Bachelorarbeit

Algorithmen für endliche Automaten mit Ausgabe

Minimierung und Äquivalenztest

Michael Thomas
Matrikelnr.: 2117470

26.08.2004

Universität Hannover
FG Theoretische Informatik

Erklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Michael Thomas

Inhaltsverzeichnis

1. Einleitung

Endliche Automaten mit Ausgabe werden heutzutage in vielen Bereichen der Forschung und Wirtschaft eingesetzt. Dazu zählen unter anderem die Analyse und Verarbeitung von Sprachen und Texten sowie Gebiete des E-Commerce.

Dabei treten häufig Automaten mit mehreren hunderttausend Zuständen auf. Die Minimierung solcher Automaten war bisher nur anhand der klassischen Minimierung für deterministische endliche Automaten möglich, wobei das Paar (Eingabe, Ausgabe) als zusammengehörige Kantenmarkierung betrachtet wird. Allerdings liefert dieser Algorithmus keinen minimalen endlichen Automaten mit Ausgabe. Der Grund dafür liegt in der Möglichkeit ganze Zeichenketten als Ausgabe eines Übergangs anzugeben, anstatt bloß einzelne Zeichen. Auf diese Weise können Teile der Ausgabe eines Transducers T für ein Wort $w \in \Sigma^*$ entlang mancher Übergänge verschoben werden, ohne dass sich die Ausgabe des Transducers $f_T(w)$ ändert.

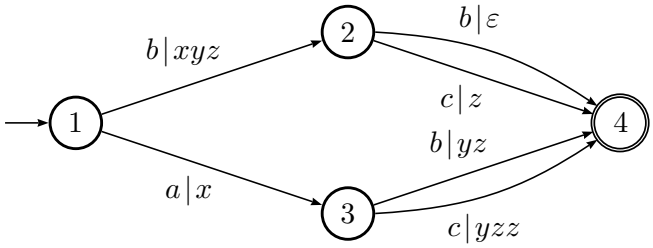


Abbildung 1.1.: Ein nicht-minimaler Transducer T

Abbildung 1.1 zum Beispiel zeigt einen nicht-minimalen Transducer, welcher nach der Automatenminimierung bereits minimal ist: Durch das Verschieben des Präfixes xy der Ausgaben aller Übergänge aus Zustand 3 an die linke Seite aller eingehenden Übergänge, werden die Zustände 2 und 3 äquivalent. Der minimale Transducer ist in Abbildung 4.6 gezeigt.

Hauptgrundlage meiner Arbeit ist der Artikel [Moh00]. Er lieferte als

erstes einen Algorithmus für die Konstruktion eines minimalen endlichen Automaten mit Ausgabe aus einem Gegebenen. Diese Bachelorarbeit soll dem Zweck dienen die Minimierung von endlichen Automaten mit Ausgabe nach [Moh00] zu erläutern, den Algorithmus in Java zu implementieren und darauf aufbauend einen Äquivalenztest zu entwickeln.

2. Grundlegende Definitionen

2.1. Notationen

Ich werde folgende Begriffe und Notationen für den Umgang mit Transducern und Zeichenketten gebrauchen: Sei

- G^T der Automat, der aus einem Automat G durch das umkehren jedes Übergangs hervorgeht,
- $Trans[q]$ die Menge der Übergänge, die einen Zustand q verlassen,
- $Trans^T[q]$ die Menge der Übergänge, die in einem Zustand q enden,
- $n(t)$ die Senke, $l_{\text{in}}(t)$ die Eingabe- und $l_{\text{out}}(t)$ die Ausgabemarkierung eines Übergangs, sowie $l(t) = (l_{\text{in}}(t), l_{\text{out}}(t))$ deren geordnetes Paar,
- ε der leere String des Ausgabealphabets.

Darüber hinaus sei für Worte $u, v \in \Delta^*$

- $uv, u \cdot v$ als die Konkatenation,
- $u \wedge v$ als das längste gemeinsame Präfix von u und v sowie
- $u^{-1}(uv)$ als die linke Truncation von u an (uv) definiert und
- durch $u \sqsubseteq v$ angegeben, dass u ein Präfix von v ist.

$u^{-1}w$ ist für den Fall, dass u kein Präfix von w ist, nicht definiert.

2.2. Automatenmodelle

Endliche Automaten mit Ausgabe sind Automaten, die das Konzept des deterministischen endlichen Automaten (DEA) insofern erweitern, als dass neben der binären Ausgabe ‚akzeptiere Wort' oder ‚akzeptiere Wort nicht'

auch eine Ausgabe über einem Ausgabealphabet liefern. Die Art der Ausgabe variiert hierbei:

Die sicherlich bekannten Moore-Machinen verknüpfen jeden Zustand mit einer Ausgabe, Mealy-Machinen hingegen verbinden die Ausgabe mit Zustandsübergängen im Automaten. Beide Modelle haben jedoch den Nachteil, dass für eine Eingabe w der Länge $n = |w|$ die Ausgabe in $n + 1$ für Moore bzw. in n für Mealy-Maschinen beschränkt ist, da die Ausgabefunktion pro Übergang nur auf ein Zeichen des Ausgabealphabets abbildet.

Um dieses Hindernis zu umgehen werde ich in dieser Bachelorarbeit den verallgemeinerten Begriff des endlichen Transducers verwenden. Diese Automaten besitzen eine Ausgabefunktion $\sigma : Q \times \Sigma \to \Delta^*$, anhand derer Ausgabelängen von $c \cdot n$ möglich sind, wobei $c \in \mathbb{N}$ konstant. Mealy- und Moore-Machinen sind Spezialfälle von Transducern, bei denen für alle Übergänge die Ausgabe von σ die Länge 1 hat. Vergleicht man das im folgenden vorgestelle Verfahren zur Minimierung von endlichen Transducern mit dem bekannten Verfahren zur Minimierung von Moore-Maschinen (z. B. in [Spe03]), so stellt man fest, dass für Transducer mit $|\lambda| = 1$ und $\forall q \in Q : \forall t, t' \in Trans[q] : (\sigma(q,t) = \sigma(q,t') \wedge |\sigma(q,t)| = 1)$ beide Verfahren äquivalent sind, also hier dargelegte Verfahren eine Verallgemeinerung darstellt.

In Anlehnung an die Definition von endlichen Automaten in [VW02] seien nun also endliche Automaten mit Ausgabe definiert:

Definition 2.1 *Ein endlicher Automat mit Ausgabe (endlicher Transducer, Transduktor) ist ein 8-Tupel $T = (Q, \Sigma, \Delta, \delta, \sigma, \lambda, q_0, F)$, wobei*

- *das 5-Tupel $(Q, \Sigma, \delta, q_0, F)$ ein deterministischer endlicher Automat,*
- *Δ ein endliches Ausgabealphabet,*
- *λ eine Zeichenkette, die links an die Ausgabe gehängt wird,*
- *σ eine Funktion $\sigma : Q \times \Sigma \to \Delta^*$ ist.*

Weiterhin bezeichnen $\hat{\delta} : Q \times \Sigma^* \to Q$ und $\hat{\sigma} : Q \times \Sigma^* \to \Delta^*$ die Fortsetzungen von δ bzw. σ definiert durch

$$\forall q \in Q, w \in \Sigma^*, a \in \Sigma : \hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a) \ , \ \hat{\delta}(q, \varepsilon) = q$$

und

$$\forall q \in Q, w \in \Sigma^*, a \in \Sigma : \hat{\sigma}(q, wa) = \sigma(\hat{\sigma}(q, w), a) \ , \ \hat{\sigma}(q, \varepsilon) = \varepsilon$$

Da die Funktionen δ und σ durch ihre Fortsetzungen ersetzen werden können, sei im weiteren Verlaufe durch δ und σ die jeweilige Fortsetzung bezeichnet.

Ein Transducer T nach obiger Definition akzeptiert ein Eingabewort $w \in \Sigma^*$, wenn er sich nach Abarbeitung von w in einem Endzustand befindet, d. h. $\delta(q_0, w) \in F$. Die von T akzeptierte Menge an Wörtern sei bezeichnet mit $L(T)$. Für $w \in L(T)$ kann die Ausgabe eines Transducers kann als eine Übersetzung zwischen zwei Mengen Σ^* und Δ^* angesehen werden. Diese Übersetzung lässt sich mathematisch durch eine Funktion ausdrücken:

Definition 2.2 *Sei T ein endlicher Transducer nach Definition 2.1, dann berechnet T eine sequentielle Funktion $f_T : \Sigma^* \to \Delta^*, f_T(w) = \lambda \cdot \sigma(q_0, w)$. f_T ist definiert für alle $w \in L(T)$. $Dom(f_T) = \{w \in \Sigma^* | w \in L(T)\}$ bezeichne die Menge aller von T akzeptierten Wörter und*

$$D(f_T) = \{u \in \Sigma^* | \exists v \in \Sigma^* : (uv) \in Dom(f_T)\}$$

die Menge aller Präfixe von $Dom(f_T)$.

Diese Definitionen 2.1 und 2.2 sind angelehnt an die Definition von [Moh00]. Allerdings werde ich hier nicht die vereinfachende Annahme $\lambda = \varepsilon$ treffen und die Algorithmen für den allgemeinen Fall vorstellen.

Darüber hinaus betrachte ich nur Transducer bzw. Automaten, deren Zustände alle vom Startzustand aus erreichbar sind und alle mindestens einen Pfad in einen Endzustand besitzen. Ein Algorithmus, zur Gewinnung solcher Transducer/Automaten aus einem gegebenen befindet sich in Anhang A.1.

Weiterhin kann ein Transducer in einen Eingabautomaten und einen Ausgabeautomaten zerlegt werden:

Definition 2.3 *Der Eingabeautomat zu einem Transducer T sei definiert als der DEA $T_{in} = (Q, \Sigma, \delta, q_0, F)$. Der Ausgabeautomat sei definiert als der nichtdeterministischer endliche Automat mit ε-Übergängen (NEA) $T_{out} = (Q, \Delta, \sigma', q_0, F)$, wobei $\sigma' : Q \times \Delta^* \to Q$ definiert ist durch*

$$\forall q, q' \in Q, u \in \Delta^* : \sigma'(q, u) = q' \Leftrightarrow \exists a \in \Sigma : \delta(q, a) = q' \wedge \sigma(q, a) = u$$

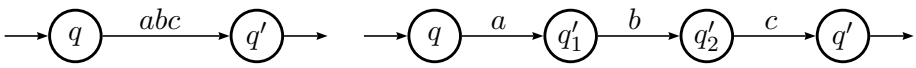
Der Ausgabeautomat T_{out} ist also der Automat, der die gleichen Zustände und Übergänge hat wie T und dessen Eingabe die von T_{out} getätigte Ausgabe ohne das Präfix λ ist.

Es ist zu beachten, dass die an dieser Stelle verwendete Definition eines NEA von den üblichen Definitionen abweicht, da die Funktion σ Zeichenketten an den Zustandsübergängen zulässt und λ ein initiales Präfix darstellt, welches von jedem von T_{out} akzeptierten Wort abgeschnitten wird. Trotzdem sind die beiden erwähnten Modelle eines NFA äquivalent:

Ein NFA, der Zeichenketten an den Übergängen zulässt, kann in einen klassischen NFA umgewandelt werden, indem für jeden Übergang $\sigma(q, w) = q'$ mit $q, q' \in Q, w \in \Delta^*, n := |w| \geq 2, w = w_1 w_2 \dots w_n$ eine Kette von neuen Zuständen $q'_1, \dots, q'_{n-1}$ Zuständen mit Ein- und Ausgangsgrad 1 eingefügt wird (siehe Abbildung 2.1). Dazu wird $\sigma(q, w) = q'$ wie folgt ersetzt:

$$\sigma(q, w_1) = q'_1, \quad \sigma(q'_{i-1}, w_i) = q_i \text{ für } 1 < i < n, \quad \sigma(q'_{n-1}, w_n) = q'$$

Danach kann λ durch das Hinzufügen eines neues Startzustands q_0^* mit $\delta(q_0^*, \lambda) = q_0$ eliminiert werden.



(a) (b)

Abbildung 2.1.: (a) Ein Übergang mit Markierung aus Δ^*, (b) der äquivalente Übergang eines NFA mit Markierungen aus Δ.

Die Eigenschaften der Minimalität eines endlichen Transducer T bzw. der Äquivalenz zweier endlicher Transducer T und T' lassen sich nun –analog zu den betreffenden Eigenschaften endlicher Automaten in [VW02]– über die berechnete sequentielle Funktion f_T definieren:

Definition 2.4 *Sei T ein endlicher Transducer, der die Funktion f_T berechnet. Dann ist T ein minimaler Transducer bezüglich f_T gdw. alle Transducer T' mit $f_{T'} \equiv f_T$ weniger oder gleich viele Zustände besitzen.*

Definition 2.5 *Seien T und T' endliche Transducer. Dann sind T und T' äquivalent genau dann, wenn $f_T \equiv f_{T'}$ ist, d. h.*

$$\forall w \in \Sigma^* : \begin{cases} w \in Dom(f_T) \Leftrightarrow w \in Dom(f_{T'}) \\ w \in Dom(f_T) \Rightarrow f_T(w) = f_{T'}(w) \end{cases}$$

3. Charakterisierung minimaler Transducer

Wie bereits in der Einleitung erwähnt, liefert das für DEAs bekannte Minimierungsverfahren für Transducer keinen minimalen Transducer, da es auf dem Paar (Eingabe, Ausgabe) als Kantenmarkierung arbeiten muss. So besteht die Möglichkeit, dass Ausgaben an Zustandsübergängen in Richtung des Startzustandes verschoben werden können und so einen Zustand ‚überflüssig' machen. Folglich benötigt man eine feinere Äquivalenzrelation als die des Automaten-Minimierungsalgorithmus, um einen minimalen endlichen Transducer zu definieren.

Definition 3.1 *Sei $\sim_f$ eine 2-stellige Relation auf $D(f_T)$ definiert als*

$$\forall (u,v) \in D(f_T) \times D(f_T) : u \sim_f v \Leftrightarrow \exists (u',v') \in \Delta^* \times \Delta^* \; \forall w \in \Sigma^* : \begin{cases} uw \in Dom(f_T) \Leftrightarrow vw \in Dom(f_T) \\ uw \in Dom(f_T) \Rightarrow u'^{-1} f_T(uw) = v'^{-1} f_T(vw)) \end{cases}$$

Bemerkung: Die umständliche Definition über $u' = f_T(u)$ und $v' = f_T(v)$ ist nötig, da die Funktion f_T nur für Worte $w \in Dom(f_T)$ definiert ist.

R_f definiert eine Äquivalenzrelation auf $D(f_T)$: $\sim_f$ ist transitiv, symmetrisch und reflexiv. u,v sind äquivalent bezüglich $\sim_f$ gdw. nach Einlesen von u bzw. v die gleichen Suffixe von Worten aus $L(T)$ akzeptiert und bei deren Erkennung gleichen Ausgaben über Δ^* getätigt werden. Die Äquivalenzklasse zu einem Präfix $u \in D(f_T)$ sei mit $[u]_f = \{v \in \Sigma^* | u \sim_f v\}$ notiert.

Lemma 3.1 *Wenn f_T eine Funktion ist, die durch einen Transducer T berechnet wird, dann ist die Anzahl der Äquivalenzklassen von $\sim_f$ kleiner oder gleich der Anzahl der Zustände von T.*

Beweis Sei $T = (Q, \Sigma, \Delta, \lambda, \delta, \sigma, q_0, F)$ ein Transducer und $u' = \lambda \cdot \sigma(q_0, u)$, $v' = \lambda \cdot \sigma(q_0, v)$ gewählt. Dann fällt das Paar Präfixe (u, v) unter $\sim_f$ in eine Äquivalenzklasse, wenn T sich nach deren Einlesen im selben

Zustand $\delta(q_0, u) = \delta(q_0, v)$ befindet: In $q' := \delta(q_0, u) = \delta(q_0, v)$ gilt für alle Fortsetzungen $w \in \Sigma^*$, die in einen Endzustand führen:

$$\delta(q', w) \in F \Rightarrow uw \in Dom(f_T) \Leftrightarrow vw \in Dom(f_T)$$

Weiterhin ist durch den Determinismus des Eingabeautomaten von T die Ausgabe nach dem Lesen von u oder v identisch, da $\sigma(q', w)$ nur von w und dem erreichten Zustand q' abhängt. Folglich muß die Anzahl der Äquivalenzklassen von $\sim_f$ größer oder gleich der Anzahl der Zustände von T sein. □

Anhand der Relation $\sim_f$ kann nun die Existenz eines minimalen Transducers hergeleitet werden:

Satz 3.1 *Für jede von einem Transducer T berechenbare Funktion f_T existiert ein minmaler Transducer $T_{\min}$, der genau so viele Zustände besitzt wie $\sim_f$ Äquivalenzklassen.*

Beweis Sei f_T eine Funktion, die von einem Transducer T berechnet wird, und $g : \Sigma^* \to \Delta^*$ eine Funktion, die das längste gemeinsame Präfix der Ausgaben aller Fortsetzungen $w \in \Sigma^*$ nach Einlesen von u bezeichnet, die den Transducer in einen Endzustand überführen:

$$g(u) = \begin{cases} \bigwedge_{\substack{w \in \Sigma^* \\ uw \in Dom(f_T)}} f_T(uw), \text{ für } u \in D(f_T) \\ \varepsilon, \text{ für } u \in \Sigma^* \setminus D(f_T) \end{cases}$$

Dann kann ein minimaler Transducer $T_{\min} = (Q, \Sigma, \Delta, \delta, \sigma, \lambda, q_0, F)$ wie folgt definiert werden:

- $Q = \{[u]_f | u \in D(f_T)\}$
- $F = \{[u]_f | u \in Dom(f_T)\}$
- $\lambda = g(\varepsilon)$
- $q_0 = [\varepsilon]_f$
- $\forall u \in \Sigma^*, \forall a \in \Sigma : u \in D(f_T), ua \in D(f_T) \Rightarrow \delta([u]_f, a) = [ua]_f$
- $\forall u \in \Sigma^*, \forall a \in \Sigma : u \in D(f_T), ua \in D(f_T) \Rightarrow \sigma([u]_f, a) = (g(u))^{-1} g(ua)$

$T_{\min}$ enthält also einen Zustand pro Äquivalenzklasse von $\sim_f$. Die Menge der Zustände ist endlich nach Lemma 3.1. Endzustände sind genau die Zustände, deren Äquivalenzklassen Worte aus $L(T)$ enthalten. Der Startzustand ist gegeben durch die Äquivalenzklasse der Zustände, die kein Zeichen der Eingabe verarbeitet haben, und $\lambda = g(\varepsilon)$ das längste gemeinsame Präfix aller möglichen Ausgaben des Transducers $T_{\min}$.

