First-order Reduktionen

von Johannes Kloibhofer

Vorbemerkungen:

- Diese Seminararbeit ist zum Großteil aus [1] übernommen. Hauptsächlich wird daraus Kapitel 3 behandelt. So werden auch jegliche Notationen und Bezeichnungen wie in [1] verwendet.
- Wie in [1] steht auch in dieser Arbeit n immer für die Größe des Universum $||\mathcal{A}||$. Gewöhnlich bezeichnet n in der Komplexitätstheorie die Länge des Inputs. Wichtig ist, dass beide Definitionen von n immer polynomiell voneinander abhängen.
- Wir betrachten immer Turingmaschinen mit einem Eingabeband und einem Arbeitsband. Die Turingmaschine kann dabei nur auf dem Arbeitsband das Band überschreiben. Eine Konfiguration einer Turingmaschine besteht aus dem Zustand der Maschine, dem Inhalt des Arbeitsbandes und der Position des Lesekopfes und des Schreibkopfes. Wir beschränken uns weiter auf Turingmaschinen mit einem eindeutigen Anfangs- und Endzustand.

1 FO $\subseteq$ L

In dieser Seminararbeit werden wir first-order Reduktionen betrachten. Obwohl dies sehr schwache Reduktionen sind, wird sich herausstellen, dass die meisten natürlichen vollständigen Probleme auch unter dieser Reduktion vollständig bleiben. Des weiteren sind viele Komplexitätsklassen unter first-order Reduktionen abgeschlossen. Um die Menge aller first-order Boolesche Abfragen mit den klassischen Komplexitätsklassen vergleichen zu können dient folgender Satz.

Satz 1.1. *Die Menge der first-order Booleschen Abfragen ist enthalten in der Menge der Booleschen Abfragen, die mit logarithmischen Platzverbrauch berechnet werden können, kurz FO $\subseteq$ L.*

Beweis. Sei $\sigma = \langle R_1^{a_1}, ..., R_r^{a_r}, c_1, ..., c_s \rangle$ und $\varphi \in \mathcal{L}(\sigma)$ bestimme eine first-order Boolesche Abfrage I_φ : STRUC$[\sigma] \to \{0,1\}$. Der Beweis ist vollbracht, wenn wir eine logspace Turingmaschine T angeben können, sodass für alle $\mathcal{A} \in$ STRUC$[\sigma]$ gilt:

$$\mathcal{A} \models \varphi \quad \iff \quad T(\text{bin}(\mathcal{A})) \downarrow \tag{1.1}$$

Wir konstruieren T induktiv nach dem Formelaufbau von φ:
Die Atomformeln von $\mathcal{L}(\sigma)$ sind die Eingaberelationen $R_i(p_1, ..., p_{a_i})$ und die numerischen Relationen $p_1 = p_2, p_1 \leq p_2$ und BIT(p_1, p_2), wobei die p_i Elemente von $\{c_1, ..., c_s, 0, 1, \max\}$ sind. Da die Beziehung $|\text{bin}(\mathcal{A})| = n^{a_1} + \cdots + n^{a_r} + s \cdot \lceil \log n \rceil$ gilt, kann eine logspace

