und ob unsere Voraussetzung nicht noch zu stark ist. Eine negative Antwort ließe sich dadurch liefern, dass wir zeigen, dass die gezeigten Ergebnisse genau dann funktionieren, wenn $dom(A)$ 1-paddable ist. Die Aussage, die dafür gezeigt werden müsste, lautet:

> Gegeben $A \in$ FNP. Falls für alle $B \in$ FNP ein $C \in$ FNP mit 1., 2. und 3. existiert, dann ist $dom(A)$ 1-paddable.
>
> 1. $dom(C) = dom(A)$

2. $C \equiv_m^p A$
3. $C_{[1]} \equiv_T^p B$

Es ist eher unwahrscheinlich, dass die Aussage gezeigt werden kann. Ein erster Anhaltspunkt, um die 1-Padding-Eigenschaft für A mit diesen Voraussetzungen zu zeigen, wäre eine geschickte Wahl von B und das Ausnutzen der Reduktion $B \leq_T^p C_{[1]}$, genauer der Orakelfragen, die während des Algorithmus gestellt werden. Dafür spricht, dass jegliche Frage $z = \langle x, l \rangle$, die der Algorithmus sich nicht selbst beantworten kann,[1] nicht nur eine gültige erste Lösung $l \in C(x)$, sondern auch eine positive Instanz $x \in dom(C)$ enthalten muss. Dabei wird die Instanz x vom Algorithmus in Polynomialzeit generiert und käme prinzipiell für die gesuchte 1-Padding-Funktion in Frage.

Jedoch spricht gegen diesen Ansatz, dass wir nicht unbedingt schnell solche Eingaben finden können, die im Reduktionsalgorithmus Fragen auslösen, und dass auch völlig unklar ist, wie wir einen Index zu einer aus einer Frage extrahierten Instanz x berechnen sollen – insbesondere angesichts dessen, dass dieselbe Frage auf verschiedenen Eingaben gestellt werden kann.

Da es aus den vorgenannten Gründen unwahrscheinlich wirkt, die Aussage beweisen zu können, kann man sich stattdessen auf die Frage konzentrieren, ob es eine natürliche Verschärfung der Aussage aus Satz 5.3 gibt, für die sich die Äquivalenz zeigen lässt.

6.3 Künstlichkeit der Konstruktion

An unserer Konstruktion lässt sich neben der Voraussetzung noch kritisieren, dass das entstandene Problem sehr künstlich ist. Basis der Kritik ist, dass die Schwierigkeit der 2. Lösung gerade auf denjenigen Instanzen beruht, bei denen die 1. Lösung einfach ist; sämtliche schwierigeren Instanzen – außerhalb von W_p – haben in unserer Konstruktion nie eine 2. Lösung. Zwar liefert die Konstruktion $\beta(A, B)$ einen „neuen Lösungsbegriff“ für das Problem A, jedoch fehlt es unseren zweiten Lösungen an praktischer Relevanz: Wenn sie überhaupt existieren, enthalten sie keine neue Information über eine Lösung zu A und beschäftigen sich stattdessen mit dem völlig anderen Problem B.

Genauso kann man kritisieren, dass der Lösungsbegriff durch unsere Konstruktion zu stark verändert wird. Schließlich kommen die ursprünglichen Beweise für die $x \in W_p$ in ganz $\beta(A, B)(x)$ gar nicht mehr vor, sondern befinden sich (bei der Konstruktion β) an einer völlig anderen Stelle, nämlich in $\beta(A, B)(x')$ für ein $x' \in W_{p_1}$.

Zu einem gewissen Anteil sind die Probleme sicherlich unserer maximal abstrakten Betrachtungsweise inhärent. Leider zeigt die Kritik auch noch keine konkrete neue Frage auf, sodass erst genauer geklärt werden müsste, welche zusätzlichen Eigenschaften an die Konstruktion gewünscht werden, bevor sich mit einer möglichen Lösung befasst werden kann.

6.4 Weitere Eigenschaften von 1-Paddability

Zu guter Letzt könnte noch der Begriff der 1-Paddability weiter bearbeitet und weitere Eigenschaften gesucht werden. Beispielsweise ließe sich noch besser herausarbeiten, welche Funktionsprobleme beweisbar 1-paddable sind. Wir äußern diesbezüglich eine Vermutung, nach der sich „beweisbar 1-paddable“ mit einem bestimmten Reduktionsbegriff übertragen lässt: Der in [YS03] eingeführte Reduktionsbegriff „$\leq_{ASP}$“ ist mindestens so stark wie „$\leq_T^p$“ und fordert neben einer Übersetzung der Frage an das Zielproblem zusätzlich eine Bijektion zwischen den Lösungen.