Darüber hinaus ist die Übergangsfunktion δ konsistent definiert: Für $u, v \in D(f_T)$, $u \sim_f v$ befindet sich der Transducer $T_{\min}$ nach Abarbeitung von u bzw. v im selben Zustand. Da der Eingabeautomat des Transducers T deterministisch ist, kann es auch nur einen Übergang mit der Eingabemarkierung a aus $[u]_f$ geben, also sind sowohl der Folgezustand als auch die Ausgabe identisch. Es folgt:

$$\forall u, v \in D(f_T) : u \sim_f v \Rightarrow ua \sim_f va$$

Betrachtet man nun die Ausgabefunktion σ, so ist diese ebenfalls unabhängig von der Wahl $u \in [v]_f, (u, v \in \Sigma^*)$: Nach Definition von g folgt $\forall a \in \Sigma : g(u) \sqsubseteq g(ua)$. Sind nun $u, v \in D(f_T)$ äquivalent unter $\sim_f$, dann folgt zunächst

$$\begin{aligned} &\exists (u', v') \in \Delta^* \times \Delta^* : \forall w \in \Sigma^* : \\ &\qquad uw \in Dom(f_T) \quad \Rightarrow \quad u'^{-1} f_T(uw) = v'^{-1} f_T(vw) \\ &\qquad\qquad\qquad\quad \Rightarrow \quad u'^{-1} g(u) = v'^{-1} g(v) \end{aligned}$$

und für $a \in \Sigma, ua \in D(f_T)$

$$\begin{aligned} &\exists (u', v') \in \Delta^* \times \Delta^* : \forall w \in \Sigma^* : \\ &\qquad u(aw) \in Dom(f_T) \quad \Rightarrow \quad u'^{-1} f_T(u(aw)) = v'^{-1} f_T(v(aw)) \\ &\qquad\qquad\qquad\quad \Rightarrow \quad u'^{-1} g(ua) = v'^{-1} g(va) \end{aligned}$$

Formt man diese beiden Gleichungen nach $g(u)$ bzw. $g(ua)$ um, so erhält man

$$g(u) = u'v'^{-1} g(v) \quad \text{und} \quad g(ua) = u'v'^{-1} g(va)$$

Damit folgt

$$\begin{aligned} \sigma([u]_f, a) &= (g(u))^{-1} g(ua) = (u'v'^{-1} g(v))^{-1} u'v'^{-1} g(va) = (g(v))^{-1} g(va) \\ &= \sigma([v]_f, a) \end{aligned}$$

Folglich ist auch für σ die Wahl des Repräsentanten $u \in [v]_f$ egal.

Zeige nun, das $T_{\min}$ die Funktion f_T berechnet: Die von $T_{\min}$ akzeptierten Eingaben $w \in \Sigma^*$ sind alle Eingaben, unter denen der Eingabeautomat in einem Endzustand anhält:

$$\delta([\varepsilon]_f, w) \in F \Leftrightarrow [w]_f \in F \Leftrightarrow w \in Dom(f_T)$$

Die bei der Abarbeitung eines $u \in D(f_T), v \in \Sigma$ mit $uv \in D(f_T)$ wird dabei die Ausgabe

$$\begin{aligned}\sigma([\varepsilon]_f, uv) &= \sigma([\varepsilon]_f, u)\sigma([u]_f, v) \\ &= (g(\varepsilon))^{-1} g(u)(g(u))^{-1} g(uv) \\ &= (g(\varepsilon))^{-1} g(uv)\end{aligned}$$

getätigt. Für $u \in Dom(f_T)$ folgt somit für die gesamte Ausgabe des Transducers $T_{\min}$ zu

$$\lambda \cdot \sigma([\varepsilon]_f, u) = \lambda \cdot (g(\varepsilon))^{-1} \cdot g(u) = \lambda \cdot \lambda^{-1} \cdot \bigwedge_{\substack{w \in \Sigma^*: \\ uw \in Dom(f_T)}} f_T(uw) = f_T(u)$$

Die Gleichung ist erfüllt, da zum einen $f_T(u) \in Dom(f_T)$ für $w = \varepsilon$ und das längste gemeinsame Präfix einer Wortmenge nicht länger sein kann als das kürzeste in ihr enthaltene Wort. Und zum anderen, weil λ gemeinsames Präfix aller Ausgaben $f_T(w)$ mit $w \in Dom(f_T)$ ist:

$$\lambda \sqsubseteq \bigwedge_{\substack{w \in \Sigma^*: \\ uw \in Dom(f_T)}} f_T(uw)$$

$T_{\min}$ definiert also einen Transducer, der mit einer minimalen Anzahl Zuständen f_T berechnet. □

Die Idee nach [Moh00] ist es nun, einen Transducer in einem Vorbereitungsschritt so zu modifizieren, dass das Ergebnis einer Automatenminimierung einen minimalen Transducer nach Satz 3.1 liefert. Dieser Vorbereitungsschritt, genannt die Quasi-Determinierung, wird im folgenden Kapitel beschrieben.

4. Minimierung

4.1. Quasi-Determinierung

4.1.1. Idee

Die Quasi-Determinierung ist der erste Schritt auf dem Weg zur Minimierung von endlichen Transducern. Die Idee des Verfahrens ist es, einen Transducer T durch die Quasi-Determinierung so vorzubereiten, dass nach einer anschließenden Automatenminimierung für das Ergebnis $T_{\min}$ eine Assoziation der Äquivalenzklassen von $\sim_f$ mit den Zuständen von $T_{\min}$ möglich wird, also $T_{\min}$ ein minimaler Transducer nach Satz 3.1 zu T ist.

Um dies zu erreichen, versucht die Quasi-Determinierung, die Ausgabe unter Zustandsübergängen soweit wie möglich in Richtung des Startzustands zu verschieben, ohne die Topologie des Transducer dabei zu verändern (Die Topologie des Transducers bezeichne hier die Topologie des zugehörigen gerichteten Graphen, siehe Definition 4.1). Dies führt zu einem Transducer, der für äquivalente Zeichenketten u, v unter $\sim_f$ die Zeichenketten $u', v' \in \Delta^*$ zu $u' = \sigma(q_0, u)$ bzw. $v' = \sigma(q_0, v)$ bestimmt. Damit lässt sich die zweite Bedingung der Definition 3.1 auf identische Ausgaben auf dem Teilpfad von $\delta(q_0, u)$ bzw. $\delta(q_0, v)$ in einen Endzustand beschränken.

Da die Quasi-Determinierung nur auf den Ausgabemarkierungen eines Transducers T arbeitet, reicht an dieser Stelle die Betrachtung des Ausgabeautomaten T_{out}.

4.1.2. Zusammenhang von Transducern und Graphen

Da ich in diesem Kapitel Begriffe der Graphentheorie für die Erklärung der Quasi-Determinierung benutzen werde, sei der Zusammenhang von endlichen Transducern und Graphen hier kurz näher erläutert: Transducer oder Automaten werden vielfach als Graphen dargestellt. Dabei wird jeder Zustand als ein Knoten und jeder Übergang als eine Kante interpretiert. Die Markierung des Übergangs wird einfach an die Kante geschrieben.

Definition 4.1 *Sei $T = (Q, \Sigma, \Delta, \delta, \sigma, \lambda, q_0, F)$ ein endlicher Transducer. Der zu T gehörende Graph G_T ist der gerichtete Multigraph $G_T = (V, E)$, wobei $V = Q$ und E die Menge der Kanten des Multigraphen ist:*

$$E = \{(u, v) \in Q \times Q \mid \exists a \in \Sigma : \delta(u, a) = v\}$$

Durch diese Interpretation ist es nun möglich, graphentheoretische Begriffe auf Automaten bzw. Transducer zu übertragen: Ein Transducer T sei als zyklisch angegeben gdw. G_T zyklisch ist, T besitzt eine topologische Sortierung gdw. G_T eine topologische Sortierung besitzt, etc. Weiterhin sei mit E die Menge der Zustandsübergänge von T wie oben definiert bezeichnet. Außerdem werden die Begriffe der starken Zusammenhangskomponente und des Komponentengraphen wie folgt definiert:

Definition 4.2 *Sei $G = (V, E)$ ein gerichteter Multigraph. Eine starke Zusammenhangskomponente $\mathcal{C} \subseteq V$ ist die Teilmenge, in der für je 2 Zustände $u, v \in \mathcal{C}$ ein Pfad von u nach v und ein Pfad von v nach u existiert.*

Starke Zusammenhangskomponenten repräsentieren Äquivalenzklassen auf Graphen/Transducern unter der Relation $\sim_c$ mit

$$u \sim_c v \Leftrightarrow \exists \text{ ein Pfad von } u \text{ nach } v \text{ und von } v \text{ nach } u$$

Außerdem sei die Äquivalenzklasse eines $v \in V$ unter $\sim_c$ notiert mit $[v]_c$.

Definition 4.3 *Sei $G = (V, E)$ ein gerichteter Multigraph. Dann ist der Komponentengraph $G^c = (V^c, E^c)$ zu G definiert durch*

- $V^c = \{[v]_c \mid v \in V\}$ *und*
- $E^c = \{([u]_c, [v]_c) \mid (u, v) \in E\}$.

Ein Verfahren zur Berechnung von G^c aus G ist im Anhang A.2.2 beschrieben und benötigt Zeit $O(|V| + |E|)$.

Satz 4.1 *Ein Komponentengraph ist azyklisch.*

Beweis Sei $G^c = (V^c, E^c)$ ein zyklischer Komponentengraph. Dann existiert ein Knoten $u \in V^c$, der einen Pfad mit Länge ≥ 1 auf sich selbst besitzt. Da jeder Knoten v in diesem Pfad von u zu erreichen ist und stets auch ein Pfad von v nach u existiert, liegen alle Knoten v und u in der selben starken Zusammenhangskomponente. Somit folgt ein Widerspruch zu Definition 4.3. □

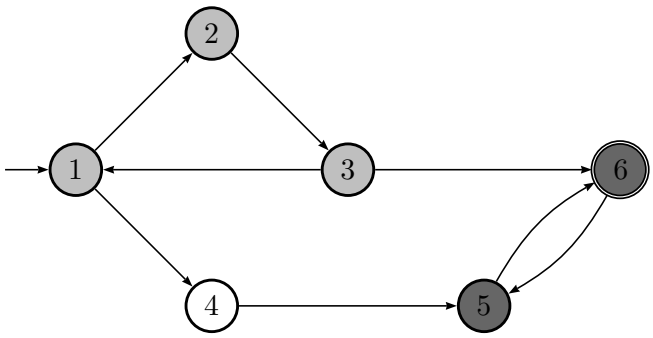


Abbildung 4.1.: Ein gerichteter Graph. Zustände einer starken Zusammenhangskomponente besitzen die gleiche Füllfarbe.

4.1.3. Berechnung

Nachdem der Zusammenhang von Graphen und Transducern eindeutig definiert ist, werde ich auf die Quasi-Determinierung an sich eingehen: Um die eingangs erwähnte Verschiebung von Ausgabemarkierungen in Richtung des Startzustands fassen zu können, sei zunächst die Funktion P definiert:

Definition 4.4 *Sei $P : Q \to \Delta^*$ eine Funktion, die für jeden Zustand $q \in Q$ das längste gemeinsame Präfix der Ausgaben aller Pfade von q in einen Endzustand angibt:*

$$\forall q \in F : P(q) = \varepsilon, \quad \forall q \in Q \setminus F : P(q) = \bigwedge_{t \in Trans(q)} l_{out}(t) P(n(t))$$

P ist wohldefiniert, da jeder Zustand mindestens einen Pfad in einen Endzustand besitzt. Anhand dieser Funktion lässt sich nun ein NFA definieren, der die gleiche Topologie wie T_{out} besitzt, aber für alle $q \in Q$: $P(q) = \varepsilon$ gilt:

Definition 4.5 *Zu einem NFA G sei $p(G)$ derjenige Automat, der die gleiche Topologie wie G besitzt und dessen Zustandsübergänge definiert sind als*

$$\forall q \in Q : \forall t \in Trans(q) : l(t)_{out_{p(G)}} = (P(q))^{-1} \, l(t)_{out_G} P(n(t))$$

mit $\lambda_{p(G)} = \lambda_G P(q_0)$. Dabei gebe der zusätzliche Index G bzw. $p(G)$ die Zugehörigkeit zum NEA G bzw. $p(G)$ an.

Sei T' nun der Transducer, der durch die Quasi-Determinierung aus einem Transducer $T = (Q, \Sigma, \Delta, \delta, \sigma, \lambda, q_0, F)$ berechnet wird. T' sei gegeben durch den Eingabeautomaten von T und den Automaten $p(T_{\text{out}})$, welcher sich nach Definition 4.5 aus T_{out} errechnet. Diese Definition ist konsistent ist, da unter der Quasi-Determinierung die Topologie von T (bis auf die Ausgabemarkierungen) invariant ist. Darüber hinaus gilt

Lemma 4.1 *Es ist $f_{T'} \equiv f_T$.*

Beweis Sei $T' = (Q, \Sigma, \Delta, \delta, \sigma', \lambda', q_0, F)$. Da T' ein Wort $w \in \Sigma^*$ akzeptiert, wenn T es akzeptiert, gilt $Dom(f_{T'}) = Dom(f_T)$. Für $u, w \in \Sigma^*, \delta(q_0, u) = q$ und $uw \in D(f_T)$ folgt nun

$$\sigma'(q, w) = (P(q))^{-1}\sigma(q, w)P(\delta(q, w))$$

Somit ist die getätigte Ausgabe für ein $w \in D(f_T)$:

$$\lambda'\sigma'(q_0, w) = \lambda P(q_0)(P(q_0))^{-1}\sigma(q_0, w)P(\delta(q_0, w)) = \lambda\sigma(q_0, w)P(\delta(q_0, w))$$

Da für $\delta(q_0, w) \in F : P(\delta(q_0, w)) = \varepsilon$ ist für $w \in Dom(f_T)$ die getätigte Ausgabe identisch und somit $f_T \equiv f_{T'}$. □

Satz 4.2 *Sei $T' = (Q, \Sigma, \Delta, \delta, \sigma', \lambda', q_0, F)$ ein quasi-determinierter Transducer. T' besitzt die in der Einleitung geforderte Eigenschaft, dass für $u, v \in \Sigma^*, u \sim_f v$ die Zeichenketten $u', v' \in \Delta^*$ aus Definition 3.1 gegeben sind durch*

$$u' = \lambda'\sigma'(q_0, u) \ \wedge \ v' = \lambda'\sigma'(q_0, v)$$

Beweis Sei $u' = \lambda'\sigma'(q_0, u)$ und $v' = \lambda'\sigma'(q_0, v)$ gewählt. Sei $q_u = \delta(q_0, u)$ und $q_v = \delta(q_0, v)$. Dann folgt aus der zweiten Bedingung der Definition 3.1, dass für alle $w \in \Sigma^*$:

$$\begin{aligned}
\delta(q_u, w) \in F \quad &\Rightarrow \quad u'^{-1}f(uw) = v'^{-1}f(vw) \\
&\Leftrightarrow \quad (\lambda'\sigma'(q_0, u))^{-1}\lambda'\sigma'(q_0, uw) = (\lambda'\sigma'(q_0, v))^{-1}\lambda'\sigma'(q_0, vw) \\
&\Leftrightarrow \quad \sigma'(q_u, w) = \sigma'(q_v, w)
\end{aligned}$$

Folglich ist die Wahl von u' und v' wie oben legitim, falls

$$\forall w \in \Sigma^* : \delta(q_u, w) \in F \Rightarrow \sigma'(q_u, w) = \sigma'(q_v, w)$$

gilt. Zeige durch Widerspruch, dass dies für T' der Fall ist:

Annahme: Sei $u \sim_f v$ und $w = w_1 \dots w_n \in \Sigma^*, w_0 = \varepsilon$ so gewählt, dass die Ausgaben $\sigma'(q_u, w)$ und $\sigma'(q_v, w)$ nicht übereinstimmen. Dann existiert eine Position $i \in [0, n-1]$ mit

$$\sigma'(\delta(q_u, w_0 \dots w_i), w_{i+1}) \neq \sigma'(\delta(q_v, w_0 \dots w_i), w_{i+1})$$

Sei o. B. d. A. $|\sigma'(\delta(q_u, w_0 \dots w_i), w_{i+1})| \geq |\sigma'(\delta(q_v, w_0 \dots w_i), w_{i+1})|$. Dann existiert ein anderer Übergang aus q_u, dessen Ausgabe mit einem Zeichen $a \in \Sigma$ beginnt und ungleich dem ersten Zeichen von $\sigma'(\delta(q_u, w_0 \dots w_i), w_{i+1})$ ist:

Da $|\sigma'(\delta(q_u, w_0 \dots w_i), w_{i+1})| \geq |\sigma'(\delta(q_v, w_0 \dots w_i), w_{i+1})|$, und für alle $q \in Q : P(q) = \varepsilon$, muss $|\sigma'(\delta(q_u, w_0 \dots w_i), w_{i+1})| \geq 1$ sein. Da die Ausgabe somit nicht mehr ε sein kann, muss ein weiterer Übergang existieren, mit einem Zeichen $a \neq w_{i+1}$. Dieser Übergang ist selbst Teil eines Pfades in mindestens einen Endzustand, da der Transducer getrimmt ist; sei das Wort $w' = ax, x \in \Sigma^*$ so gewählt, dass $\delta(q_u, w_0 \dots w_i w') \in F$.