Turingmaschine die Werte n und $\log n$ berechnen. Danach kann sie die Werte c_i bestimmen. So besteht zum Beispiel die Konstante c_2 aus dem $n^{a_1} + \cdots + n^{a_r} + \lceil \log n \rceil + 1$-ten bis zum $n^{a_1} + \cdots + n^{a_r} + 2 \cdot \lceil \log n \rceil$-ten Bit von $\text{bin}(\mathcal{A})$. Um nun eine Eingaberelation zu berechnen, gibt T das passende Bit der Eingabe aus. Um zum Beispiel $R_2(c_2, \max, 1)$ zu berechnen kopiert T zuerst die Werte $c_2, n-1, 1$ auf des Arbeitsband. Danach bewegt es den Lesekopf zur Position $n^{a_1} + 1$, dies ist der Beginn des Codes von R_2. Zum Schluss bewegt es den Lesekopf $n^2 \cdot c_2 + n \cdot (n-1) + 1$ Bits nach rechts. Dieses Bit ist nun genau dann 1, wenn $\mathcal{A} \models R_3(c_2, \max, 1)$. Bei dieser Konstruktion wurde nur $\mathcal{O}(\log n)$ Speicherplatz benötigt, um die Werte $c_2, n-1$ und 1 zu speichern. Weiters erkennt man relativ leicht, dass die numerischen Relationen von einer logspace Turingmaschine berechnet werden können.
Als nächstes habe die Formel φ die Form $\varphi = \varphi_1 \vee \varphi_2$, wobei es Turingmaschinen T_1 und T_2 gibt, sodass (1.1) erfüllt ist. Wir konstruieren T als Zusammensetzung von T_1 und T_2 mit einem neuen Endzustand, der erreicht wird, wenn T_1 oder T_2 den Endzustand erreicht hat. Dabei wird nur soviel Arbeitsspeicher benötigt, den T_1 und T_2 benötigen. Ähnlich geht man für $\wedge, \neg$ und $\rightarrow$ vor.
Habe nun φ die Form $\varphi = \exists x(\tilde{\varphi}(x))$. Nach Induktionsvoraussetzung existiert eine logspace Turingmaschine $\tilde{T}$, die die Abfrage $I_{\tilde{\varphi}(c)}$ berechnet. c ist dabei ein neues Konstantensymbol, das für die freie Variable x substituiert wird. Um die Abfrage I_φ zu berechnen konstruieren wir eine logspace Turingmaschine, die alle möglichen Werte von x durchläuft, diese für c substituiert und dann $\tilde{T}$ ausführt. T akzeptiert die Eingabe genau dann wenn eine dieser Ausführungen von $\tilde{T}$ akzeptiert wird. Es wird dabei nur $\log n$ Bits extra Speicher benötigt, um die möglichen Werte von x zu speichern. Für eine Formel $\forall x(\tilde{\varphi}(x))$ funktioniert die Konstruktion analog.
Somit haben wir für jede Formel eine Turingmaschine konstruiert, sodass (1.1) erfüllt ist und der Satz ist bewiesen.

□

Korollar 1.2. *Jede Menge von Booleschen Abfragen $\mathcal{S}$, die abgeschlossen bezüglich logspace Reduktion ist, ist auch bezüglich first-order Reduktion abgeschlossen.*

2 Vollständige Probleme

Das Ziel, das wir vor Augen haben, ist, von einer Menge von Booleschen Abfragen $\mathcal{L}$ und einer Komplexitätsklasse $\mathcal{C}$ zu zeigen, dass $\mathcal{L} = \mathcal{C}$ gilt. Die Vorgangsweise dafür werden dabei folgende Schritte sein:

1. Zeige $\mathcal{L} \subseteq \mathcal{C}$. Dies erreicht man, indem für jede Boolesche Abfrage in $\mathcal{L}$ eine Turingmaschine in der Komplexitätsklasse $\mathcal{C}$ konstruiert wird, die diese berechnet.

2. Konstruiere eine Boolesche Abfrage T, die vollständig für $\mathcal{C}$ bezüglich first-order Reduktion ist.

3. Zeige, dass $\mathcal{L}$ abgeschlossen bezüglich first-order Reduktion ist.

4. Zeige, dass $T \in \mathcal{L}$ gilt.

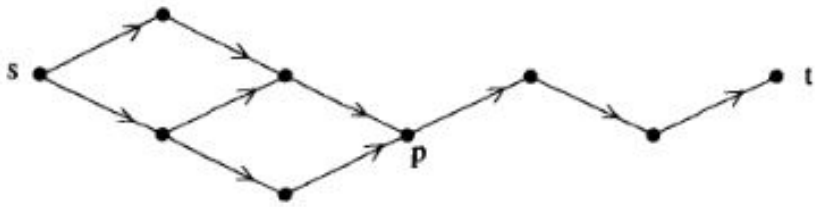

Abbildung 1: Ein Graph der in REACH ist, aber nicht in REACH_d

Für Schritt 3 lässt sich Korollar 1.2 verwenden, um dies zu vereinfachen. Wir werden uns als nächstes mit Schritt 2 beschäftigen und vollständige Probleme für einige wichtige Komplexitätsklassen konstruieren.