Vermutung 6.1. *Seien $f, g \in$ FNP. Dann folgt aus $f \equiv_{ASP} g$ und g beweisbar 1-paddable, dass f beweisbar 1-paddable.*

Die Vermutung begründet sich darin, dass sich die in g leicht erzeugbaren Instanzen mitsamt Lösungen und die Übersetzung, die die „$\leq_{ASP}$“-Reduktion in beide Richtungen liefert, für eine Konstruktion einer 1-Padding- und Beweisfunktion für f nutzen lassen. Falls die Vermutung gilt, folgt auch konkreter:

Vermutung 6.2. *Sei $f \in$ FNP. Dann folgt aus SAT $\leq_{ASP} f$, dass f beweisbar 1-paddable ist.*

Aus Beispiel 3.15 wissen wir bereits, dass SAT beweisbar 1-paddable ist. Nach [YS03] ist SAT vollständig bezüglich „$\leq_{ASP}$“ in FNP, also gilt immer $f \leq_{ASP}$ SAT. Die Eigenschaft „beweisbar 1-paddable“ würde sich also auf alle $\leq_{ASP}$-vollständigen FNP-Probleme übertragen.

[1] Fragen, die nicht diese Struktur haben, sind uninteressant, da sie trivialerweise mit „nein“ beantwortet werden können.

Offen ist auch noch der genaue Zusammenhang zwischen „1-paddable“ und „beweisbar 1-paddable“. In Abschnitt 3.5 wurde bereits besprochen, weshalb „$dom(A)$ 1-paddable $\not\Rightarrow$ A beweisbar 1-paddable“ schwer zu beweisen ist. Möglicherweise wird der Beweis durch Hinzunahme gängiger Voraussetzungen praktikabel.

Ebenfalls in Abschnitt 3.5 wurden bereits Indizien gegen die dem entgegenstehende Aussage „$dom(A)$ 1-paddable via $p \Rightarrow$ es ex. q mit A beweisbar 1-paddable via p, q“ vorgelegt. Noch zu betrachten ist die folgende, allgemeinere Aussage:

$$A \in \text{FNP und } dom(A) \text{ 1-paddable} \Rightarrow A \text{ beweisbar 1-paddable}$$

Der Unterschied besteht darin, dass hier keine bestimmte 1-Padding-Funktion festgehalten wird. Es könnte sein, dass zwar nicht zu jeder Padding-Funktion p die Beweise leicht zu finden sind, aber es doch immer noch eine andere Padding-Funktion p' gibt, die sich passend verhält, sodass $dom(A)$ beweisbar 1-paddable via p', q' für ein geeignetes q' ist. Mit anderen Worten: folgt aus „$dom(f)$ 1-paddable“ ebenfalls, dass es eine solche 1-Padding-Funktion p' gibt, die nur Instanzen mit einfachen Beweisen liefert? Hierfür bräuchte es eine dichte Teilmenge an Instanzen mit leichten Beweisen.

Literatur

[AFW92] K. Abrahamson, M. Fellows und C. Wilson. “Parallel self-reducibility”. In: *Proceedings ICCI ‘92: Fourth International Conference on Computing and Information*. Toronto, Ont., Canada: IEEE Comput. Soc. Press, 1992, S. 67–70. ISBN: 978-0-8186-2812-2. DOI: 10.1109/ICCI.1992.227703. URL: http://ieeexplore.ieee.org/document/227703/ (besucht am 18. 08. 2024).

[AR88] Eric W. Allender und Roy S. Rubinstein. “P-Printable Sets”. In: *SIAM Journal on Computing* 17.6 (Dez. 1988), S. 1193–1202. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/0217075. URL: http://epubs.siam.org/doi/10.1137/0217075 (besucht am 07. 09. 2024).

[Ber76] Leonard Berman. “On the structure of complete sets: Almost everywhere complexity and infinitely often speedup”. In: *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. Houston, TX, USA: IEEE, Okt. 1976, S. 76–80. DOI: 10.1109/SFCS.1976.22. URL: http://ieeexplore.ieee.org/document/4567890/ (besucht am 08. 09. 2024).

[BH77] L. Berman und J. Hartmanis. “On Isomorphisms and Density of NP and Other Complete Sets”. In: *SIAM Journal on Computing* 6.2 (Juni 1977). Publisher: Society for Industrial and Applied Mathematics, S. 305–322. ISSN: 0097-5397. DOI: 10.1137/0206023. URL: https://epubs.siam.org/doi/10.1137/0206023 (besucht am 16. 01. 2022).