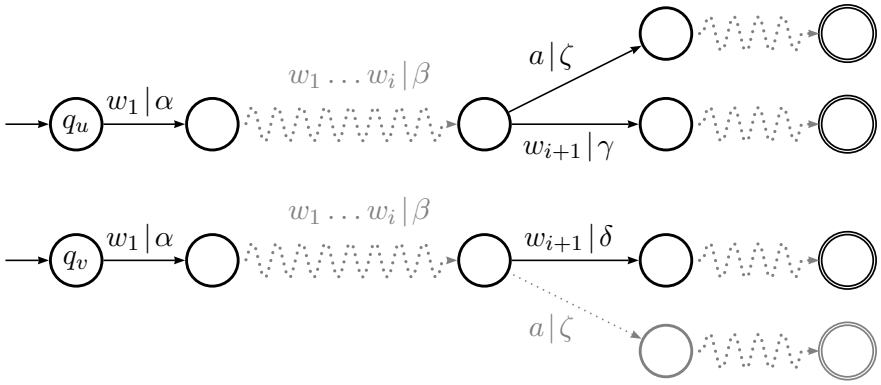


Abbildung 4.2.: $(\alpha, \beta, \gamma, \zeta \in \Delta^*)$

Im Zustand $\delta(q_v, w_0 \dots w_i w')$ muss ebenfalls ein entsprechender Pfad mit $\delta(q_v, w_0 \dots w_i w') \in F$ existieren. Andernfalls würde aus $\delta(q_u, w_0 \dots w_i w') \in F$ und $\delta(q_u, w_0 \dots w_i w') \notin F$ ein Widerspruch zu $u \sim_f v$ folgen.

In diesem Fall lässt sich jedoch kein $u', v' \in \Delta^*$ finden, so dass für alle $y \in \Sigma^*$ die Ausgaben $u'^{-1} f(uy) = v'^{-1} f(vy)$ identisch sind: Wählt man u' und v' wie oben oder gar kürzer, so ist $u'^{-1} f(uw) \neq v'^{-1} f(vw)$. Wählt man hingegen u' und v' so lang, dass $u'^{-1} f(uw) = v'^{-1} f(vw)$, so ist

$u'^{-1}f(uw_0 \dots w_i w')$ nicht definiert, da u'^{-1} die Ausgabe $g(uw_0 \dots w_i w_{i+1})$ mit einschließt und für $\sigma'(\delta(q_u, w_0 \dots w_i), w_{i+1}) \neq \sigma'(\delta(q_u, w_0 \dots w_i), a)$ u'^{-1} kein Präfix von $f(uw_0 \dots w_i w')$ sein kann. Somit folgt ein Widerspruch zu $u \sim_f v$.

Folglich gilt für T', dass $\forall w \in \Sigma^* : \sigma'(q_u, w) = \sigma'(q_v, w)$ für $u' = \lambda'\sigma'(q_0, u)$ und $v' = \lambda'\sigma'(q_0, v)$ gilt. □

Zunächst sei jedoch die genaue Berechnung von $p(T_{\text{out}})$ beschrieben: Ist $P(q)$ für alle Zustände $q \in Q$ bekannt, so kann die Berechnung von $p(T_{\text{out}})$ in Zeit $O(|E|)$ erfolgen. Die Berechnung von $P(q)$ allerdings auf direktem Wege nur für Transducer möglich, die eine topologische Sortierung der Zustände besitzen, also wenn G_T azyklisch ist.

Azyklischer Fall Für den Fall eines azyklischen Transducers kann der Automat $p(T_{\text{out}})$ rekursiv aus T_{out} berechnet werden, indem man von den Endzuständen ausgehend entgegen einer topologischen Sortierung vorgeht. Ein Verfahren zur Ermittlung einer topologischen Sortierung ist im Anhang A.2.1 geschildert. Es ist angelehnt an den Algorithmus von [Tar74] und benötigt Zeit $O(|V| + |E|)$ auf dem gerichteten Multigraphen (Q, E), wobei E die Menge der Zustandsübergänge angibt.

Zyklischer Fall Für den allgemeineren Fall eines zyklischen Transducers hingegen existiert keine topologische Sortierung der Zustände, da z. B. für zwei Knoten $u, v \in V$ mit $(u, v) \in E$ und $(v, u) \in E$ in einer Sortierung sowohl u vor v, als auch v vor u stehen müsste. Es existiert jedoch eine topologische Sortierung der starken Zusammenhangskomponenten. Da für azyklische Transducer jeder Zustand eine eigene Zusammenhangskomponente bildet, lässt sich dieser Fall auf den allgemeinen Fall eines zyklischen Transducers zurückführen und das im folgenden erläuterte Verfahren auf alle Transducer anwenden:

Sei T ein Transducer mit zugehörigen Multigraphen G_T. Dann ist G_T^c ein gerichteter azyklischer Graph. G_T^c erhält man durch die Zerlegung von G_T in seine starken Zusammenhangskomponenten wie in Anhang A.2.2 beschreiben. Aus G_T^c kann nun $p(T_{\text{out}})$ berechnet werden, indem man auf G_T^c wie im azyklischen Fall entgegen einer topologischen Sortierung vorgeht. Jedoch muss zusätzlich für jede starke Zusammenhangskomponente ein Gleichungssystem gelöst werden, dass innerhalb einer solchen Kom-

ponente die gemeinsamen Präfixe der Ausgabemarkierungen entgegen der Übergangsrichtungen verschiebt.

Sei $\mathcal{C}$ eine beliebig gewählte starke Zusammenhangskomponente und $q \in \mathcal{C}$:

$$I[q] = \{t \in Trans[q] \mid n(t) \in \mathcal{C}\}$$
$$O[q] = \{t \in Trans[q] \mid n(t) \notin \mathcal{C}\}$$

Dann lautet dieses Gleichungssystem für $\mathcal{C}$:

$$(X_q)_{q\in\mathcal{C}} = \begin{cases} \varepsilon \text{ für } q \in F \\ \bigwedge_{t\in I[q]} l_{\text{out}}(t) X_{n(t)} \text{ für } q \notin F \wedge O[q] = \emptyset \\ \left(\bigwedge_{t\in I[q]} l_{\text{out}}(t) X_{n(t)}\right) \wedge \left(\bigwedge_{t\in O[q]} l_{\text{out}}(t)\right) \text{ für } q \notin F \wedge O[q] \neq \emptyset \end{cases} \tag{4.1}$$

Lemma 4.2 *Die Lösung des Gleichungssystems* (4.1) *ist gegeben durch* $X_q = P(q)$.

Beweis Da der Komponentengraph endlich und azyklisch ist, existiert eine topologische Sortierung $(\mathcal{C}_{\sigma(1)}, \ldots, \mathcal{C}_{\sigma(n)})$ der Zusammenhangskomponenten $\mathcal{C}_i$, $1 \leq i \leq n$.

Für die letzte starke Zusammenhangskomponente dieser Sortierung $\mathcal{C}_{\sigma(n)}$ gilt dann $\forall q \in \mathcal{C}_{\sigma(n)} : O[q] = \emptyset$, also

$$(X_q)_{q\in\mathcal{C}_{\sigma(n)}} = \begin{cases} \varepsilon \text{ für } q \in F \\ \bigwedge_{t\in Trans[q]} l_{\text{out}}(t) X_{n(t)} \text{ für } q \notin F \end{cases}$$

Folglich ist für $q \in \mathcal{C}_{\sigma(n)} : X_q = P(q)$.

Geht man nun entgegen der topologischen Sortierung der starken Zusammenhangskomponenten vor, gilt für jede starke Zusammenhangskomponente $\mathcal{C}_{\sigma(i)}$, dass alle Übergänge deren Zielzustand nicht mehr in $\mathcal{C}_{\sigma(i)}$ liegt, in starke Zusammenhangskomponenten $\mathcal{C}_{\sigma(j)}$ mit $j > i$ führen, d. h. die in der topologischen Sortierung hinter $\mathcal{C}_{\sigma(i)}$ stehen (siehe Abbildung 4.3). Durch die gewählte Vorgehensweise ist gewährleistet, dass die hinter $\mathcal{C}_{\sigma(i)}$ stehenden starken Zusammenhangskomponenten $\mathcal{C}_{\sigma(j)}$ mit $i < j$ bereits korrekt verarbeitet worden, die Übergangsmarkierungen der $\mathcal{C}_{\sigma(j)}$

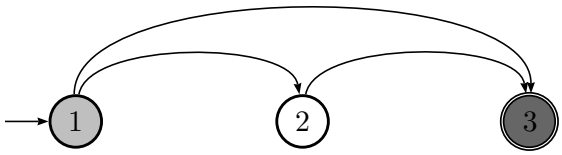


Abbildung 4.3.: Der Komponentengraph zu Abbildung 4.1. Eine topologische Sortierung der starken Zusammenhangskomponenten ist $(1, 2, 3)$.

also bereits denen von $p(T_{\text{out}})$ entsprechen. Nach Definition 4.5 gilt also für $q \in \mathcal{C}_{\sigma(j)} : X_q = \varepsilon$ oder

$$\forall u \in \mathcal{C}_{\sigma(i)} : \forall t \in O[q] : \exists j > i : n(t) \in \mathcal{C}_{\sigma(j)} \wedge X_{n(t)} = \varepsilon$$

Somit ergibt sich das Gleichungssystem zu

$$\begin{aligned}(X_q)_{q\in\mathcal{C}_{\sigma(i)}} &= \begin{cases} \varepsilon \text{ für } q \in F \\ \bigwedge_{t\in I[q]} l_{\text{out}}(t) X_{n(t)} \text{ für } q \notin F \wedge O[q] = \emptyset \\ \left(\bigwedge_{t\in I[q]} l_{\text{out}}(t) X_{n(t)}\right) \wedge \left(\bigwedge_{t\in O[q]} l_{\text{out}}(t)\right) \text{ für } q \notin F \wedge O[q] \neq \emptyset \end{cases} \\ &= \begin{cases} \varepsilon \text{ für } q \in F \\ \bigwedge_{t\in Trans[q]} l_{\text{out}}(t) X_{n(t)} \text{ für } q \notin F \end{cases} \\ &= (P(q))_{q\in\mathcal{C}_{\sigma(i)}}\end{aligned}$$

□

Um nun auch $\mathcal{C}$ korrekt zu verarbeiten und die gemeinsamen Präfixe von ausgehenden Kanten eines Zustands zu eliminieren, erstellt man nun aus $(X_q)_{q\in\mathcal{C}}$ ein neues Gleichungssystem $(Y_q)_{q\in\mathcal{C}}$, das für ein $q_0 \in \mathcal{C}$ das längste gemeinsame Präfix aller ausgehenden Kanten hinter die Markierungen der eingehenden Kanten verschiebt (siehe Abbildung 4.4). Sei dazu

$$\pi_q = \begin{cases} \bigwedge_{t\in Trans[q]} l_{\text{out}}(t) \text{ für } q \notin F \\ \varepsilon, \text{ sonst.} \end{cases} \tag{4.2}$$

und $q_0 \in \mathcal{C}_{\sigma(i)}$ mit $P(q_0) \neq \varepsilon$ beliebig aber fest gewählt. Dann ergibt sich das neue Gleichungssystem zu

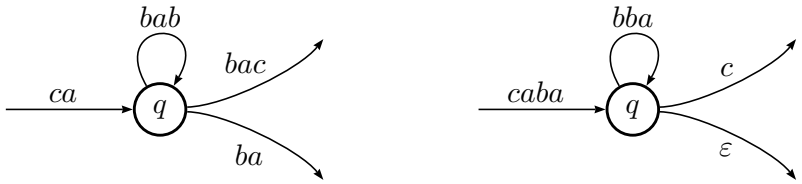
$$(Y_q)_{q\in\mathcal{C}} = \begin{cases} \pi_q^{-1} X_q \text{ für } q = q_0 \\ X_q \text{ für } q \neq q_0 \end{cases} \tag{4.3}$$

$$= \begin{cases} \varepsilon \text{ für } q \in F \\ \left(\bigwedge_{t\in I[q]} l'(t) Y_{n(t)}\right) \text{ für } q \notin F \wedge O[q] = \emptyset \\ \left(\bigwedge_{t\in I[q]} l'(t) X_{n(t)}\right) \wedge \left(\bigwedge_{t\in O[q]} l'(t)\right) \text{ für } q \notin F \wedge O[q] \neq \emptyset \end{cases} \tag{4.4}$$

mit

$$\begin{array}{lcl} \forall t \notin Trans[q_0] \cup Trans^T[q_0] & : & l'(t) = l_{\text{out}}(t) \\ \forall t \in Trans[q_0] \cap Trans^T[q_0] & : & l'(t) = \pi_{q_0}^{-1} l_{\text{out}}(t) \pi_{q_0} \\ \forall t \in Trans[q_0] \setminus Trans^T[q_0] & : & l'(t) = \pi_{q_0}^{-1} l_{\text{out}}(t) \\ \forall t \in Trans^T[q_0] \setminus Trans[q_0] & : & l'(t) = l_{\text{out}}(t) \pi_{q_0} \end{array} \tag{4.5}$$

Das Gleichungssystem (4.4) besitzt offensichtlich die Lösung $\forall q \in \mathcal{C}, q \neq q_0 : Y_q = X_q,\ Y_{q_0} = \pi_{q_0} X_{q_0}$ genau dann, wenn $(X_q)_{q\in\mathcal{C}}$ eine Lösung von (4.1) ist.



(a) (b)

Abbildung 4.4.: Ausgabemarkierungen (a) $l(t)$ und (b) $l'(t)$ für $t \in Trans[q]$ bzw. (a) vor und (b) nach dem Verschieben des längsten gemeinsamen Präfixes ba.

Lemma 4.3 *Sei $\mathcal{C}$ eine starke Zusammenhangskomponente. Dann gilt für die Gleichungssysteme $(X_q)_{q\in\mathcal{C}}$ und $(Y_q)_{q\in\mathcal{C}}$:*

$$\sum_{q\in\mathcal{C}} |Y_q| < \sum_{q\in\mathcal{C}} |X_q|$$

Beweis Da nach Wahl von q_0 $\pi_{q_0} \neq \varepsilon$, gilt $\sum_{q\in\mathcal{C}} |Y_q| = \sum_{q\in\mathcal{C},q\neq q_0} |Y_q| + |Y_{q_0}| = \sum_{q\in\mathcal{C},q\neq q_0} |X_q| + |\pi_{q_0}^{-1} X_{q_0}| < \sum_{q\in\mathcal{C},q\neq q_0} |X_q| + |X_{q_0}| = \sum_{q\in\mathcal{C}} |X_q|$ □

Die Konstruktion eines neuen Gleichungssystems $(Y_q)_{q\in\mathcal{C}}$ aus einem alten lässt sich iterativ wiederholen, solange ein $q \in \mathcal{C} \setminus F$ existiert, für dass $\pi_q \neq \varepsilon$ ist. Die Anzahl der möglichen Konstruktionen ist dabei nach Lemma 4.3 beschränkt durch $\sum_{q\in\mathcal{C}} |X_q|$.

Lemma 4.4 *Seien $(Z_q)_{q\in\mathcal{C}}$ die Variablen des letzten Gleichungssystems, dass durch den iterativen Rekonstruktionsprozess aus dem Gleichungssystem* (4.1) *hervorgeht. Dann ist $(Z_q)_{q\in\mathcal{C}} = \varepsilon$.*

Beweis Für das letzte Gleichungssystem des iterativen Prozesses gilt $\forall q \in \mathcal{C} : \pi_q = \varepsilon$. Somit folgt für $\forall q \in \mathcal{C} \setminus F : \pi_q = \bigwedge_{t\in Trans[q]} l_{\text{out}}(t) = \varepsilon$. Aus $\bigwedge_{t\in Trans[q]} l_{\text{out}}(t) = \varepsilon$ folgt nun, dass nach Definition des Gleichungssystems (4.1) für $(Z_q)_{q\in\mathcal{C}} = \varepsilon$. □

Bezeichne $(\pi_q^1, \ldots, \pi_q^{k_q})$ die geordnete Folge der an einem Zustand q im Laufe der Iterationen abgeschnittenen π_q und $(\pi_{n(t)}^1, \ldots, \pi_{n(t)}^{k_{n(t)}})$ selbiges für die Senke eines Übergangs $t \in Trans[q]$. Dann ergibt sich aus der Lösung des ersten Gleichungssystems $\forall q \in \mathcal{C} : X_q = P(q)$, der Lösung des letzten Gleichungssystems $\forall q \in \mathcal{C} : Z_q = \varepsilon$ und dem iterativen Vorgehen, dass

$$\forall q \in \mathcal{C} : P(q) = (\pi_q^{k_q})^{-1} \cdots (\pi_q^1)^{-1}$$

ist. Weiterhin folgt anhand der Gleichungen (4.5) für die Übergangsmarkierungen $l'(t)$ (mit $t \in Trans[q], q \in \mathcal{C}$) des letzten Gleichungssystems

$$\begin{aligned} l'(t) &= (\pi_q^{k_q})^{-1} \cdots (\pi_q^1)^{-1} l_{\text{out}}(t) \pi_{n(t)}^1 \cdots \pi_{n(t)}^{k_{n(t)}} \\ &= P(q)^{-1} l_{\text{out}}(t) P(n(t)) \end{aligned}$$

Folglich entsteht durch das beschriebene Verfahren der iterativen Rekonstruktion der Gleichungssysteme nach 4.3 ein Gleichungssystem, dessen $l'(t)$ genau die Übergangsmarkierungen des Automaten $p(T_{\text{out}})$ sind. Die Berechnung von $p(T_{\text{out}})$ ist als Pseudocode in Listing 4.1 angegeben.