2.1 Vollständige Probleme für L und NL

In diesem Kapitel betrachten wir die Komplexitätsklassen L = DSPACE[log n] und NL = NSPACE[log n]. Zuerst werden Probleme vorgestellt die in diesen Klassen liegen und danach auch deren Vollständigkeit gezeigt.

Definition 2.1. *Definiere* REACH *als die Menge aller gerichteten Graphen, sodass es einen Pfad von s nach t gibt.* REACH_d *bezeichne die Teilmenge von* REACH *sodass es einen deterministischen Pfad von s nach t gibt. Jede Kante (u, v) auf dem Pfad ist also die einzige Kante, die von u wegführt.*

Proposition 2.2. REACH *ist in NL.*

Beweis. Folgender Algorithmus in NL entscheidet, ob der gegebene Graph in REACH ist. Es wird dabei nur Speicher benötigt um a und b zu benennen. Dies sind $\mathcal{O}(\log n)$ Bits. Beachte, dass es einen Berechnungspfad geben kann, der nicht terminiert, da wir uns aber nur für den benötigten Speicher interessieren ist dies kein Problem.

Algorithm 1

Input: $G = (V, E, s, t)$

```
1: b = s
2: while b ≠ t do
3:     a = b
4:     wähle nicht-deterministisch neues b ∈ V
5:     if ¬E(a, b) then
6:         return G ∉ REACH
7:     end if
8: end while
9: return G ∈ REACH
```

□

Satz 2.3. REACH *ist NL-vollständig bezüglich first-order Reduktion.*

Beweis. Sei $S \subseteq \text{STRUC}[\sigma]$ eine Boolesche Abfrage in NL. T sei eine logspace Turingmaschine, die S akzeptiert. Wir konstruieren eine first-order Reduktion $I : \text{STRUC}[\sigma] \to \text{STRUC}[\tau_g]$, sodass für alle $\mathcal{A} \in \text{STRUC}[\sigma]$ gilt:

$$T(\text{bin}(\mathcal{A})) \downarrow \quad \Longleftrightarrow \quad I(\mathcal{A}) \in \mathit{REACH}. \tag{2.1}$$

Die Idee von I ist relativ simpel: Wir konstruieren einen Graphen aus allen Konfigurationen von T, wobei eine Kante von einem Knoten u zu einem Knoten v genau dann führt wenn die Konfiguration v in einem Schritt von T aus der Konfiguration u hervorgeht. Nun terminiert T genau dann wenn es einen Pfad von der Initialkonfiguration zu einer Endkonfiguration[1] gibt.

Sei $\sigma = \langle R_1^{a_1}, ..., R_r^{a_r}, c_1, ..., c_s \rangle$ und $a = \max\{a_i | 1 \leq i \leq r\}$. Sei c so, dass T höchstens $c \log n$ Bits Arbeitsspeicher für Eingaben $\text{bin}(\mathcal{A})$ benötigt. n bezeichnet dabei $||\mathcal{A}||$. Nun definiere $k = 1 + a + c$. Wir beschreiben eine Konfiguration (ID) von T als ein k-Tupel von Variablen:

$$\text{ID} = (p, r_1, ..., r_a, w_1, ..., w_c)$$

Jede Variable repräsentiert dabei ein Element aus dem n-elementigen Universum von $\mathcal{A}$, ist also eine $\log n$-bit große Zahl. Die Variablen $w_1, ..., w_c$ codieren den Inhalt von T´s Arbeitsband. $r_1, ..., r_a$ codieren, wo in einer der Eingaberelationen sich der Lesekopf von T befindet. In der Variable p sind $\mathcal{O}(\log \log n)$ Bits weitere Information codiert:

- der Zustand von T (dies benötigt sogar nur $\mathcal{O}(1)$ Bits),
- in welcher Eingaberelation oder Konstante der Lesekopf gerade steht (als Zahl zwischen 1 und $r + s$ benötigt auch das $\mathcal{O}(1)$ Bits),
- die Position des Arbeitskopfes (dies ist eine Zahl zwischen 1 und $c \log n$, benötigt also $\mathcal{O}(\log \log n)$ Bits).

Wir nehmen an, dass n genügend groß ist, sodass diese Informationen in einer Variable p codiert werden können. Dabei soll der Zustand in den ersten, in welcher Eingaberelation der Lesekopf steht in den zweiten und die Position des Arbeitskopfes in den dritten $\log \log n$ Bits codiert werden.