[Del+20] Argyrios Deligkas u. a. “Exact and Approximate Algorithms for Computing a Second Hamiltonian Cycle”. In: *LIPIcs, Volume 170, MFCS 2020* 170 (2020), 27:1–27:13. ISSN: 1868-8969. DOI: 10.4230/LIPICS.MFCS.2020.27. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.MFCS.2020.27 (besucht am 19. 08. 2024).

[Din22] David Dingel. *Separation der relativierten Vermutungen SAT und TFNP*. Nov. 2022. URL: https://www1.pub.informatik.uni-wuerzburg.de/pub/theses/2023-dingel-bachelor.pdf (besucht am 22. 05. 2024).

[DK14] Ding-Zhu Du und Ker-I Ko. *Theory of Computational Complexity*. Somerset, United States: John Wiley & Sons, Incorporated, 2014. ISBN: 978-1-118-59303-5. URL: http://ebookcentral.proquest.com/lib/ub-wuerzburg/detail.action?docID=1695068 (besucht am 16. 01. 2022).

[Fen+93] Stephen Fenner u. a. “On using oracles that compute values”. In: Bd. 665. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, S. 398–407. ISBN: 978-3-540-56503-1 978-3-540-47574-3. DOI: 10.1007/3-540-56503-5_40. URL: http://link.springer.com/10.1007/3-540-56503-5_40 (besucht am 11. 06. 2024).

[GP17] Paul W. Goldberg und Christos H. Papadimitriou. “TFNP: An Update”. In: *Algorithms and Complexity*. Cham: Springer International Publishing, 2017, S. 3–9. ISBN: 978-3-319-57586-5. DOI: 10.1007/978-3-319-57586-5_1.

[GS88] Joachim Grollmann und Alan L. Selman. “Complexity Measures for Public-Key Cryptosystems”. In: *SIAM Journal on Computing* 17.2 (Apr. 1988), S. 309–335. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/0217018. URL: http://epubs.siam.org/doi/10.1137/0217018 (besucht am 05. 06. 2024).

[HH03] Lane A. Hemaspaandra und Harald Hempel. “P-immune sets with holes lack self-reducibility properties”. In: *Theoretical Computer Science* 302.1-3 (Juni 2003), S. 457–466. ISSN: 03043975. DOI: 10.1016/S0304-3975(02)00576-5. URL: https://linkinghub.elsevier.com/retrieve/pii/S0304397502005765 (besucht am 08. 09. 2024).

[HIS85] J. Hartmanis, N. Immerman und V. Sewelson. “Sparse sets in NP-P: EXPTIME versus NEXPTIME”. In: *Information and Control* 65.2-3 (Mai 1985), S. 158–181. ISSN: 00199958. DOI: 10.1016/S0019-9958(85)80004-8. URL: https://linkinghub.elsevier.com/retrieve/pii/S0019995885800048 (besucht am 07. 09. 2024).

[HY84] J. Hartmanis und Y. Yesha. “Computation times of NP sets of different densities”. In: *Theoretical Computer Science* 34.1 (Jan. 1984), S. 17–32. ISSN: 0304-3975. DOI: 10.1016/0304-3975(84)90111-7. URL: https://www.sciencedirect.com/science/article/pii/0304397584901117 (besucht am 09. 06. 2024).

[Jay08] Jennifer June Jaynes. *NP-completeness and another solution problem*. ProQuest Dissertations Publishing, 2008. ISBN: 978-0-549-50433-7. URL: https://www.proquest.com/docview/304828078 (besucht am 09. 09. 2024).

[Joh07] David S. Johnson. “The NP-completeness column: Finding needles in haystacks”. In: *ACM Transactions on Algorithms* 3.2 (Mai 2007), 24–es. ISSN: 1549-6325. DOI: 10.1145/1240233.1240247. URL: https://dl.acm.org/doi/10.1145/1240233.1240247 (besucht am 28. 05. 2024).

[JYP88] David S. Johnson, Mihalis Yannakakis und Christos H. Papadimitriou. “On generating all maximal independent sets”. In: *Information Processing Letters* 27.3 (März 1988), S. 119–123. ISSN: 0020-0190. DOI: 10.1016/0020-0190(88)90065-8. URL: https://www.sciencedirect.com/science/article/pii/0020019088900658 (besucht am 18. 06. 2024).

[KB91] Sanjeev N. Khadilkar und Somenath Biswas. “Padding, commitment and self-reducibility”. In: *Theoretical Computer Science* 81.2 (Apr. 1991), S. 189–199. ISSN: 0304-3975. DOI: 10.1016/0304-3975(91)90190-D. URL: https://www.sciencedirect.com/science/article/pii/030439759190190D (besucht am 28. 05. 2024).