Es ist anzumerken, dass der Algorithmus zur korrekten Berechnung von $p(T_{\text{out}})$ ebenfalls $\lambda' = \lambda P(q_0)$ bestimmen muss. Zur Vereinfachung

des Listings 4.1, sei diese Konkatenation implizit vorgenommen, indem man annimmt λ sie die Markierung einer eingehenden Kante von q_0.

In Listing 4.1 bezeichne

- C eine sortierte Liste der starken Zusammenhangskomponenten von G_T und $C[i]$, $1 \leq i \leq n$ das i. Element dieser Liste
- Q eine Queue von Zuständen,
- $N[q]$ die Anzahl an Übergängen eines Zustands q, deren Markierung ε sind und
- F eine Menge, die alle Zustände q enthält, für die die Transformation abgeschlossen ist d. h. für die $\pi_q = \varepsilon$ bis zum Ende des Algorithmus gilt.

Die in Zeile 10 aufgerufene Funktion $\mathrm{lcp}(T_{\mathrm{out}}, q)$

- berechnet den Rückgabewert π_q nach Gleichung (4.2),
- modifiziert alle ausgehenden Markierungen $l_{\mathrm{out}}(t)$ $(t \in Trans[q])$ durch Abschneiden von π_q auf der linken Seite,
- erhöht $N[q]$ falls nötig und
- fügt gegebenenfalls q zu F hinzu.

Ein Zustand q wird hierbei zur Menge F hinzugefügt genau dann, wenn q ein Endzustand ist oder wenn nach einem Aufruf der Funktion $N[q] = 0$ ist. In diesem Fall existiert eine Nichtübereinstimmung der Markierungen $l_{\mathrm{out}}(t)$ mit $t \in Trans[q]$; da sich in der weiteren Ausführung des Algorithmus aus Listing 4.1 nur noch die Suffixe der Ausgabemarkierungen ändern, kann $\pi_q = \varepsilon$ nicht mehr verändert werden.

Die Wahl des $q \in C[i]$ in Zeile 5 ist egal, da jeder Zustand von jedem anderen in $C[i]$ zu erreichen ist. Da zunächst $F = \emptyset$ und $\forall q \in C : N[q] = 0$ wird jeder Zustand mindestens einmal in die Queue eingefügt. Weiterhin kann jeder Zustand q maximal $|P(q)| + 2$ mal in die Queue eingefügt werden, da nach jedem Aufruf der Funktion $\mathrm{lcp}(T_{\mathrm{out}}, q)$ entweder

- $N[q] = 0$: q wird nie wieder in die Queue eingereiht,
- $N[q] \neq 0, \pi_q = \varepsilon$: q wird zu F hinzugefügt, also q ebenfalls nie wieder in die Queue eingereiht oder

```
1  C ← topologicalSort(G_T^C);
2  for (i=C.length(); i>0; i--){
3     for each (q ∈ C[i]) N[q] ← 0;
4     F ← ∅;
5     Q.enqueue(q); // zufällig in C[i]
6     while(!Q.empty()){
7        q ← Q.dequeue();
8        π ← lcp(T_out, q);
9        for each (t ∈ Trans^T[q]){
10          if (π ≠ ε){
11             if (n(t) ∈ C[i] ∧ N[n(t)] > 0 ∧ n(t) ∉ F ∧ l_out(t) = ε){
12                N[n(t)]--;
13             }
14             l_out(t) ← l_out(t)π;
15             if (n(t) ∉ Q ∧ N[n(t)] = 0 ∧ n(t) ∉ F){
16                Q.enqueue(n(t));
17             }
18          }
19       }
20    }
21 }
```

Listing 4.1: Algorithmus zur Quasi-Determinierung

- $N[q] \neq 0, \pi_q \neq \varepsilon$: $P(q)$ wird also links um π_q gekürzt.

Der Algorithmus stoppt also nach spätestens nach $\Sigma_{q \in Q} (|P(q)| + 2)$ Durchläufen der **while**-Schleife und liefert $p(T_{\text{out}})$ zurück.

4.1.4. Komplexität

Die Funktionen $Q.\text{enqueue}()$, $Q.\text{dequeue}()$ und der Test $q \in Q$ bzw. $q \in F$ für einen Zustand q benötigen bei einer entsprechend gewählten Datenstruktur konstante Zeit $O(1)$. Folglich besitzt ein Durchlauf der **for**-Schleife in den Zeilen 9 – 19 eine Zeitkomplexität von $O(1)$ und wird $|Trans^T[q]|$ mal durchlaufen.

Die Berechnung des längsten gemeinsamen Präfixes π_q der Übergangsmarkierungen benötigt höchstens $(\pi_q + 1)(|Trans[q]| - 1)$ Vergleiche; die Modifikation der Übergangsmarkierungen und die Überprüfung, ob $N[q]$ korrigiert werden muss, können parallel geschehen. Der Test, ob q zu F hinzugefügt werden muss, ist konstanter Zeit möglich. Somit sind für einen Zustand q insgesamt nur $(|P(q)| + 1)(|Trans[q]| - 1)$ Zeichenvergleiche möglich, da bei jedem Aufruf entweder $P(q)$ während der Vergleiche um π_q verkürzt oder, falls $P(q) = \varepsilon$ gilt, die Bearbeitung von q als beendet markiert wird. Da $P(q)$ nicht anwachsen kann, folgt für die Zeit, die alle Aufrufe der Funktion $\text{lcp}(T_{\text{out}}, q)$ für einen Zustand q benötigen

$$O\left(|P(q)| \cdot |Trans[q]|\right)$$

Mit

$$P_{\max} := \max_{q \in Q} |P(q)|$$

folgt daraus für die Zeit, die alle Aufrufe der Funktion $\text{lcp}(T_{\text{out}}, q)$ zusammen benötigen:

$$O\left(\sum_{q \in Q} |P(q)| \cdot |Trans[q]|\right) = O\left(P_{\max} \cdot |E|\right)$$

Betrachtet man nun die **while**-Schleife in den Zeilen 6 – 20 zunächst einmal ohne den Aufruf $\text{lcp}(T_{\text{out}}, q)$, so wird diese für jeden Zustand q maximal $P_{\max}$ mal durchlaufen, wobei jeder Durchlauf Zeit $O(|Trans^T[q]|)$

benötigt. Mit $\mathrm{lcp}(T_{\mathrm{out}}, q)$ ergibt sich dann für die Komplexität der gesamten Schleife zu

$$O(P_{\max}|E|) + O\left(\sum_{q \in Q} |P(q)| \cdot |Trans^T[q]|\right) = O(P_{\max} \cdot |E|)$$

Da die Berechnung der starken Zusammenhangskomponenten und deren topologischer Sortierung in Zeit $O(|Q| + |E|)$ möglich ist, folgt für den kompletten Algorithmus aus Listing 4.1 eine Komplexität von

$$O(|Q| + |E| \cdot (P_{\max} + 1))$$

Dieses Laufzeitverhalten ist sehr effizient, da im Allgemeinen $P_{\max} \ll |Q|$ bzw. $P_{\max} \ll |E|$.

Für den Spezialfall eines azyklischen Transducers, kann die Laufzeit noch genauer angegeben werden: Da die **while**-Schleife in den Zeilen 6 – 20 für jeden Zustand genau einmal durchlaufen wird und $\mathrm{lcp}(T_{\mathrm{out}}, q)$ für Zustände q mit $|Trans[q]| = 1$ konstante Zeit benötigt, muss nur für Zustände $q \in Q \setminus F$ mit Ausgangsgrad > 1 eine Abschätzung mit $P_{\max}$ erfolgen:

$$O(|Q| + |E| + (|E| - (|V| - |F|)) \cdot P_{\max})$$

Eine Implementation des Algorithmus in Java ist in Abschnitt 6.3.1 gegeben.

4.2. Minimierung von endlichen Automaten

Das in diesem Kapitel beschriebene Verfahren zur Minimierung von endlichen Transducern ist das allgemein bekannte Verfahren zur Minimierung von deterministischen endlichen Automaten. Ich werde die Minimierung und deren Korrektheit zunächst kurz erläutern und anschließend zeigen, dass die Anwendung der Quasi-Determinierung und der Minimierung hintereinander auf einen endlichen Transducer T einen zu T äquivalenten minimalen Transducer $T_{\min}$ liefert.

4.2.1. Verfahren

Die Minimalität eines endlichen Automaten A steht immer in Bezug zu der von ihm erkannten Sprache $L(A)$. Um eine Aussage über die Minimalität

eines endlichen Automaten zu treffen bedarf es daher einer Spracheigenschaft. Diese geht auf den Satz von Myhill-Nerode (in [HU00]) zurück. Dieser besagt, dass mit jeder regulären Sprache (nach [VW02] eine Menge $L \subseteq \Sigma^*$ die von einem endlichen Automaten erkannt wird) eine Äquivalenzrelation $\sim_L$ definiert durch

$$\forall x, y \in \Sigma^* : x \sim_L y \text{ gdw. } \forall z \in \Sigma^* : (xz \in L \Leftrightarrow yz \in L) \tag{4.6}$$

assoziiert werden kann, die eine endliche Anzahl Äquivalenzklassen besitzt. Die Äquivalenzklasse eines $w \in \Sigma^*$ unter $\sim_L$ sei notiert als $[w]_L$. Außerdem ist $\sim_L$ rechts-invariant, d. h. für $u, v, w \in \Sigma^* : u \sim_L v \Rightarrow uw \sim_L vw$.

Anhand dieses Ergebnisses ist es möglich zu jeder regulären Menge (Sprache) L einen eindeutigen DEA mit einer minimalen Anzahl an Zuständen zu definieren.

Definition 4.6 *Der minimale DEA* $A_{\min} = (Q_{\min}, \Sigma, \delta_{\min}, q_{0_{\min}}, F_{\min})$ *zu einer Sprache* L *sei definiert durch*

- $Q_{\min} = \{[q]_L | \exists w \in \Sigma^* : \delta_{\min}(q_{0_{\min}}, w) = [q]_L\}$ *die Menge der Äquivalenzklassen, die von* $q_{0_{\min}}$ *aus erreichbar sind,*
- $\delta_{\min} : Q_{\min} \times \Sigma \to Q_{\min}$ *mit* $\forall [u]_L \in Q_{\min}, a \in \Sigma : \delta_{\min}([u]_L, a) = [ua]_L$
- $q_{0_{\min}} = [\varepsilon]_L$ *und*
- $F_{\min} = \{[u]_L | u \in L\}$.

Satz 4.3 *Der minimale Automat zu einer regulären Sprache* L *ist -bis auf Zustandsbenennungen- eindeutig und durch Definition 4.6 definiert, d. h. alle minimalen DEAs* A *mit* $L(A) = L$ *sind isomorph zu* $A_{\min}$.

Beweis Aus dem Satz von Myhill-Nerode folgt, dass jede Äquivalenzrelation $\sim$ die mit einem endlichen Automaten A über $u \sim v$ gdw. $\delta(q_0, u) = \delta(q_0, v),\ u, v \in \Sigma^*$ assoziiert werden kann, rechts-invariant ist und darüber hinaus eine Verfeinerung von $\sim_L$ darstellt:

Seien $u, v, w \in \Sigma^*$ und gelte $u \sim v$. Da $\sim$ rechts invariant ist, gilt $uw \sim vw$ und daher $u \in L \Leftrightarrow v \in L$. Folglich ist $u \sim_L v$ und die

Äquivalenzklasse von u unter $\sim$ in $[u]_L$ enthalten. Also sind insbesondere alle Äquivalenzklassen von $\sim$ in Äquivalenzklassen von $\sim_L$ enthalten.

Also ist jede durch einen DEA $A = (Q, \Sigma, \delta, q_0, F)$ induzierte Äquivalenzrelation $\sim$ eine Verfeinerung von $\sim_L$ und somit $|Q| \geq |Q_{\min}|$. Für den Fall der Gleichheit kann jeder Zustand von A mit einem Zustand von $A_{\min}$ identifiziert werden: Sei $q \in A$, dann existiert ein $w \in \Sigma^*$ mit $\delta(q_0, w) = q$, andernfalls wäre q nicht aus q_0 erreichbar und $|Q|$ nicht minimal. q mit $\delta_{\min}(q_{0_{\min}}, w)$ in $A_{\min}$ konsistent zu identifizieren, da für $u, v \in \Sigma^*$: $\delta(q_0, u) = \delta(q_0, v) = q \Rightarrow u \sim_L v \Rightarrow \delta_{\min}(q_{0_{\min}}, u) = \delta_{\min}(q_{0_{\min}}, v)$. □

Ein Verfahren zur Konstruktion eines minimalen DEA $A_{\min}$ zu einer Sprache L lässt sich für die Eingabe eines DEA $A = (Q, \Sigma, \delta, q_0, F)$ mit $L = L(A)$ wie folgt formulieren:

1. Sei $\Pi_0 = (Q_{0,0}, Q_{0,1}),\ Q_{0,0} = F,\ Q_{0,1} = Q \setminus F$

2. Sei $\Pi_i = (Q_{i,0}, Q_{i,1}, \ldots, Q_{i,k_i}), k_i \geq 0$. Dann gehören zwei Zustände $q, q' \in Q$ zu einem Block $Q_{i+1,j} \in \Pi_{i+1}, 0 \leq j \leq |\Pi_{i+1}|$ genau dann, wenn für alle $a \in \Sigma$:

$$\delta(q, a) \in Q_{i,k} \Leftrightarrow \delta(q', a) \in Q_{i,k}, 1 \leq k \leq k_i$$

3. Solange $\Pi_i \neq \Pi_{i+1}$, ist gehe zu Schritt 2.

4. Π_i die Zustandsmenge des minimalen Automaten $A_{\min}$.

Satz 4.4 *Der oben angegebene Algorithmus ist korrekt und berechnet den zu $L(A)$ minimalen DEA $A_{\min} = (Q_{\min}, \Sigma, \delta_{\min}, q_{0_{\min}}, F_{\min})$, wobei*

- $Q_{\min} = \Pi_i$,
- $\delta_{\min} : Q_{\min} \times \Sigma \to Q_{\min}$ *mit* $\forall q, q' \in Q, a \in \Sigma : \exists Q_{i,j}, Q_{i,k} \in \Pi_i :$ $\delta(q, a) = q' \Rightarrow \delta_{\min}(Q_{i,j}, a) = Q_{i,k}, q \in Q_{i,j}, q \in Q_{i,k}$,
- $q_{0_{\min}} = Q_{i,j}$ *so, dass* $q_0 \in Q_{i,j}$,
- $F_{\min} = \{Q_{i,j} | \exists q \in F : q \in Q_{i,j}\}$ *ist.*

Beweis Die durch die Eingabe A induzierte Äquivalenzrelation $\sim$ mit

$$u \sim v \text{ gdw. } \delta(q_0, u) = \delta(q_0, v)$$

stellt eine Assoziation der Zustände $q \in Q$ mit je einer Äquivalenzklassen von $\sim$ dar und ist eine Verfeinerung von $\sim_L$ nach Satz 4.3.

Seien nun $q, q' \in Q$. Dann heißen q und q' unterscheidbar in Länge $\leq x$, falls

$$\exists w \in \Sigma^*, |w| \leq x : \delta(q, w) \in F \wedge \delta(q', w) \notin F$$

Falls dies nicht der Fall ist, so heißen q und q' ununterscheidbar (geschrieben als $q \equiv q'$), und $\forall w \in \Sigma^* : \delta(q, w) \in F \Leftrightarrow \delta(q', w) \in F$. Also folgt mit $q = \delta(q_0, u), q' = \delta(q_0, v)$:

$$\forall u, v \in \Sigma^* : \delta(q_0, uw) \in F \Leftrightarrow \delta(q_0, vw) \in F$$

Somit gilt

$$u \sim v \Rightarrow q \equiv q' \Rightarrow [u]_L = [v]_L$$

Zeige nun per Induktion über x, dass der Algorithmus die Partition $\Pi_x = (Q_{x,0}, \ldots, Q_{x,k_x})$, $k_x \geq 0$ so erstellt, dass je zwei Zustände $q, q' \in Q_{x,k}, 0 \leq k \leq k_x$ nicht in Länge $\leq x$ unterscheidbar sind:

Induktionsbeginn: Sei $x = 0$, also $w = \varepsilon$. Da folgt mit $\forall q \in Q : \delta(q, \varepsilon) = q$ aus $\Pi_0 = (Q \setminus F, F)$, dass alle q, q' aus einem Block in Π_0 nicht unterscheidbar in Länge $x = 0$ sind.

Induktionsschluss: Sei nun $x > 0$ beliebig aber fest gewählt und für alle $y < x$ je zwei Zustände $q, q' \in Q_{y,k}, 0 \leq k \leq |\Pi_y|$ nicht in Länge $\leq y$ unterscheidbar sind. Dann gehören q und q' in Π_x zu einem Block genau dann, wenn für alle $a \in \Sigma$ die Übergänge in gleiche Blöcke aus Π_{x-1} führen. In den Blöcken von Π_{x-1} sind nach Induktionsvoraussetzung alle Paare (q, q') von Zustände nicht in Länge $\leq y$ unterscheidbar, gilt $\forall w \in \Sigma^*, |w| < x : \delta(q, w) \in F \Leftrightarrow \delta(q', w) \in F$. Für alle Paare $(q, q') \in Q_{x,k}, 0 \leq k \leq |\Pi_x|$ ist also $\forall w \in \Sigma^*, a \in \Sigma : \delta(q, aw) \in F \Leftrightarrow \delta(q', aw) \in F$. Somit sind alle Paare an Zuständen aus einem Block in Π_x nicht in Länge $\leq |aw| = x$ unterscheidbar.