Jetzt wird die gesuchte k-stellige first-order Abfrage I konstruiert und gezeigt dass sie (2.1) erfüllt:

$$I = \lambda_{\text{ID},\text{ID}'} \langle \textbf{true}, \varphi_T, \alpha, \omega \rangle,$$

wobei

- Die Relation **„true“** bedeutet, dass für das Universum von $I(\mathcal{A})$ gilt, dass $|I(\mathcal{A}) = |\mathcal{A}|^k$.

[1] Wir nennen jene Konfigurationen Endkonfigurationen, deren Zustand ein Endzustand ist.

- $\mathcal{A} \models \varphi_T(\mathrm{ID}, \mathrm{ID}')$ genau dann, wenn $(\mathrm{ID}, \mathrm{ID}')$ ein gültiger Schritt von T bei Eingabe $\mathrm{bin}(\mathcal{A})$ ist.
- $\mathcal{A} \models \alpha(\mathrm{ID}_i)$ genau dann, wenn ID_i die eindeutige Initialkonfiguration von T ist.
- $\mathcal{A} \models \omega(\mathrm{ID}_f)$ genau dann, wenn ID_f die eindeutige Endkonfiguration ist.

Die entsprechenden Formeln für α und ω sind

$$\begin{aligned} \alpha(x_1, ..., x_k) &\equiv x_1 = x_2 = \cdots = x_k = 0, \\ \omega(x_1, ..., x_k) &\equiv x_1 = x_2 = \cdots = x_k = \max. \end{aligned}$$

Dies klingt willkürlich, leistet aber das Gewünschte. α induziert, dass vor Beginn der Berechnung der Arbeitsspeicher leer ist, der Lese- sowie der Schreibkopf am Anfang des Bandes stehen und der Zustand von T der Anfangszustand ist. Die Formel für ω bedeutet, dass nach der Berechnung der Arbeitsspeicher mit 1-en gefüllt ist, der Lese- und der Schreibkopf sich am Ende des Bandes befinden und der Zustand von T der einzige Endzustand ist. Diese Bedingungen wirken auf den ersten Blick einschränkend, man kann sich jedoch überlegen, dass man jede Turingmaschine modifizieren kann, sodass sie genau eine Endkonfiguration hat, wo der Arbeitsspeicher mit 1-en gefüllt ist und der Lese- sowie der Schreibkopf sich am Ende des Bandes befinden. Dabei bleibt sie eine logspace Turingmaschine. Wir betrachten also O.B.d.A. nur solche Turingmaschinen.

Die Formel φ_T ist eine Disjunktion über alle möglichen Übergänge von T. Ein Übergang sieht dabei so aus: $(\langle q, b, w\rangle, \langle q', b', v, w'\rangle)$. Das bedeutet, dass im Zustand q, wenn sich der Lesekopf auf Bit b und der Schreibkopf auf Bit w befindet, T in Zustand q' gehen kann, den Lesekopf nach Bit b' bewegen kann, v auf das Arbeitsband schreiben kann und den Arbeitskopf nach Bit w' bewegen kann. Die Formel φ_T muss dabei von der Variablen p den Zustand decodieren, sowie welche Eingaberelation gerade betrachtet wird. Angenommen dies ist R_i, so ist Bit b genau dann 1, wenn $R_i(r_1, ..., r_{a_i})$ gilt. Genau so muss sie von p die Position s des Arbeitskopfes decodieren, Bit w ist dann genau 1, wenn $BIT(w_1 \cdots w_c, s)$ gilt. Damit φ_T diese Informationen von p decodieren kann, benötigt es die Zahl $l_2 = \lceil \log \log n \rceil$. Diese lässt sich aber durch eine first-order Formel beschreiben. Dafür definiere die Relation

$$\begin{aligned} \phi(n, l) &= [\mathrm{BIT}(n, l) \wedge \exists x < l(\mathrm{BIT}(n, x)) \wedge \forall x > l(\neg \mathrm{BIT}(n, x))] \vee \\ &\quad [\mathrm{BIT}(n, l+1) \wedge \forall x < l+1(\neg \mathrm{BIT}(n, x)) \wedge \forall x > l+1(\neg \mathrm{BIT}(n, x))], \end{aligned}$$

die beschreibt, dass $\lceil \log n \rceil = l$ gilt. Da $\lceil \log \lceil \log n \rceil \rceil = \lceil \log \log n \rceil$ gilt, folgt

$$l_2 = \lceil \log \log n \rceil \quad \Longleftrightarrow \quad \exists l_1 \, [\phi(n, l_1) \wedge \phi(l_1, l_2)].$$