[KUW88] Richard M. Karp, Eli Upfal und Avi Wigderson. “The complexity of parallel search”. In: *Journal of Computer and System Sciences* 36.2 (Apr. 1988), S. 225–253. ISSN: 00220000. DOI: 10.1016/0022-0000(88)90027-X. URL: https://linkinghub.elsevier.com/retrieve/pii/002200008890027X (besucht am 18. 08. 2024).

[OS84] Pekka Orponen und Uwe Schöning. “The structure of polynomial complexity cores”. In: *Mathematical Foundations of Computer Science 1984*. Bd. 176. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, S. 452–458. ISBN: 978-3-540-13372-8 978-3-540-38929-3. DOI: 10.1007/BFb0030328. URL: http://link.springer.com/10.1007/BFb0030328 (besucht am 18. 07. 2024).

[Pap94] Christos H. Papadimitriou. “On the complexity of the parity argument and other inefficient proofs of existence”. In: *Journal of Computer and System Sciences* 48.3 (Juni 1994), S. 498–532. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(05)80063-7. URL: https://www.sciencedirect.com/science/article/pii/S0022000005800637 (besucht am 28. 05. 2024).

[Pap95] Christos H. (1949-) Papadimitriou. *Computational complexity*. Reprint. with corr. Reading, Mass. [u.a.]: Addison-Wesley, 1995. ISBN: 978-0-201-53082-7.

[Ric08] Elaine Rich. *Automata, computability and complexity*. Upper Saddle River, NJ [u.a.]: Pearson/Prentice Hall, 2008. ISBN: 978-0-13-228806-4.

[Sel88] Alan L. Selman. “Natural Self-Reducible Sets”. In: *SIAM Journal on Computing* 17.5 (Okt. 1988), S. 989–996. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/0217062. URL: http://epubs.siam.org/doi/10.1137/0217062 (besucht am 04. 07. 2024).

[Sel92] Alan L. Selman. “A survey of one-way functions in complexity theory”. In: *Mathematical systems theory* 25.3 (Sep. 1992), S. 203–221. ISSN: 1433-0490. DOI: 10.1007/BF01374525. URL: https://doi.org/10.1007/BF01374525 (besucht am 21. 05. 2024).

[Sel94] Alan L. Selman. “A taxonomy of complexity classes of functions”. In: *Journal of Computer and System Sciences* 48.2 (Apr. 1994), S. 357–381. ISSN: 00220000. DOI: 10.1016/S0022-0000(05)80009-1. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022000005800091 (besucht am 05. 06. 2024).

[Sel96] A.L. Selman. “Much ado about functions”. In: *Proceedings of Computational Complexity (Formerly Structure in Complexity Theory)*. Philadelphia, PA, USA: IEEE Comput. Soc. Press, 1996, S. 198–212. ISBN: 978-0-8186-7386-3. DOI: 10.1109/CCC.1996.507682. URL: http://ieeexplore.ieee.org/document/507682/ (besucht am 05. 06. 2024).

[UN96] Nobuhisa Ueda und Tadaaki Nagao. *NP-completeness Results for NONOGRAM via Parsimonious Reductions*. 1996. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=1bb23460c7f0462d95832bb876ec2ee0e5bc46cf (besucht am 19. 08. 2024).

[VV86] L. G. Valiant und V. V. Vazirani. "NP is as easy as detecting unique solutions". In: *Theoretical Computer Science* 47 (Jan. 1986), S. 85–93. ISSN: 0304-3975. DOI: 10.1016/0304-3975(86)90135-0. URL: https://www.sciencedirect.com/science/article/pii/0304397586901350 (besucht am 10. 06. 2024).

[YS03] Takayuki Yato und Takahiro Seta. "Complexity and completeness of finding another solution and its application to puzzles". In: 86-A (Mai 2003), S. 1052–1060. ISSN: 0916-8508. URL: https://academic.timwylie.com/17CSCI4341/sudoku.pdf (besucht am 09. 09. 2024).

Erklärung zur Selbstständigkeit

Ich versichere, dass ich die vorgelegte Thesis selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Schweinfurt, den 30. September 2024

Fynn Godau

Erklärung zur Plagiatserkennung

Insoweit nach §27a Abs. 2 Satz 2 der aktuell gültigen ASPO erforderlich, versichere ich, dass ich mit der Überprüfung der Arbeit mittels Plagiatserkennungssoftware einverstanden bin und erteile die Einwilligung für einen etwaigen Datenupload für die Archivierung der Arbeit zum Zwecke der Erweiterung des Datenpools. Begleitende, identifizierende, personenbezogene Daten, die Rückschlüsse auf mich als Urheber zulassen, sind gemäß §27a Abs. 3 ASPO vor dem Einsatz einer Plagiatserkennungssoftware zu anonymisieren.

Schweinfurt, den 30. September 2024

Fynn Godau