Falls für ein $i > 0$ nun $\Pi_i = \Pi_{i-1}$ gilt, so existiert kein w endlicher Länge, dass zwei Zustände aus einem Block von Π_i unterscheidet und für $q, q' \in Q_{x,k}, 0 \leq k \leq |\Pi_i| : q \equiv q'$. Anders formuliert fasst jeder Block

in Π_i die Menge der Zustände zusammen, die gleiche Äquivalenzklassen repräsentieren. Da jeder Block nur eine Äquivalenzklasse repräsentiert und anhand von Satz 4.3 folgt durch Interpretation der Blöcke als Zustände von $A_{\min}$:

$$Q_{\min} = \Pi_i$$

Da nach Definition für jeden Übergang $\delta(q, a) = q'$ mit $q \neq q'$ Blöcke $Q_{i,j}, Q_{i,k} \in Q_{\min}$ mit $Q_{i,j} \neq Q_{i,k}$ existieren, so dass

$$q \in Q_{i,j} \wedge q' \in Q_{i,k} \Rightarrow \delta_{\min}(Q_{i,j}, a) = Q_{i,k},$$

ist $\delta_{\min}$ konsistent definiert und $A_{\min}$ wird korrekt nach Definition 4.6 berechnet. □

4.2.2. Komplexität

Das oben vorgestelle Verfahren zur Minimierung eines DEA lässt sich algorithmisch auf verschiedene Weise umsetzen. Die bekanntesten Algorithmen sind die Algorithmen von Huffmann und Hopcroft.

Der Huffmann-Algorithmus aus [Huf54] und [Moo56] ist ein oft in den Grundlagenbüchern der theoretischen Informatik erläuterte Algorithmus. Er betrachtet je Paare von Zuständen und markiert anhand von Listen alle unterscheidbaren Zustände. Der Algorithmus besitzt eine Zeitkomplexität von $O(|Q|^2)$.

Der effizienteste Algorithmus für die Minimierung geht auf Hopcroft zurück und wird in [AHU74] beschrieben. Er besitzt für eine alphabetsabhängige Implementierung eine Zeitkomplexität von $O(|\Sigma| \cdot |Q| \cdot \log |Q|)$. Macht man den Algorithmus alphabetsunabhängig, so erhält man eine Variante, die lediglich vom Eingangsgrad jedes Zustands abhängt, diese Variante benötigt somit $O(|E| \cdot \log |Q|)$ Zeit. In Abschnitt 6.3.2 ist eine Implementierung dieses Algorithmus in Java gegeben.

Für den Spezialfall eines azyklischen Automaten existiert ein weiterer Algorithmus, der in linearer Zeit $O(|Q| + |E|)$ arbeitet und dabei ähnlich wie die Quasi-Determinierung entgegen einer topologischen Sortierung vorgeht. Der Algorithmus geht auf die Doktorarbeit von Daniel Revuz zurück, die ein Jahr später in [Rev92] veröffentlicht wurde.

Ein weiterer erwähnenswerter Algorithmus für die Minimierung von endlichen Automaten geht auf [Brz62] zurück. Er benutzt einen Trick indem er zunächst den gegebenen Automaten umkehrt, das Ergebnis determiniert und diesen wiederum umkehrt und determiniert (für einen

endlichen Automaten A erkennt der umgekehrte Automat A^R die Sprache $L(A^R) = \{w|w^R \in L(A)\}$). Der Algorithmus arbeitet in Zeit $O(2^{|Q|\cdot|\Sigma|})$, allerdings besitzt der Algorithmus in der Praxis meist ein gutes mittleres Laufzeitverhalten, welches teilweise sogar unter dem des Hopcroft-Algorithmus liegt.

4.3. Minimierung von Transducern

Wie bereits in der Einleitung erwähnt, gilt es nun zu darzulegen, dass die Anwendung der Minimierung für endliche Automaten auf einen endlichen Transducer zu einem äquivalenten minimalen Transducer führt, falls man diesen vorher anhand der Quasi-Determinierung modifiziert hat.

Da ein endlicher Automat nur eine Markierung pro Übergang, ein Tansducer jedoch jeweils eine Ein- und Ausgabemarkierung besitzt, muss das Paar (Eingabe, Ausgabe) als Kantenmarkierung für den Minimierungsalgorithmus interpretiert werden, damit auch nach der Minimierung eines Transducers T $f_T \equiv f_{T_{\min}}$ gilt, wobei $T_{\min}$ das Ergebnis der Minimierung bezeichnet.

Definition 4.7 *Sei $T = (Q, \Sigma, \Delta, \delta, \sigma, \lambda, q_0, F)$ ein Transducer. Dann kann T als DEA interpretiert werden. Der T entsprechende Automat A_T ist gegeben durch $A_T = (Q, (\Sigma \times \Delta), \delta_A, q_0, F)$ mit $\delta_A : Q \times (\Sigma \times \Delta^*) \to Q$ definiert als*

$$\delta_A(q, (a, w)) = q' \Leftrightarrow \delta(q, a) = q' \wedge \sigma(q, a) = w$$

Minimiert man nun den Automaten A_T, so werden alle äquivalenten Zustände q, q' verschmolzen. Zwei Zustände q, q' sind in diesem Fall äquivalent genau dann, wenn

$$\forall x \in (\Sigma^* \times \Delta^*) : \delta_A(q, x) \in F \Leftrightarrow \delta(q', x) \in F$$

Übertragen auf einen Transducer T werden hier also die Zustände q, q' von T verschmolzen, für die $\forall w \in \Sigma^*$:

$$\begin{cases} \delta(q, w) \in F \Leftrightarrow & \delta(q', w) \in F \\ \delta(q, w) \in F \Rightarrow & \forall i \in [0, |w| - 1] : \\ & \sigma(\delta(q, w_0 \dots w_i), w_{i+1}) = \sigma(\delta(q', w_0 \dots w_i), w_{i+1}) \end{cases}$$

gilt. Ich zeige nun, dass die Automatenminimierung angewendet auf einen quasi-determinierten Transducer T' wie in Definition 4.7 beschrieben zu dem minimalen Transducer $T_{\min}$ aus Satz 3.1 führt:

Satz 4.5 *Sei T ein Transducer und $T'_{\min}$ das Ergebnis, der Anwendung der Quasi-Determinierung und der Minimierung nacheinander auf T. Seien $u, v \in \Sigma^*$ zwei Zeichenketten mit $u \sim_f v$. Dann gilt*

$$u \sim_f v \Rightarrow \delta(q_0, u) = \delta(q_0, v)$$

und $T'_{\min}$ ist der minimaler Transducer zu T nach Satz 3.1.

Beweis Sei $T'_{\min} = (Q, \Sigma, \Delta, \delta, \sigma', \lambda', q_0, F)$ der nach obigem Verfahren ermittelte minimale Transducer. Sei $q_u = \delta(q_0, u), q_v = \delta(q_0, v)$ und $w \in \Sigma^*, w = w_1 \dots w_n$, $w_0 := \varepsilon$. Mit $u \sim_f v$ folgt nach Definition 3.1:

$$\delta(q_u, w) \in F \Leftrightarrow \delta(q_v, w) \in F \tag{4.7}$$

Nach Satz 4.2 gilt weiterhin

$$q_u \in F \Rightarrow \sigma(q_u, w) = \sigma(q_v, w)$$

Da $\sim_f$ rechts invariant ist, folgt $\forall i \in [0, n] : u \sim_f v \Rightarrow uw_0 \dots w_i \sim_f vw_0 \dots w_i$. Für $\delta(q_u, w) \in F$ und $i \in [0, n-1]$ ist somit:

$$\sigma(\delta(q_u, w_0 \dots w_i), w_{i+1} \dots w_n) = \sigma(\delta(q_v, w_0 \dots w_i), w_{i+1} \dots w_n)$$

Also insbesondere

$$\forall i \in [0, n-1] : \sigma(\delta(q_u, w_0 \dots w_i), w_{i+1}) = \sigma(\delta(q_v, w_0 \dots w_i), w_{i+1}) \tag{4.8}$$

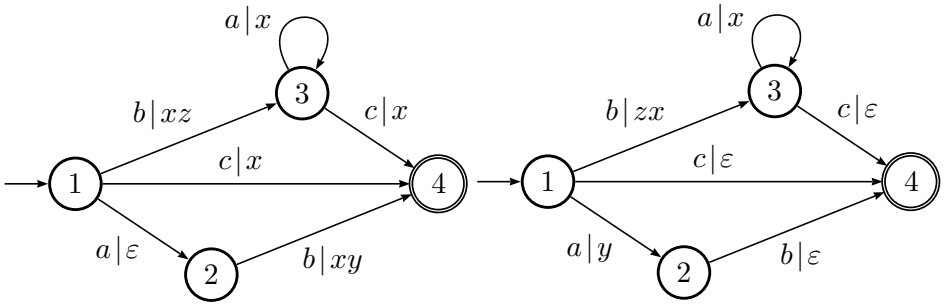
Da Gleichungen (4.7) und (4.8) darauf führen, dass die Zustände q_u und q_v äquivalent sind, wurden q_u und q_v durch die Automatenminimierung verschmolzen. Da $T'_{\min}$ nach Satz 4.5 nicht mehr Zustände als $\sim_f$ Äquivalenzklassen besitzen kann und $f_{T'_{\min}} \equiv f_T$ ist, ist $T'_{\min}$ ein minimaler Transducer zu T.

Weiterhin gilt nach der Abarbeitung von $u \in D(f_{T'_{\min}})$:

$$\begin{aligned} g(u) &= \bigwedge_{\substack{w \in \Sigma^* \\ uw \in Dom(f_T)}} f_T(uw) = \bigwedge_{\substack{w \in \Sigma^* \\ \delta(q_0, uw) \in F}} \lambda\sigma(q_0, uw) \\ &= \lambda\sigma(q_0, u) P(\delta(q_0, u)) = \sigma'(q_0, u) \end{aligned}$$

Folglich ist die Verteilung der Ausgaben im Transducer identisch mit der, die für den minimalen Transducer aus Satz 3.1 festgelegt wurde und $T'_{\min}$ der minimale Transducer nach Satz 3.1. □

Transducer besitzen jedoch im Gegensatz zu Automaten keinen eindeutig bestimmten minimalen Transducer, weil Ausgabemarkierungen entlang der Übergänge im Transducer verschoben werden können, ohne dass sich die Minimalität verändert. Trotzdem lässt sich eine Aussage über die Struktur aller minimalen äquivalenten Transducer bezüglich einer sequentiellen Funktion treffen:



Transducer T, $\lambda = \varepsilon$ Transducer T', $\lambda = x$

Abbildung 4.5.: Zwei minimale, äquivalente Transducer mit unterschiedlichen Verteilungen der Ausgabemarkierungen

Satz 4.6 *Sei T ein Transducer, der die Funktion f_T berechnet. Dann unterscheiden sich alle zu T äquivalenten minimalen Transducer nur in der Verteilung der Ausgabemarkierungen auf den Pfaden aus dem Start- in einen Endzustand.*

Beweis Sei $\mathcal{T} = \{T' \mid f_{T'} \equiv f_T\}$ die Menge der Transducer T, die f_T berechnen. Dann kann jeder Transducer in $\mathcal{T}$ anhand des obigen Verfahrens minimiert werden. Das Ergebnis dieses Verfahrens ist der minimale Transducer nach Satz 3.1. Da der zweite Schritt des Verfahrens, die Automatenminimierung, keine Auswirkungen mehr auf einen Transducer aus $\mathcal{T}$ haben kann, fallen alle Modifikationen an Transducern $T \in \mathcal{T}$ zurück auf den ersten Schritt, die Quasi-Determinierung. Da diese lediglich die Ausgabemarkierungen eines Transducers modifiziert, folgt die Behauptung. □

Es existiert jedoch noch ein wichtiger Unterschied zur Minimierung DEAs. Die in dieser Arbeit verwendeten Transducer lassen Zeichenketten beliebiger Länge als Ausgabemarkierungen der Übergänge zu. Dies

führt dazu, dass in einer Vorverarbeitungsphase die Übergangsmarkierungen so vorbereitet werden müssen, dass eine Überprüfung der Gleichheit zweier Kantenmarkierung während der Minimierung trotzdem in konstanter Zeit möglich ist. Eine Möglichkeit dazu ist die Konstruktion eines Tries mit nummerierten Blättern, der die Ausgabemarkierungen den jeweiligen Kanten in A_T zuordnen. Die Konstruktion dies solches Tries benötigt lineare Zeit und lässt sich somit durch die Summe der Längen der Ausgaben eines Transducers beschränken:

$$O\left(\sum_{q \in Q} \sum_{t \in Trans[q]} |l_{\text{out}}(t)|\right)$$

4.4. Fazit

Das vorgestelle Verfahren ermöglicht die Minimierung endlicher Automaten mit Ausgabe und basiert im wesentlichen einer Hintereinanderausführung der Quasi-Determinierung und der Automatenminimierung auf einem gegebenen Transducer.

Das Verfahren ist effizient und arbeitet im allgemeinen Fall unter Verwendung der alphabetsunabhängigen Variante des Hopcroft-Algorithmus zur Minimierung in Zeit

$$O(S + |Q| + |E| \cdot (P_{\max} + \log |Q|))$$

Für den Fall eines azyklischen Transducers kann dieses Verhalten durch die Verwendung der Minimierung nach [Rev92] weiter beschränkt werden auf

$$O(S + |Q| + |E| + (|E| - (|V| - |F|)) \cdot P_{\max})$$

Darüber hinaus lässt sich das Verfahren auf ein weiteres Feld, als das hier vorgestelle übertragen:

Die Quasi-Determinierung kann auf Transducer mit Zeichenketten und Gewichten an der Übergängen erweitert werden. Auf diese Weise ist die Minimierung auf für gewichtete Transducer anwendbar, welche häufig für die Verarbeitung natürlicher Sprache eingesetzt werden.

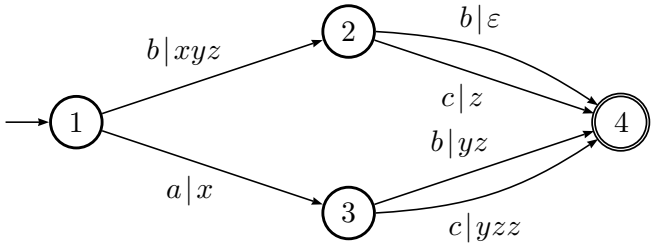
Die Minimierung von endlichen Transducern funktioniert auch für p-subsequentielle Transducer. p-Subsequentielle Transducer sind Transducer, die in einem Endzustand q eine aus bis zu p finalen Ausgaben $(\Phi(q))_i \in$

$\Delta^*, 1 \leq i \leq p$ erlauben. Ein subsequentieller Transducer T ist also ein 9-Tupel
$T = (Q, \Sigma, \Delta, \delta, \sigma, \lambda, \Phi, q_0, F)$, wobei $(Q, \Sigma, \Delta, \delta, \sigma, \lambda, q_0, F)$ ein endlicher Transducer T nach Definition 2.1 ist und $\Phi : F \rightarrow (\Delta^*)^p$ eine Funktion, die mit jedem Endzustand eine Ausgabe verbindet. Die von T getätigte Ausgabe für ein Wort $w \in Dom(f_T)$ ist also eine Menge von bis zu p verschiedenen Zeichenketten $f(w) = \lambda\sigma(q_0, w)\phi(\delta(q_0, w))$.

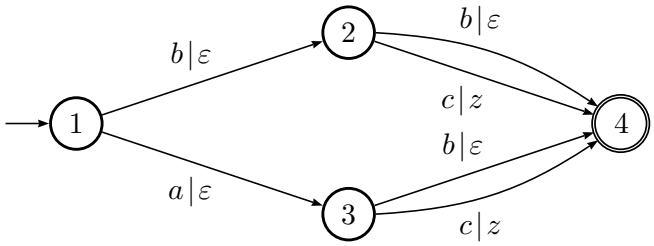
Die Minimierung kann hier durchgeführt werden, indem man eine bijektive Abbildung h von der Menge der subsequentiellen Transducer auf die endlichen Transducer definiert: Sei T ein subsequentieller Transducer, dann erhält man einen endlichen Transducer $h(T)$ durch folgende Schritte:

- Man fügt einen neuen Zustand f hinzu. f wird einziger Endzustand.
- Für jede der Ausgaben $(\Phi(q))_i, 1 \leq i \leq p$ aus einem Endzustand q wird ein neuer Übergang von q zu f hinzugefügt mit der Ausgabe $(\Phi(q))_i$ und einer Eingabemarkierung $\phi_i \notin \Sigma$.

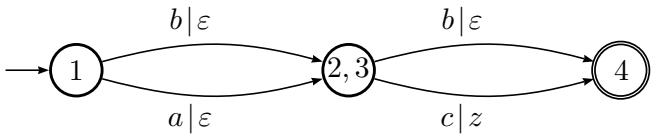
Dann ist $h(T) = (Q \cup f, \Sigma \cup (\bigcup_{q \in F}\{\phi(q)\}), \Delta \cup (\bigcup_{1 \leq i \leq p}\{\Phi_i\}), \delta, \sigma, \lambda, q_0, \{f\})$ ein endlicher Transducer, der wie oben erläutert minimiert werden kann und anschließend anhand der Umkehrabbildung h^{-1} wieder in einen subsequentiellen Transducer umgewandelt werden kann.



(a) $\lambda = xzx$



(b) $\lambda = xzxxyz$



(c) $\lambda = xzxxyz$

Abbildung 4.6.: (a) Transducer T aus Abbildung 1.1 mit $\lambda = xzx$; (b) der zu T äquivalente quasi-determinierte Transducer T'; (c) das Ergebnis der Minimierung von T': der zu T äquivalente minimale Transducer $T_{\min}$

5. Äquivalenztest

5.1. Motivation

Gegeben sei das Problem: Sind zwei Transducer T und T' äquivalent bzw. gilt $f_T \equiv f'_T$?