Damit lässt sich φ_T als first-order Formel darstellen. Es folgt nun, dass T $\mathrm{bin}(\mathcal{A})$ akzeptiert, genau dann wenn es einen Weg in $I(\mathcal{A})$ von ID_i nach ID_f gibt, es gilt also (2.1).

□

Proposition 2.4. REACH_d *ist in L.*

Beweis. Wir modifizieren Algorithmus 1, sodass er deterministisch wird. Weiters benötigen wir einen Zähler um Kreise zu entdecken. Dies ist wichtig, damit der Algorithmus immer terminiert. Es wird wieder nur $\mathcal{O}(\log n)$ Bits Speicher benötigt um b, i und n zu benennen.

Algorithm 2

Input: $G = (V, E, s, t)$
1: $b = s$; $i = 0$; $n = |V|$
2: **while** $b \neq t \wedge i < n \wedge \exists! a(E(b, a))$ **do**
3: $\quad b = a$, sodass $E(b, a)$
4: $\quad i = i + 1$
5: **end while**
6: **if** $b = t$ **then**
7: $\quad$ **return** $G \in REACH$
8: **else**
9: $\quad$ **return** $G \notin REACH$
10: **end if**

□

Satz 2.5. REACH_d *ist L-vollständig bezüglich first-order Reduktion.*

Beweis. Der Beweis ist ähnlich zum Beweis von Satz 2.3. Für eine beliebige Boolesche Abfrage $S \subseteq \text{STRUC}[\sigma]$ aus L kopieren wir die Konstruktion. Der einzige Unterschied ist, dass T nun eine deterministische logspace Turingmaschine ist, die S berechnet. Für jedes $\mathcal{A}$ hat jeder Knoten in $I(\mathcal{A})$, da T deterministisch ist, höchstens eine wegführende Kante. Es folgt, dass $I(\mathcal{A})$ in REACH ist, genau dann, wenn es in REACH_d ist. Daher gilt

$$T(\text{bin}(\mathcal{A}))\downarrow \quad \Longleftrightarrow \quad I(\mathcal{A}) \in REACH_d.$$

□

2.2 Alternierende Turingmaschinen

Um ein vollständiges Problem für P zu finden, benötigen wir zuerst eine Verallgemeinerung von nichtdeterministischen Turingmaschinen. Damit eine nichtdeterministische Turingmaschine eine Eingabe akzeptiert benötigt es einen nichtdeterministischen Weg, der die Eingabe akzeptiert. Jedoch müssen alle Wege die Eingabe ablehnen, damit die Eingabe abgelehnt wird. Daraus ergibt sich eine Asymmetrie, die sich auch darin widerspiegelt, dass es noch immer ein ungelöstes Problem ist, ob nichtdeterministische Komplexitätsklassen unter Komplementbildung[2] abgeschlossen sind, ob z.B. $NP = co - NP$ gilt. Die vorherrschende Meinung ist, dass dies nicht der Fall ist.
Man kann sich nichtdeterministische Turingmaschinen auch so vorstellen, dass die Turingmaschine bei jeder nichtdeterministischen Auswahl eine OR-Verknüpfung einbaut, die 1

[2]Für eine Boolesche Abfrage $A \subseteq \text{STRUC}[\tau]$ ist das Komplement definiert als $\bar{A} = \text{STRUC}[\tau] - A$. Das Komplement einer Komplexitätsklasse $\mathcal{C}$ ist dann $co - \mathcal{C} = \{\bar{A} \mid A \in \mathcal{C}\}$

zurückgibt, falls eine der Berechnungen 1 zurückgibt. Erweitert man dies so, dass neben OR-Verknüpfungen auch AND-Verknüpfungen möglich sind, so erhält man den Begriff der alternierenden Turingmaschine:

Definition 2.6. *Eine alternierende Turingmaschine ist eine Turingmaschine, wo die Zustände aufgeteilt sind in existenzielle und universelle Zustände. Induktiv definieren wir: Eine alternierende Turingmaschine in einer gegebenen Konfiguration c* akzeptiert *genau dann wenn*

1. *c ist in einem Endzustand, der akzeptiert, oder*
2. *c ist in einem existentiellen Zustand und es existiert eine nächste Konfiguration c' die akzeptiert, oder*
3. *c ist in einem universellen Zustand, es existiert mindestens eine nächste Konfiguration und alle nächsten Konfigurationen akzeptieren.*

Eine alternierende Turingmaschine akzeptiert eine Eingabe x genau dann wenn sie in der Initialkonfiguration für x akzeptiert.

Dies ist eine Verallgemeinerung einer nichtdeterministischen Turingmaschine, da solche eine alternierende Turingmaschine ist, wo alle Zustände existentiell sind.

Wie bei den üblichen Komplexitätsklassen definieren wir die Komplexitätsklassen $ASPACE[s(n)]$ und $ATIME[t(n)]$ als die Mengen aller Booleschen Abfragen die von einer alternierenden Turingmaschine akzeptiert werden, die höchstens $\mathcal{O}(s(n))$ Bits am Arbeitsband, beziehungsweise höchstens $\mathcal{O}(t(n))$ Schritte in jedem Berechnungsweg für Eingaben der Größe n benötigt. Man schließt aus der Definition, dass die alternierenden Komplexitätsklassen unter Komplementbildung abgeschlossen sind.
Nun kommen wir zum für uns wichtigsten Satz über alternierende Turingmaschinen.

Satz 2.7. *([1, Theorem 2.25]) Für $s(n) \geq \log n$ und $t(n) \geq n$ gilt*

$$\bigcup_{k=1}^{\infty} ATIME[t(n)^k] = \bigcup_{k+1}^{\infty} DSPACE[t(n)^k],$$

$$ASPACE[s(n)] = \bigcup_{k=1}^{\infty} DTIME[k^{s(n)}].$$

Insbesondere gilt $ASPACE[\log n] = P$.

2.3 Vollständige Probleme für P

Mit den Vorkenntnissen aus dem letzten Unterkapitel können wir nun zwei Probleme definieren, die P-vollständig sind. Das erste wird dabei sehr ähnlich zu REACH sein.

Definition 2.8. *Ein* alternierender Graph $G = (V, E, A, s, t)$ *sei ein gerichteter Graph, der aus universellen und existenziellen Knoten besteht. $A \subseteq V$ sei die Menge der universellen Knoten. $\tau_{ag} = \langle E^2, A^1, s, t\rangle$ ist die Sprache der alternierenden Graphen.*
Wir definieren nun, was es bedeutet, dass ein Weg von x nach y führt. $P_a^G(x, y)$ sei die kleinste Relation, sodass:

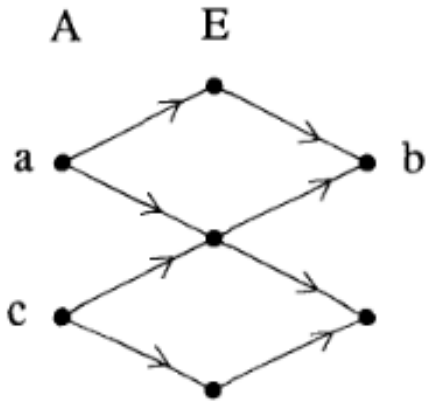

Abbildung 2: Ein alternierender Graph mit zwei universellen Knoten: a, c.

1. $P_a^G(x,x)$ *für alle* $x \in V$.
2. *Falls* x *existentiell ist und* $P_a^G(z,y)$ *für eine Kante* (x,z), *so gilt* $P_a^G(x,y)$.
3. *Falls* x *universell ist, mindestens eine Kante von* x *wegführt und* $P_a^G(z,y)$ *für alle Kanten* (x,z) *gilt, dann gilt auch* $P_a^G(x,y)$.

Abbildung 2 zeigt einen Graph mit zwei universellen Knoten und drei existentiellen Knoten. Es gilt dabei $P_a^G(a,b)$, jedoch nicht $P_a^G(c,b)$.