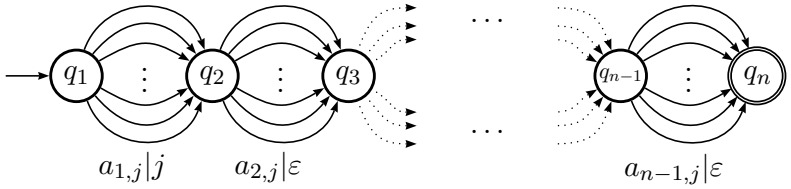
Für die Lösung dieses Problems gilt es die Äquivalenz der sequentiellen Funktionen f_T und $f_{T'}$ zu prüfen. Also zu zeigen, dass

$$\forall w \in \Sigma^* : \begin{cases} w \in Dom(f_T) \Leftrightarrow w \in Dom(f_{T'}) \\ w \in Dom(f_T) \Rightarrow f_T(w) = f_{T'}(w) \end{cases}$$

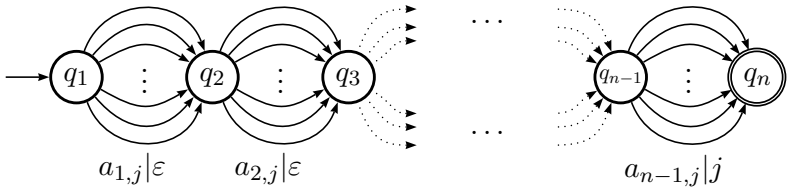
hält. Die erste Bedingung $\forall w \in \Sigma^* : w \in Dom(f_T) \Leftrightarrow w \in Dom(f_{T'})$ kann dabei in polynomieller Zeit geprüft werden, da sich dies auf das Problem der Äquivalenz zweier endlicher Automaten zurückführen lässt. Die zweite Bedingung allerdings bereitet Probleme, da für zwei Pfade mit identischen Ausgaben, diese entlang der Übergänge vom Start- in einen Endzustand gegeneinander verschoben sein können. Zur Überprüfung der identischen Ausgaben müsste also jeder Pfad in T und T' in einen Endzustand durchlaufen werden.

Sei nun T ein Transducer mit n Zuständen $Q = \{q_1, \ldots, q_n\}$, wobei q_1 der Startzustand, q_n einziger Endzustand ist. Weiterhin habe jeder Zustand q_i $(1 \leq i < n-1)$ m Übergänge in den Zustand q_{i+1}, wobei für einen Übergang von q_i nach q_{i+1} die Eingabe eindeutig durch $a_{i,j} \in \Sigma$ und die Ausgaben durch $b_{i,j} \in \Delta^*$ $(1 \leq j \leq m)$ mit $b_{i,j} = \varepsilon$ für $1 < i \leq n-1$, $q_{1,j} = j$ bestimmt ist. Weiterhin sei T' definiert wie T, habe jedoch die Ausgabemarkierungen $b_{i,j} = \varepsilon$ für $1 \leq i < n-1$, $q_{n-1,j} = j$ $(1 \leq j \leq m)$. Die Transducer T und T' sind in Abbildung 5.1 skizziert.

Wie sieht sind T und T' äquivalent und besitzen je n Zustände und $(n-1)m$ Übergänge. Folglich existieren in jedem der beiden Transducer $(n-1)^m$ verschiedene Pfade vom jeweiligen Startzustand in den Endzustand. Selbst unter der Annahme, dass ein Paar von Pfaden in konstanter Zeit überprüfbar ist, benötigt die Überprüfung aller Pfade Zeit



(a) Transducer T



(b) Transducer T'

Abbildung 5.1.: Die angegebene Markierung $a_{i,j}|b_{i,j}$ bezieht sich für $1 \leq j \leq m$ jeweils auf den j. Übergang in der Adjazenzliste des Zustands i.

$O((n-1)^m) = O(2^m)$. Also ist ein effizienter Test auf Äquivalenz für zwei gegebene Transducer auf diesem Wege nicht möglich.

5.2. Algorithmus

Aufgrund der Ergebnisse des letzten Kapitels ist jedoch nun zu jedem Transducer T die Konstruktion eines eindeutig bestimmten, zu T äquivalenten minimalen Transducers $T_{\min}$ möglich. Demnach kann für zwei Transducer $T_1 = (Q_1, \Sigma_1, \Delta_1, \delta_1, \sigma_1, \lambda_1, q_{1,0}, F_1)$ und $T_2 = (Q_2, \Sigma_2, \Delta_2, \delta_2, \sigma_2, \lambda_2, q_{2,0}, F_2)$ zunächst der minimale Transducer $T_{1,\min}$ bzw. $T_{2,\min}$ nach Satz 3.1 berechnet werden. Da diese minimalen Transducer $T_{1,\min}$ und $T_{2,\min}$ eindeutig für die jeweilige sequentielle Funktion f_{T_1} und f_{T_2} sind, folgt

$$f_{T_1} \equiv f_{T_2} \Leftrightarrow (\lambda_1 = \lambda_2) \wedge \left(A_{T_{1,\min}} \text{ isomorph zu } A_{T_{2,\min}}\right) \tag{5.1}$$

Hierbei sei $\Sigma_1 = \Sigma_2$ und $\Delta_1 = \Delta_2$ implizit vorausgesetzt.

Eine Umsetzung dieses Tests kann wie folgt geschehen: Für die Eingabe zweier Transducer T_1 und T_2 werden zunächst die minimalen Transducer $T_{1,\min}$ und $T_{2,\min}$ berechnet. Falls nun $\lambda_{1,\min} \neq \lambda_{2,\min}$ gebe ‚nicht äquivalent' aus. Andernfalls wird nun die Isomorphie der $T_{1,\min}$ und $T_{2,\min}$ entsprechenden Automaten überprüft und deren Ergebnis zurück gegeben. Die Überprüfung der Isomorphie ist dabei als parallele gekoppelte Tiefensuche in beiden Automaten möglich:

Seien $T_{1,\min}$ auf $T_{2,\min}$ isomorph, dann existiert eine bijektive Abbildung $f : Q_1 \to Q_2$. Unter dieser bilden zwei Zustände $q_1 \in Q_1, q_2 \in Q_2$ Original und Abbild $f(q_1) = q_2$ falls gilt

$$\begin{cases} q_1 \in F \Leftrightarrow q_2 \in F \\ |Trans[q_1]| = |Trans[q_2]| \\ \forall t_1 \in Trans[q_1] \exists t_2 \in Trans[q_2] : l(t_1) = l(t_2) \wedge f(n(t_1)) = f(n(t_2)) \end{cases} \tag{5.2}$$

Im folgenden seien Zustände q_1, q_2 mit $f(q_1) = q_2$ auch als äquivalent bezeichnet.

Zwar ist die Berechnung dieser Abbildung auf Graphen ist ein NP-vollständiges Problem, da jedoch Transducer genau einen Startzustand besitzen, ist eine anfängliche Assoziation zwischen den beiden Mengen Q_1 und Q_2 gegeben durch $f(q_{1,0}) = q_{2,0}$, wobei $q_{1,0}$ und $q_{2,0}$ die Startzustände der Transducer $T_{1,\min}$ und $T_{2,\min}$ angeben. Ab diesem Anfangspunkt kann die rekursive Bedingung (5.2) nun mittels einer Tiefensuche in beiden Transducern geprüft werden, indem in beiden Transducern je Kanten gleicher Markierung für die Fortsetzung der Suche gewählt werden. Eine Umsetzung dieses Vorgehens ist in Listing 5.1 gegeben. Dabei bezeichne

- M ein Array der Größe $|Q_1| + |Q_2|$, welches die bereits markierten Zustände enthält,
- A eine Tabelle, welche die Abbildung $A : Q_1 \to Q_2$ modelliert, d. h. die Zustände der beiden Transducer miteinander assoziiert,
- e eine boolesche Variable.

Satz 5.1 *Der Algorithmus aus Listing 5.1 arbeitet korrekt nach Gleichung* (5.1).

```
1  function areIsomorph(q1, q2):boolean {
2    e ← true;
3    M[q1] ← M[q2] ← 1;
4    A.put(q1, q2);
5    if(|Trans[q1]| ≠ |Trans[q2]| ∨ ¬(q1 ∈ F1 ⇔ q2 ∈ F2))
6      e ← false;
7    else for each(t1 ∈ Trans[q1]){
8        if(∃t2 ∈ Trans[q2] : l(t1) = l(t2)){
9          if(M[n(t1)] ≠ M[n(t2)])
10           e ← false;
11         if(M[n(t1)] = M[n(t2)] = 1 ∧ A.get(n(t1)) ≠ n(t2))
12           e ← false;
13         if(M[n(t1)] = M[n(t2)] = 0 ∧ e)
14           e ← areEquivalent(n(t1), n(t2));
15       }else{
16         e ← false;
17       }
18     }
19   return e;
20 }
21
22 function areEquivalent(T1, T2):boolean {
23   for each(q ∈ Q1 ∪ Q2)
24     M[q] ← 0;
25   A ← ∅;
26   T1,min ← minimize(T1);
27   T2,min ← minimize(T2);
28   if (λ1 ≠ λ2) return false;
29   return areIsomorph(q1,0, q2,0)
30 }
```

Listing 5.1: Der Äquivalenztest-Algorithmus

Beweis Sei zunächst die Funktion areIsomorph(q_1, q_2) betrachtet: e wird initial auf den Wert **true** gesetzt und erhält den Wert **false** genau dann, wenn

1. in Zeile 5 $(q_1 \in F_1 \wedge q_2 \notin F_2) \vee (q_1 \notin F_1 \wedge q_2 \in F_2) \vee |Trans[q_1]| \neq |Trans[q_2]|$ gilt,

2. in Zeile 8 ein Übergang $t_1 \in Trans[q_1]$ existiert, so dass $\forall t_2 \in Trans[q_2] : l(t_1) \neq l(t_2)$,

3. in Zeile 9 genau eine der beiden Senken eines Übergangs noch nicht besucht worden ist oder in Zeile 11, falls die Senken der Übergänge markiert, aber nicht miteinander assoziierte Zustände sind, sowie

4. in Zeile 13, die Zustände $n(t_1)$ und $n(t_2)$ nicht äquivalent sind.

Aus 1. lassen sich sofort die ersten beiden Gleichungen von Bedingung (5.2) ableiten. Darüber hinaus korrespondiert Punkt 2 mit dem ersten Konjunktionsglied der dritten Gleichung.

Betrachtet man nun Punkt 3, so folgt aus $M[n(t_1)] \neq M[n(t_2)]$ zwangsläufig $T.\text{get}(n(t_1)) \neq n(t_2)$ und hieraus wiederum, dass die beiden Zustände nicht äquivalent sein können. Außerdem gilt, dass Übergänge $t_1 \in Trans[q_1]$ und $t_2 \in Trans[q_2]$, deren Senken beide in assoziierte Zustände (d. h. $f(n(t_1)) = n(t_2)$) führen, eine Isomorphie von q_1 und q_2 nicht verändern: entweder wurde areIsomorph$(n(t_1), n(t_2))$ aufgerufen und abgearbeitet. Folglich sind $n(t_1)$ und $n(t_2)$ äquivalent (andernfalls würde areIsomorph wegen Zeile 13 nicht weiter aufgerufen). Oder $n(t_1)$ (und damit $n(t_2)$) befindet sich im beschrittenen Pfad vom Startzustand zu q_1, d. h. die Funktion areIsomorph$(n(t_1), n(t_2))$ ist aufgerufen und wartet auf die Rückgabe eines rekursiven Aufrufs. In diesem Fall sind die zu den Zustände $n(t_1), n(t_2)$ führenden Übergänge nicht relevant, da $n(t_1), n(t_2)$ äquivalent sind genau dann, wenn q_1 und q_2 äquivalent sind.

Also wird in den Zeilen 5 – 17 e auf **false** gesetzt gdw. $n(t_1)$ und $n(t_2)$ nicht äquivalent sind. Es folgt also, dass die Funktion areIsomorph(q_1, q_2) **true** zurückliefert, falls q_1 und q_2 nach (5.2) äquivalent sind.

Da der Rest des Algorithmus in den Zeilen 23–29 Gleichung (5.1) modelliert, folgt die Behauptung. □

5.3. Komplexität

Der Algorithmus arbeitet rekursiv mit der Funktion areIsomorph(q_1, q_2) und markiert dabei in Zeile 4 bei jedem Aufruf zwei Zustände. Da diese Zustände nicht markiert sein können, da areIsomorph nur für Zustände aufgerufen wird, die die Bedingung $M[n(t_1)] = M[n(t_2)] = 0$ (Zeile 13) erfüllen, wir die Funktion areIsomorph(q_1, q_2) höchstens $\min\{|Q_1|, |Q_2|\}$ mal aufgerufen.

Die **for**-Schleife in den Zeilen 7–18 wird dabei insgesamt $\min\{|E_1|, |E_2|\}$ mal durchlaufen. Jeder dieser Durchläufe benötigt Zeit $O(S_{\min})$ mit

$$S_{\min} := \sum_{\substack{t_1 \in Trans[q_1] \\ t_2 \in Trans[q_2] \\ l_{\text{in}}(t_1) = l_{\text{in}}(t_2)}} \min\left\{|l_{\text{out}}(t_1)|, |l_{\text{out}}(t_2)|\right\},$$

sieht man vom rekursiven Funktionsaufruf in Zeile 14 ab: Die Markierungen der ausgehenden Übergänge eines Zustands können ggf. in einer Vorverarbeitungsphase in nach Eingabemarkierung sortierten Tabellen bzw. Arrays gespeichert werden. In diesen ist die Suche nach dem entsprechenden Übergang t_2 aus q_2 in konstanter Zeit möglich. Der Vergleich der Markierungen benötigt nun Zeit $O(\min\{|l_{\text{out}}(t_1)|, |l_{\text{out}}(t_2)|\})$. Die restlichen Vergleiche in den Zeilen 9, 11 und 13 können, für eine geeignete Implementation von M, in konstanter Zeit durchgeführt werden.

Da die Zeilen 2–6 in ebenfalls konstanter Zeit abgearbeitet werden können, folgt für die Summe der Laufzeiten der Funktion areIsomorph(q_1, q_2) für einen kompletten Durchlauf des Algorithmus

$$O(S_{\min} + \min\{|Q_1|, |Q_2|\} + \min\{|E_1|, |E_2|\})$$

Die Minimierung der Transducer T_1, T_2 in den Zeilen 26 und 27 ist nach Kapitel 4 in Zeit $O(S_i + |Q_i| + |E_i| \cdot (P_{i,\max} + \log |Q_i|))$ möglich. Somit folgt letztlich mit der Zeit für die Initialisierung von M für den gesamten Algorithmus eine Komplexität von

$$O(S + |Q| + |E| \cdot (P_{\max} + \log |Q|))$$

wobei $S = S_1 + S_2$, $|Q| = |Q_1| + |Q_2|$, $|E| = |E_1| + |E_2|$ und $P_{\max} = P_{1,\max} + P_{2,\max}$.

6. Implementierung

In diesem Kapitel werden die Datenstrukturen sowie die Implementierungen der erwähnten Algorithmen in Java vorgestellt. Da mir die Angabe der qualifizierten Paketnamen zu lang erscheint, sei das Präfix de.unihannover.thi.thomas.bachelor in den folgenden Angaben ausgelassen.

6.1. Reales Laufzeitverhalten in Java

Betrachtet man die laufzeit von Algorithmen auf realen Systemen, so gilt es zu berücksichtigen, dass die theoretisch getroffenen Annahmen meist nicht mehr erfüllt sind. Es treten Randeffekte auf, die durch die Architektur heutiger PC-Systeme bedingt sind, wie z. B. beschränkte Adresslängen oder Verzögerungen durch die Reorganisation zwischen RAM- und Swap-Speicher.

So ist unter anderem die Größe der Zustandsmenge eines Transducers durch den verwendeten Datentyp integer auf $2^{31} - 1$ Zustände beschränkt. Allerdings wäre auch für einen Datentyp unbeschränkten Wertebereichs die Verarbeitung einer Adresse oder Zahl n nicht in konstanter Zeit möglich, da durch die binäre Repräsentation $\lceil \log_2 n \rceil$ Bits zur Speicherung benötigt würden. Eine bessere Annahme für die Laufzeit wäre daher $O(\log n)$.

Da diese Beschränkungen bzw. Randeffekte jedoch unabdingbar auftreten seien sie zum Zwecke der besseren Handhabe und Vergleichbarkeit mit den vorigen Laufzeitangaben vernachlässigt. Die dazu von mir getroffenen Annahmen sind:

1. Operationen auf nativen Datentypen oder Zeigern (d. h. Wertzuweisungen, o. ä.) haben konstante Laufzeit.

2. Die Operationen add, remove, contains bzw. containsKey und size auf den gehashten Datentypen des Java Collection Frameworks benötigen konstante Laufzeit (unter der Voraussetzung, dass die Hashfunktion die Objekte gleichmäßig über die einzelnen Buckets verteilt).

3. Mengenoperationen wie addAll, removeAll und retainAll benötigen lineare Laufzeit in der Größe des Parameters.

Zusätzlich werde ich nach der theoretischen Untersuchung Zeitmessungen der einzelnen Algorithmen für verschiedene Problemgrößen angeben, um die reale Laufzeit darzustellen. Diese Messungen wurden auf einem Testsystem mit AMD Athlon^{TM} XP Prozessor, 1200 MHz, 512 MB RAM und dem Java^{TM} 2 Runtime Environment, Standard Edition (build 1.4.2_03-b02) für Linux ermittelt und sind Mittelwerte aus mindestens 5 Messungen.

Die Transducer zur Messung des Laufzeitverhaltens werden durch die Klasse model.transducer.implementation.TransducerGenerator erzeugt. Diese Klasse erstellt einen zufälligen, nicht getrimmten Transducer aus einer Alphabetsgröße $|\Sigma|, |\Delta|$, einer Längenbegrenzung der Ausgaben $l_{\max} := \max\{l_{\text{out}}(t) | t \in E\}$, sowie einer Anzahl von Zustände $|Q|$ und Übergängen $|E|$. Dabei werden die Übergänge zufällig auf die $|Q|$ Zustände verteilt. Der verwendete Pseudo-Zufallszahlengenerator ist java.util.Random. Die Messreihen für die Diagramme finden sich im Anhang A.4.

6.2. Datenstrukturen

6.2.1. Interfaces

Zunächst einmal ist es sinnvoll, definierte Schnittstellen für die Datentypen zu schaffen, um die konkrete Implementierung von den darauf anwendbaren Algorithmen unabhängig zu halten. Hierzu kann in Java das native Sprachkonstrukt **interface** benutzt werden.

Zur Strukturierung der Algorithmen habe ich dazu zwei Interface-Pakete geschaffen (siehe Abbildung 6.1): Zum einen Transducer, State und Transition auf denen die Transducer-Algorithmen der Unterabschnitte 6.3.1–6.3.3 und die GUI aufsetzen. Und zum anderen DirectedGraph, Vertex und DirectedEdge für die Graphen-Algorithmen, welche als Grundlage für die oben genannten Transducer-Algorithmen herangezogen werden.

Die Schaffung eines weiteren Interface-Pakets DEA für die Interpretation der Datenstruktur als endlicher Automat (z. B. für die Minimierung) habe ich nicht umgesetzt, da die so gewonnene Struktur leider zu Problemen bei der Benennung der Methoden und Klassen führt sowie unverhältnismäßig großem Aufwand gegenübersteht.

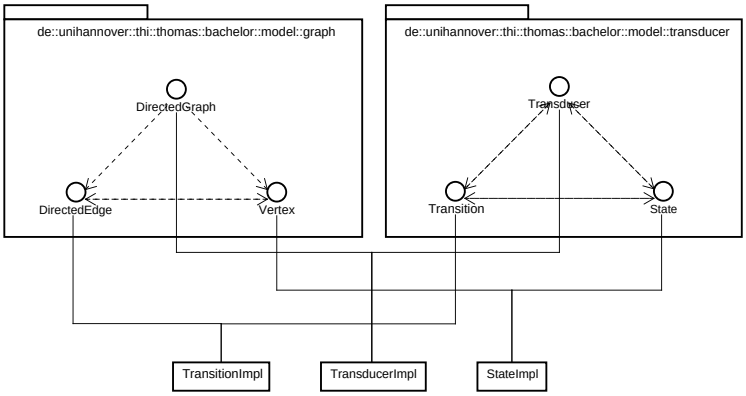


Abbildung 6.1.: Datentypen Interfaces

6.2.2. Umsetzung

Zur Realisierung der konkreten Datenstruktur für einen Transducer ist es sinnvoll, die gängigen Ansätze für Datenstrukturen gerichteter Graphen zu analysieren und entsprechend anzupassen. Drei gebräuchliche Ansätze für Graphen finden sich hierbei in [GT98]:

- die Übergangslisten-Struktur
- die Adjazenzlisten-Struktur
- die Adjazenzmatrix-Struktur

Die Übergangslisten-Struktur ist unter diesen die einfachste Repräsentation eines Graphen: als Knoten- und Kantenmenge, wobei nur letztere Verweise auf die Knoten als Beginn und Ende einer Kante enthält. Dies führt zu einer linearen Laufzeit, möchte man die ein- oder ausgehenden Kanten von Knoten erhalten.

Die Adjazenzmatrix-Struktur hingegen ist eine Realisierung der Adjazenzmatrix eines Graphen z. B. als boolesches Array. Der Nachteil dieser Struktur ist ihr quadratischer Speicherplatzbedarf und die quadratische Laufzeit von Einfüge- bzw. Löschoperationen von Zuständen.

Der von mir gewählte und erweiterte Ansatz ist die Adjazenzlisten-Struktur. Dieser ist eine Erweiterung der Übergangslisten-Struktur um Adjazenzlisten für jeden Knoten des Graphen, bei der der oben genannte

Nachteil nicht auftritt und alle Operationen in konstanter oder linearer Laufzeit realisierbar sind. Darüber hinaus lässt sich diese Ansatz gut in einer objekt-orientierten Sprache wie Java umsetzen.

Jeder Zustand besitzt also eine Menge an eingehenden Übergängen und eine Tabelle, in welcher die ausgehenden Übergänge nach Ausgabemarkierung ablegt sind, sowie eine boolesche Variable, welche angibt ob dies ein Endzustand ist. Jeder Übergang kennt wiederum seinen Start- und Zielzustand, sowie die eigene Ein- und Ausgabe. Da letztere eine Zeichenkette ist, welche während der Quasi-Determinierung eventuell mehrfach modifiziert wird, habe ich hier den Datentyp StringBuffer verwendet. Zusätzlich beinhalten beide Objekte noch einen Verweis auf den Transducer, um sie diesem eindeutig zuzuordnen. Der Transducer besitzt abgesehen von der Zustands- und Übergangsmenge zusätzlich noch je ein Ein- und Ausgabealphabet vom Typ model.Alphabet, einen Verweis auf einen Startzustand und eine initiale Ausgabe.

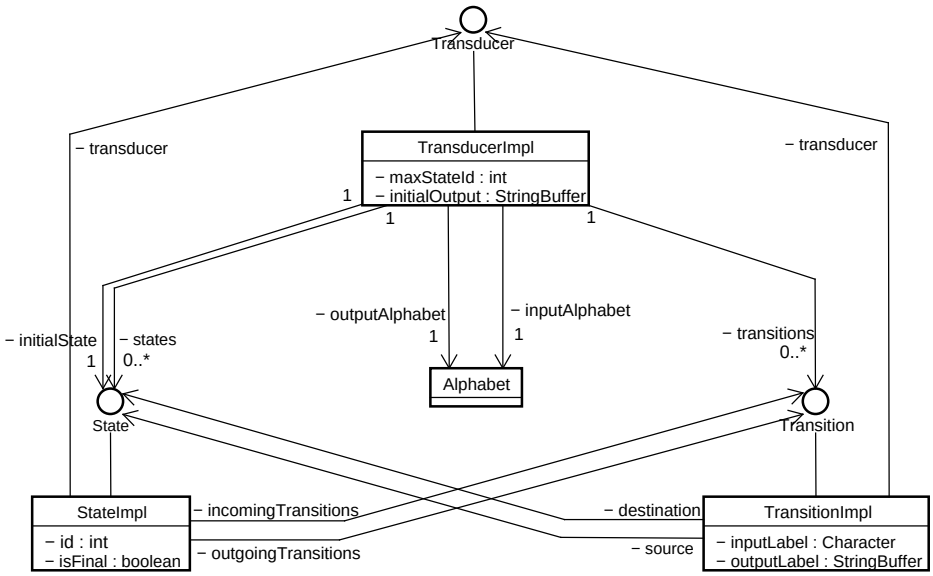


Abbildung 6.2.: Implementierung der Datenstruktur Transducer

Die darüber hinaus intern verwendeten Integer (id, maxStateId) dienen für die Erzeugung der Hashwerte, damit der Zugriff auf ein Objekt in Mengen in konstanter Zeit möglich ist, sowie der Kennzeichnung zwecks einer einfache Identifikation durch den Benutzer.

Die implementierenden Klassen befinden sich im Paket model.transducer.implementation. Deren Beziehungen untereinander sind in Abbildung 6.2 dargestellt. Die Laufzeiten der implementierten Methoden finden sich, nach Interfaces geordnet, im Anhang A.3.

6.3. Algorithmen

In diesem Abschnitt werden nun die Implementierungen der erwähnten Algorithmen in Java vorgestellt. Diese Algorithmen befinden sich im Paket model.transducer.algorithm und implementieren das Interface TransducerAlgorithm, damit eine einheitliche Schnittstelle für den Zugriff existiert (siehe Abbildung 6.3).

Innerhalb des Interfaces sind die Methoden execute(), execute(Transducer transducer) und execute(Transducer transducer1, Transducer transducer2) die Einstiegspunkte für den implementierten Algorithmus. Die Auskunft, welche der beiden Funktionen auszuführen ist, wird durch getCardinality() geliefert: der Rückgabewert bestimmt die Anzahl der benötigten Parameter für die execute-Methode. Die weiteren Methoden dienen zur Beschreibung des implementierten Algorithmus bzw. der Darstellung in einer graphischen Benutzeroberfläche.

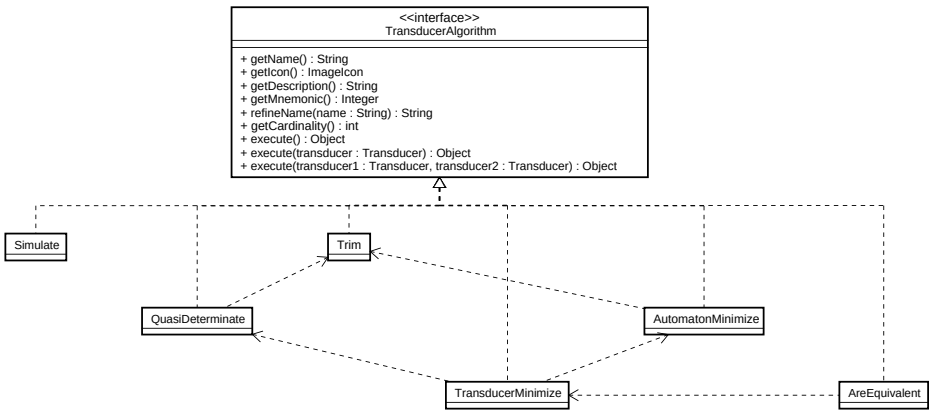


Abbildung 6.3.: Klassendiagramm des Pakets model.transducer.algorithm

6.3.1. Quasi-Determinierung

Die Implementierung der Quasi-Determinierung ist in der Klasse model.transducer.algorithm.QuasiDeterminate zu finden. Der eigentliche Algorithmus verteilt sich dabei im Kern auf 3 Methoden, welche in den folgenden Listings 6.1 und 6.2 abgedruckt sind. Die Klasse besitzt zwei private Variablen n und f, welche die Variablen $N[q]$ und F aus Listing 4.1 nachbilden. Zusätzlich werden die Methoden **void** increaseN(State s), **void** decreaseN(State s) und **boolean** isNzero(State s) definiert, welche nur den Umgang mit der Variablen n kapseln und erleichtern. Alle drei Funktionen arbeiten in konstanter Zeit $O(1)$.

Das Listing 6.1 zeigt zunächst die Methode computeLCP(State state), welche die Berechnung der Funktion $\mathrm{lcp}(T_{\mathrm{out}}, q)$ aus Listing 4.1 realisiert. Die Methode lcp(String [] strings) ist eine Hilfsfunktion, die für die eigentliche Berechnung des längsten gemeinsamen Präfixes eines Arrays von Strings zuständig ist.

```
1  private String lcp(String [] strings) {
2    if (strings.length == 0) return new String();
3    else if (strings.length == 1) return strings[0];
4    else {
5      int min = strings[0].length();
6      for (int i = 1; i < strings.length; i++)
7        min = Math.min(min, strings[i].length());
8
9      boolean equal = true;
10     int indexLCP = min;
11     for (int j = 0; j < min; j++) {
12       for (int i = 1; i < strings.length; i++) {
13         equal = equal && (strings[0].charAt(j) == strings[i].charAt(j));
14         if (!equal) indexLCP = Math.min(indexLCP, j);
15       }
16     }
17     return strings[0].substring(0, indexLCP);
18   }
19 }
20
21 private String computeLCP(State state) throws TransducerException {
22   if (currentState.isFinal()) {
23     f.add(currentState);
24     return "";
```

```
25    }
26
27    // Collect all outgoing output labels
28    Set strings = new HashSet();
29    Iterator iter = state.getOutTransitions().values().iterator();
30    while (iter.hasNext())
31      strings.add(((Transition) iter.next()).getOutputLabel());
32
33    // Compute LCP
34    String stringArray[] = new String[strings.size()];
35    strings.toArray(stringArray);
36    String lcp = lcp(stringArray);
37
38    // Remove LCP from outgoing transitions
39    iter = state.getOutTransitions().values().iterator();
40    while (iter.hasNext()) {
41      Transition t = (Transition) iter.next();
42      t.deleteOutputLabelSuffix(lcp.length());
43      if (t.getOutputLabel().length() == 0) increaseN(state);
44    }
45
46    // Correct F
47    if (isNZero(currentState)) f.add(currentState);
48
49    return lcp;
50  }
```

Listing 6.1: Implementierung der Funktion $\text{lcp}(T_{\text{out}}, q)$

Die Laufzeit der Methode lcp(String[] strings) ist $O((n-1)(\pi+1))$, wobei π die Länge des längsten gemeinsamen Präfixes und n die Größe des Arrays angebe. Darauf aufbauend folgt für die Methode computeLCP(State state) für die Eingabe eines Zustands q eine Laufzeit von $O(|P(q)| \cdot |Trans(q)|)$ wie in Unterabschnitt 4.1.4 angegeben.

In Listing 6.2 ist nun die Umsetzung des Algorithmus aus Listing 4.1 angegeben. Die Laufzeit ergibt sich wieder entsprechend der Argumentation in Unterabschnitt 4.1.4 zu $O(|Q| + |E| \cdot (P_{\max} + 1))$.

```
1  public Object execute(Transducer transducer) throws AlgorithmException{
2    try{
3      Transducer quasidetTransducer = (Transducer)new Trim().execute(
           transducer);
```

```
4       if (quasidetTransducer.getNumberOfStates() > 0){
5         LinkedList sorting = (LinkedList) new TopologicalSortSCCs().execute(
6             quasidetTransducer.toGraph(),
7             quasidetTransducer.getInitialState().toVertex());
8         while (!sorting.isEmpty()) {
9           Set scc = (Set) sorting.removeLast();
10          n.clear();
11          f.clear();
12          queue.clear();
13          queue.addLast(scc.iterator().next());
14          while (!queue.isEmpty()) {
15            State currentState = (State) queue.removeFirst();
16            String pi = computeLCP(currentState);
17            if (currentState == quasidetTransducer.getInitialState())
18              quasidetTransducer.appendToInitialOutput(pi);
19
20            Iterator transIter = currentState.getInTransitions().iterator();
21            while (transIter.hasNext()) {
22              Transition t = (Transition) transIter.next();
23              if (pi.length() > 0) {
24                if (scc.contains(t.getSource())
25                    && !isNZero(t.getSource())
26                    && t.getOutputLabel().equals("")
27                    && !f.contains(t.getSource()))
28                  decreaseN(t.getSource());
29                t.appendToOutput(pi);
30              }
31              if (!queue.contains(t.getSource())
32                  && isNZero(t.getSource())
33                  && !f.contains(t.getSource()))
34                queue.addLast(t.getSource());
35            }
36          }
37        }
38      }
39      return quasidetTransducer;
40    } catch (TransducerException e) {
41      throw new AlgorithmException("Error in Quasi-Determinization", e);
42    }
43  }
```

Listing 6.2: Implementierung des Algorithmus zur Quasi-Determinierung

Laufzeitmessung

Zur Untersuchung des reales Laufzeitverhaltens bietet es sich an, die Einflussgrößen $|Q|$, $|E|$ und $P_{\max}$ getrennt zu untersuchen. Da die Quasi-Determinierung jedoch auf getrimmten Transducern arbeitet, ist für die zufällig generierten Transducer eine Korrelation zwischen $|Q|$ und $|E|$ zu erwarten: Für einen Quotienten $\frac{|Q|}{|E|}$ nahe 1 ist die Wahrscheinlichkeit, dass der generierte Transducer zusammenhängend ist sehr gering, daher wird nach dem Trimmen die Anzahl der Zustände wahrscheinlich erheblich kleiner sein als die Vorgabe für $|Q|$. Dies führt zu dem sprunghaften Anstieg in folgendem Diagramm 6.4 im Bereich $\frac{|Q|}{|E|} < 3$. Für den Bereich $\frac{|Q|}{|E|} > 3$ kann dann das erwartete lineare Laufzeitverhalten der Quasi-Determinierung abgelesen werden.

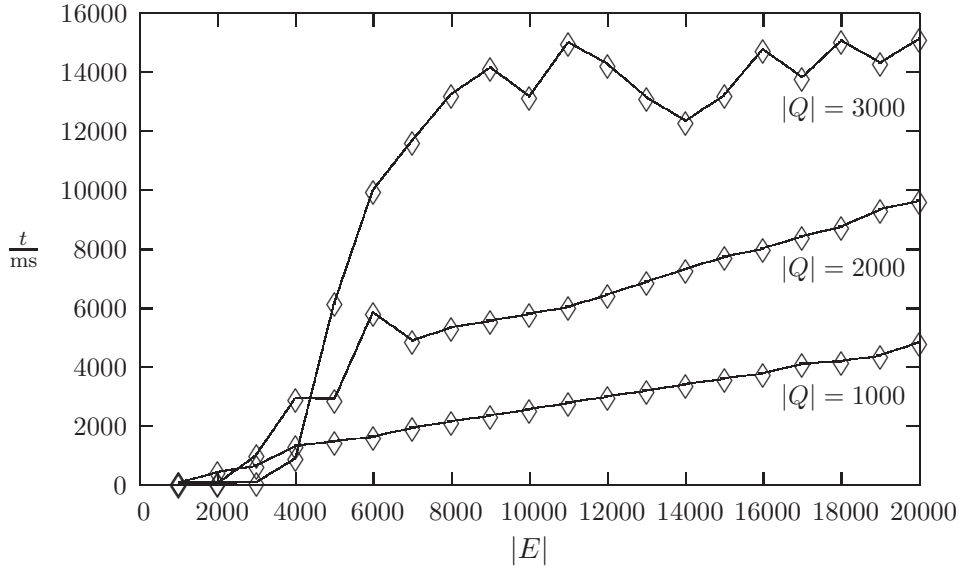


Abbildung 6.4.: Laufzeit der Quasi-Determinierung in Abhängigkeit von $|E|$. Übrige Parameter sind $|\Sigma| = |\Delta| = 26$ und $l_{\max} = 3$, siehe Messreihe A.8

Für große $|E|$ ist im Diagramm 6.4 auch erkennbar, dass die Messreihen in ungefähr gleichem Abstand parallel verlaufen. Die Quasi-Determinierung also auch linear in $|Q|$ arbeitet.