Betrachte nun

$$REACH_a \quad = \quad \{G \mid P_a^G(s,t)\}.$$

Proposition 2.9. REACH_a *ist in P.*

Beweis. Folgender Algorithmus entscheidet in quadratischer Zeit, ob ein alternierender Graph in REACH_a liegt.

Algorithm 3

Input: $G = (V, E, A, s, t)$

Markiere t; $Q = \{t\}$
while $Q \neq \varnothing$ **do**
    entferne beliebiges $x \in Q$
    for jeden unmarkierten Knoten y, sodass $E(y, x)$ **do**
        entferne Kante (y, x)
        if y ist existentiell oder hat keine wegführenden Kanten **then**
            markiere y; $Q = Q \cup y$
        end if
    end for
end while
if s ist markiert **then**
  return $G \in REACH_a$
else
  return $G \notin REACH_a$
end if

□

Satz 2.10. REACH_a *ist P-vollständig bezüglich first-order Reduktion.*

Beweis. Der Beweis ist ähnlich zum Beweis von Satz 2.3. Sei $S \subseteq \text{STRUC}[\sigma]$ eine Boolesche Abfrage, sodass $S \in P$ gilt. T sei eine alternierende logspace Turingmaschine, die S berechnet. Diese gibt es nach Satz 2.7. Wir konstruieren eine first-order Reduktion $I_a : \text{STRUC}[\sigma] \to \text{STRUC}[\tau_{ag}]$, sodass für alle $\mathcal{A} \in \text{STRUC}[\sigma]$ gilt:

$$T(\text{bin}(\mathcal{A})) \downarrow \quad \iff \quad I_a(\mathcal{A}) \in REACH_a. \tag{2.2}$$

Der einzige Unterschied zwischen I_a und I aus dem Beweis von Satz 2.3 ist, dass I_a die Relation A, welche die universellen Zustände von T identifiziert, beschreiben muss. Um dies zu bewerkstelligen nehmen wir o.B.d.A. an, dass die universellen Zustände genau die ungeraden Zustände sind. Weiters soll die Variable p in einer Konfiguration ID den Zustand von T in den niederwertigen Bits codieren. Daraus folgt, dass der Zustand einer ID genau dann universell ist, falls p ungerade ist. Dies ist der Fall wenn $\text{BIT}(p, 0)$ gilt. Definiere also

$$I_a = \lambda_{\text{ID},\text{ID}'}\langle \mathbf{true}, \varphi_T, \psi_A, \alpha, \omega \rangle, \qquad \psi_A = \text{BIT}(p, 0),$$

wobei φ_T, α und ω genau so wie im Beweis zu Satz 2.3 definiert sind. Die alternierende Turingmaschine T akzeptiert $\text{bin}(\mathcal{A})$ nun genau dann wenn $P_a^G(s, t)$ für die Anfangskonfiguration s und Endkonfiguration t gilt.Es folgt, dass (2.2) erfüllt ist.

□

Das nächste Problem, dass betrachtet wird behandelt Boolesche Schaltkreise. Es geht dabei um die Frage ob gegebene aussagenlogische Formel unter einer Bestimmten Variablenbelegung wahr ist.

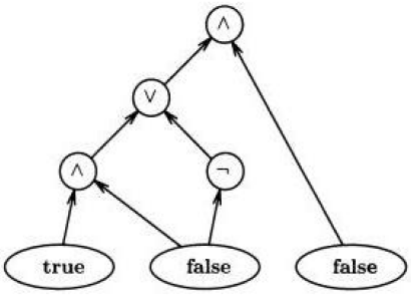

Abbildung 3: Ein Boolescher Schaltkreis, der nicht in CVP liegt.

Definition 2.11. *Ein Boolescher Schaltkreis ist ein gerichteter azyklischer Graph,*

$$C = (V, E, G_\wedge, G_\vee, G_\neg, I, r) \in STRUC[\tau_c], \quad \tau_C = \langle E^2, G^1_\wedge, G^1_\vee, G^1_\neg, I^1, r\rangle.$$

*Ein Knoten ist dabei ein Und-Gatter, wenn $G_\wedge(w)$ gilt, ein Oder-Gatter, wenn $G_\vee(w)$ gilt und ein Nicht-Gatter, wenn $G_\neg(w)$ gilt. Die Blätter v werden als **true** oder **false** interpretiert, wobei $I(v)$ bedeutet, dass v **true** ist. r bezeichnet die Wurzel des Graphen. Interpretiert man einen Booleschen Schaltkreis als aussagenlogische Formel, so sind die Blätter die Variablen und I bezeichnet die Variablenbelegung.*
*Das Circuit Value Problem (CVP) besteht aus den Booleschen Schaltkreisen, wo r als **true** ausgewertet wird. Das monotone Circuit Value Problem (MCVP) bezeichnet die Teilmenge von CVP, wo kein Nicht-Gatter vorkommt.*

Proposition 2.12. *CVP ist in P.*

Beweis. Wegen Satz 2.7 gilt $P = ASPACE[\log n]$. Folgender Algorithmus für eine alternierende Turingmaschine entscheidet, ob ein Knoten a als **true** oder **false** ausgewertet wird. Es wird dabei nur Speicherplatz benötigt, um a und b zu speichern. Da bei jedem rekursiven Aufruf a und b neu überschrieben werden können, wird also $\mathcal{O}(\log n)$ Speicher benötigt. EVAL(r) berechnet nun, ob gegebener Boolescher Schaltkreis in CVP liegt.

Algorithm 4 EVAL

Input: $a \in C$

1: **if** $I(a)$ **then return** *true*
2: **else if** $\nexists b\ E(b,a)$ **then return** *false*
3: **else if** $G_\neg(a)$ **then return** $\neg$EVAL(b), wobei $E(b,a)$ gilt
4: **else if** $G_\wedge(a)$ **then** wähle in universellem Zustand b, sodass $E(b,a)$ gilt
5: **else if** $G_\vee(a)$ **then** wähle in existentiellem Zustand b, sodass $E(b,a)$ gilt
6: **end if**
7: **return** EVAL(b)

□

Satz 2.13. REACH$_a$ *ist first-order reduzierbar auf CVP.*

Beweis. Sei $G = (V, E, A, s, t)$ ein alternierender Graph. Wir definieren eine 1-stellige first-order Abfrage $I_c = \text{STRUC}[\tau_{ag}] \to \text{STRUC}[\tau_c]$:

$$\begin{aligned} I_c = \lambda_{v,w}\langle &\textbf{true}, \psi_E, \psi_\wedge, \psi_\vee, \textbf{false}, \psi_I, \psi_r\rangle, \\ \psi_E(v,w) &\equiv E(w,v), \\ \psi_\wedge(v) &\equiv A(v), \\ \psi_\vee(v) &\equiv \neg A(v), \\ \psi_I(v) &\equiv v = t, \\ \psi_r(v) &\equiv v = s. \end{aligned}$$

Beachte, dass in Definition 2.8 die Relation $P_a^G(x, y)$, dass ein Weg von x nach y führt, induktiv von rechts nach links definiert wurde. Dadurch müssen wir den Graph $I_c(G)$ genau in die andere Richtung richten. $G_\wedge$ und $G_\vee$ entsprechen genau den universellen und existenziellen Knoten. Das einzige Blatt, dass als **true** interpretiert wird, ist t, alle anderen sind **false**. Nun führt genau dann ein Weg von s nach t, wenn s als **true** ausgewertet wird. Es folgt, dass

$$G \in REACH_a \quad \iff \quad I_c(G) \in CVP.$$

□

Korollar 2.14. *CVP ist P-vollständig.*

Beweis. Dies folgt direkt aus Proposition 2.12, Satz 2.13 und daraus, dass REACH_a P-vollständig ist. □

Beachte, dass Algorithmus 4 genau so auch für MCVP funktioniert und aus dem Beweis von Satz 2.13 folgt, dass REACH_a auch auf MCVP first-order reduzierbar ist. Daraus folgt, dass auch MCVP P-vollständig ist.

Literatur

[1] N. Immerman. *Descriptive Complexity.* Texts in Computer Science. Springer New York, 2012.

[2] C.H. Papadimitriou. *Computational Complexity.* Theoretical computer science. Addison-Wesley, 1994.