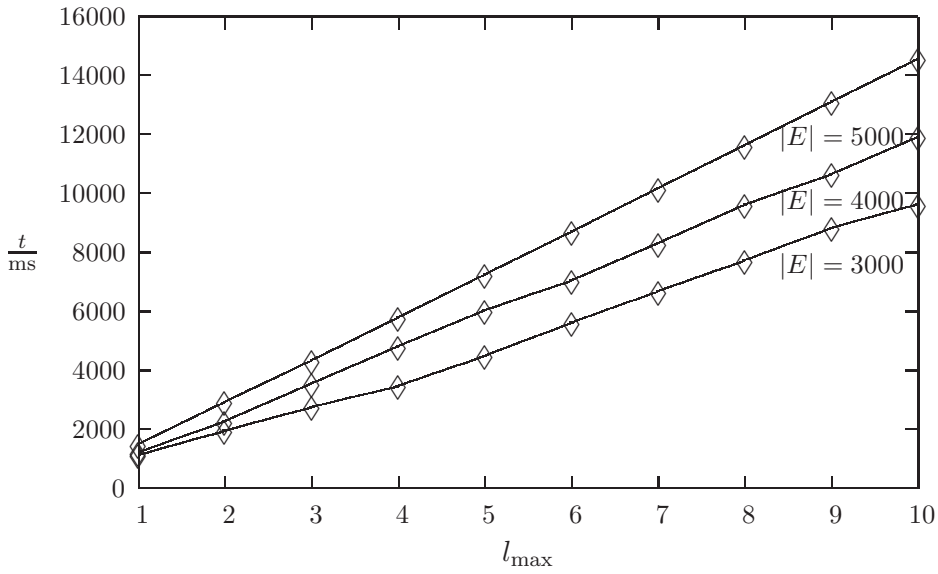


Abbildung 6.5.: Laufzeit der Quasi-Determinierung in Abhängigkeit von $l_{\max}$. Übrige Parameter sind $|\Sigma| = |\Delta| = 26$ und $|Q| = 1000$, siehe Messreihe A.8

In Diagramm 6.5 ist nun die Laufzeit in Abhängigkeit von der maximalen Länge der Ausgabemarkierungen $l_{\max} := \max\{|l_{\text{out}}(t)| \mid t \in E\}$ dargestellt. Es ist gut erkennbar, dass die Quasi-Determinierung linear in der maximalen Länge der Ausgabemarkierungen $l_{\max}$ und damit in der maximalen Länge des Längsten der längsten gemeinsamen Präfixe $P_{\max}$ arbeitet und multiplikativ mit $|E|$ zusammenhängt.

6.3.2. Minimierung

Die Minimierung von Transducern geschieht nach Kapitel 4 in zwei Schritten: der Quasi-Determinierung und einer anschließenden Automaten-Minimierung. Daher sei an dieser Stelle zuerst die Implementierung der Automaten-Minierung nach Hopcroft in der Klasse AutomatonMinimize erläutert und anschließend kurz auf die Minimierung für Transducer in der Klasse TransducerMinimize eingegangen. Beide Klassen befinden sich im Paket model.transducer.algorithm.

Automaten-Minimierung

Die Implementierung der Automaten-Minimierung nach Hopcroft verteilt auf mehrere Methoden. Daher werde ich von der execute-Methode aus vorgehen und absteigend die aufgerufenen Hilfsmethoden erläutern und ggf. anfügen:

```
1  public Object execute(Transducer transducer) throws AlgorithmException {
2    try {
3      Transducer trimmedTransducer = (Transducer)new Trim().execute(
           transducer);
4      Set partition = computeCoartestPartition(trimmedTransducer);
5      return constructMinimizedTransducer(trimmedTransducer, partition);
6    } catch (TransducerException e) {
7      throw new AlgorithmException("Error in Automata−Minimization:",e);
8    }
9  }
```

Listing 6.3: Methode AutomatonMinimize.execute(Transducer)

Wie man in Listing 6.3 erkennt, wird zu Beginn der dem im Parameter übergebenen Transducer mittels der Klasse model.transducer.algorithm.Trim getrimmt (siehe dazu Anhang A.1). Anschließend wird die Berechnung an zwei private Methoden delegiert. Die Methode computeCoartestPartition(Transducer transducer) berechnet die gröbste Partitionierung der Zustände in Mengen äquivalenter Zustände anhand des Partitionierungs-Algorithmus nach Hopcroft. Die Methode constructMinimizedTransducer(Transducer transducer, Set blocks) konstruiert aus diesen Mengen von Zuständen nun den minimierten Transducer. Da die besagte Konstruktion relativ einfach in linearer Zeit der Größe des Eingabetransducers geschehen kann, sei dieser Teil zugunsten der Erklärung des Partitionierungs-Algorithmus vernachlässigt.

Die Methode computeCoartestPartition(Transducer transducer) beschreibt den Kern des Partitionierungs-Algorithmus, wie er auch in [AHU74, S. 157ff] zu finden ist. Für die Anwendung der Minimierung werden dazu Paare (Q', l) als Blöcke angesehen, wobei $Q' \subseteq Q$ die Menge der Zustände und $l \in (\Sigma \times \Delta)$ die zu untersuchende Übergangsmarkierung des Blocks ist.

```
1  private Set computeCoartestPartition(Transducer transducer) {
2    Queue queue = getInitialBlocks(transducer);
```

```
3
4     while (!queue.isEmpty()) {
5       Block block = queue.dequeue();
6
7       // determine concerned state sets and compute intersections
8       Map intersections = getIntersectingStateSets(block);
9
10      // split blocks
11      Iterator iter = intersections.keySet().iterator();
12      while (iter.hasNext()) {
13        StateSet cutStates = (StateSet) iter.next();
14        Set intersection = (Set) intersections.get(cutStates);
15        cutStates.removeAll(intersection);
16        coartestPartition.add(intersection);
17        Set states;
18        if (cutStates.enqueuedLabels.contains(block.labelNr) || cutStates.size
              () > intersection.size())
19          states = intersection;
20        else
21          states = cutStates;
22
23        // enqueue
24        Set inverseLabels = getInLabels(states);
25        Iterator subiter = inverseLabels.iterator();
26        while (subiter.hasNext()) {
27          Integer labelNr = (Integer) subiter.next();
28          Block newBlock = new Block(intersection, labelNr);
29          queue.enqueue(newBlock);
30        }
31
32        // update inBlock
33        Iterator stateIter = states.iterator();
34        while (stateIter.hasNext()){
35            State state = (State) stateIter.next();
36            state2Partition.put(state,states);
37        }
38      }
39    }
40    return coartestPartition;
41  }
```

Listing 6.4: Methode AutomatonMinimize.computeBlocks(Transducer)

Zunächst werden durch die Methode getInitialBlocks (Transducer transducer) die initiale Queue konstruiert und die Variablen state2Partition, label2int, int2label und coartestPartition initialisiert:

- coartestParitition ist eine Menge von Mengen und bildet die aktuelle Partitionierung der Zustände.
- state2Partition ist eine Tabellen, die zu jedem Zustand den ihn enthaltenden Block bestimmt.
- label2int und int2label sind Tabellen, welche eine bijektive Abbildung der Übergangsmarkierungen auf Integer-Zahlen bereitstellen, damit deren Vergleiche in konstanter Zeit möglich werden.

Da zur Initialisierung der oben genannten Variablen der Transducer komplett durchlaufen werden muss und die Übergangsmarkierungen zur Identifikation mit Integer-Zahlen in einer geeigneten Datenstruktur abgelegt werden müssen, folgt für getInitialBlocks (Transducer transducer) eine Laufzeit von $O(S + |E| + |Q|)$, wobei S die Summe der Längen der Ausgabemarkierungen ist. Anschließend wird die initiale Queue nun in der **while**-Schleife (Zeilen 4 – 39) abgearbeitet. Der Rumpf dieser Schleife kann dabei in Zeit $O(|E|)$ abgearbeitet werden:

Mittels der Methode getIntersectingStateSets (block) werden die Zustandsmengen bestimmt, die eine echt Schnittmenge mit der Menge der Zustände besitzen sind, die Übergänge in den Block block besitzen. Die Methode ist in Listing 6.5 angeführt und wird später noch näher erläutert. Zunächst sei jedoch die Laufzeit von der inneren **while**-Schleife in den Zeilen 12 – 38 erklärt:

Die in intersections ermittelten Zustandsmengen werden in Zeile 15 aufgespalten. Diese Operation kann in Zeit $O(|\,\text{intersection}\,|)$ durchgeführt werden. Anschließend wird bestimmt, welche der beiden neuen Zustandsmenge wieder in die Queue eingereiht werden muss. Für die wieder in die Queue einzureihende Zustandsmenge werden nun in Zeile 24 durch die Methode getInLabels(Block block) alle eingehenden Markierungen gesammelt und als neue Blöcke der Queue hinzugefügt. Das sammeln der eingehenden Markierungen und das Einreihen in die Queue in der **while**-Schleife der Zeilen 26 – 30 kann in Zeit $O(\sum_{q \in \text{states}} |Trans^T[q]|)$ geschehen. Die Aktualisierung der Tabelle inBlock in den Zeile 34 – 37 benötigt eine Laufzeit von $O(|\,\text{states}\,|)$.

Somit folgt für einen Durchlauf des Codes in der **while**-Schleife der Zeilen 12 – 38 eine Laufzeit von $O(|\text{cutStates}| + \sum_{q \in \text{cutStates}} |Trans^T[q]|)$. Da jeder Zustand in genau einer Zustandsmenge liegt, kann intersections nur disjunkte Zustandsmengen enthalten. Also kann in der **while**-Schleife in den Zeilen 12 – 38 jeder Zustand maximal einmal behandelt werden und es ergibt sich die Laufzeit von $O(|Q| + |E|) = O(|E|)$, da der Partitionierungs-Algorithmus einen getrimmten Transducer als Eingabe übergeben bekommt.

Im folgenden Listing 6.5 ist nun der oben noch nicht berücksichtige Methodenaufruf getIntersectingBlocks(block) aufgelistet. Diese Methode bestimmt alle Blöcke und zugehörigen Schnittmengen, die den Bedingungen des Originalalgorithmus in Zeile 6 entsprechen: alle Blöcke, deren Zustandsmenge eine echte Teilmenge der Zustände sind, die Übergänge in Zustände des Blocks block enthalten.

```
1  private Map getIntersectingStateSets(Block block) {
2    Map intersections = new HashMap();
3    Iterator iter = getLinkedStates(block).iterator();
4    while (iter.hasNext()) {
5      State state = (State) iter.next();
6      Set linkedStateSet = (Set) state2Partition.get(state);
7      if (! intersections.containsKey(linkedStateSet)) intersections.put(
           linkedStateSet, new StateSet());
8      ((Set) intersections.get(linkedStateSet)).add(state);
9
10     if (((Set) intersections.get(linkedStateSet)).size() >= linkedStateSet.
           size())
11       intersections.remove(linkedStateSet);
12   }
13   return intersections;
14 }
```

Listing 6.5: Methode AutomatonMinimize.getIntersectingBlocks(Transducer)

Zur Bestimmung der betroffenen Blöcke wird die Variable inBlock ausgenutzt. Auf diese Weise ist das Bestimmen des Blocks, in dem ein Zustand liegt, in konstanter Zeit möglich und die Zeilen 4–12 können in Zeit $O(\sum_{q \in \text{block.states}} |Trans^T[q]|)$ bestimmt werden.

Insgesamt folgt somit für die Laufzeit eines Durchlaufs der **while**-Schleife in den Zeilen 4 – 39 eine Komplexität von

$$O(\sum_{q \in Q} |Trans^T[q]| + |E|) = O(|E|)$$

Wie oft kann nun ein Zustand s, welcher nicht innerhalb eines Blocks in der Queue wartet, in selbige eingereiht werden? Dies kann nur in Zeile 29 geschehen, nachdem durch die Bedingung in Zeile 18 die kleinere der beiden Schnittmengen ausgewählt worden ist. (cutBlock.inQueue ist **false**, da die aufzuspaltende Zustandsmenge und damit der Zustand s sonst schon in der Queue warten würde!)

Somit enthält der Block mit s höchstens die Hälfte der Zustände, die er enthielt, als er das letzte Mal in die Queue eingereiht wurde. Da ein Block mit nur einem Zustand nicht mehr aufgespalten werden kann und damit nicht mehr in die Queue eingereiht wird, kann jeder Block nur $\log |Q| + 1$ mal in die Queue gesetzt werden. Darüber hinaus enthalten die Blöcke lediglich Zeiger auf Zustandmengen und keine Kopien, folglich werden durch einen Teilvorgang bis zu $|E|$ Blöcke modifiziert bzw. verkleinert. Somit kann auch jeder Zustand nur $O(\log |Q|)$ mal in die Queue gesetzt werden und die **while**-Schleife in den Zeilen 4 – 39 terminiert nach maximal $O(\log |Q|)$ Durchläufen. Also folgt für den gesamten Algorithmus (mitsamt der Konstruktion des minimalen Transducers) eine Laufzeitkomplexität von

$$O(S + |Q| + |E| \cdot \log |Q|) = O(S + |E| \cdot \log |Q|)$$

Transducer-Minimierung

Der Algorithmus für die Transducer-Minimierung ist in der Klasse model.transducer.algorithm.TransducerMinimize abgelegt. Der Algorithmus besteht lediglich aus einer Komposition des Quasi-Determinierung und der Automaten-Minimierung: zunächst instanziiert er den Quasi-Determinierungs- und den Automaten-Minimierungs-Algorithmus und wendet sie schließlich nacheinander auf den Eingabetransducer an.

Die Laufzeit dieser Minimierung setzt sich aus den Laufzeiten der beiden Algorithmen zusammen und ergibt sich zu $O(S+|Q|+|E|\cdot(P_{\max}+\log |Q|))$, wobei S wiederum die Summe der Längen der Eingabemarkierungen angibt.

Laufzeitmessung

Untersucht man das Laufzeitverhalten der Automaten-Minimierung, so interessiert vor allem der Kern des Algorithmus, welcher in $O(|E| \cdot \log |Q)$ laufen sollte. Diagramm 6.6 zeigt die Abhängigkeit der Laufzeit der Automaten-Minimierung von den Anzahl der Übergänge. Wie man gut erkennen kann, ist für den Bereich $\frac{|E|}{|Q|} \geq 3$ der Anstieg der Laufzeit in Übereinstimmung mit den theoretischen Ergebnissen annähernd linear. Darüber hinaus hängt die Laufzeit multiplikativ von $|Q|$ ab, wobei der der Laufzeitzuwachs von einer Funktion $f(|Q|)$ mit $f(|Q|) < |Q|$ abhängt.

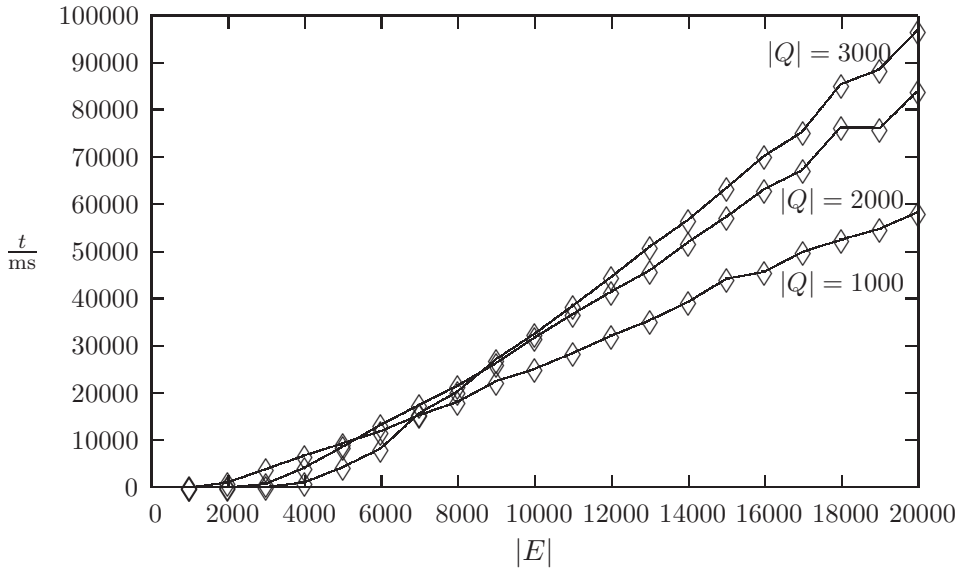


Abbildung 6.6.: Laufzeit der Automaten-Minimierung in Abhängigkeit von $|E|$ für $|Q| = \{1000, 2000, 3000\}$, $|\Sigma| = |\Delta| = 26$ und $l_{\max} = 3$, siehe Messreihe A.9

In Diagramm 6.7 ist die Abhängigkeit der Laufzeit von $\log |Q|$ dargestellt. Wie man erkennen kann, ist auch dieser Anstieg ungefähr linear, also die Laufzeit abhängig von $\log |Q|$. Das leichte Abflachen der Messreihe für große $|Q|$ ist durch den den Quotienten $\frac{|E|}{|Q|}$ zu erklären, welcher für eine anwachsenden Zustandsmenge schrumpft und daher die Anzahl der

Zustände im getrimmten Transducer, auf dem die Automaten-Minimierung arbeitet, statistisch gesehen abnimmt.

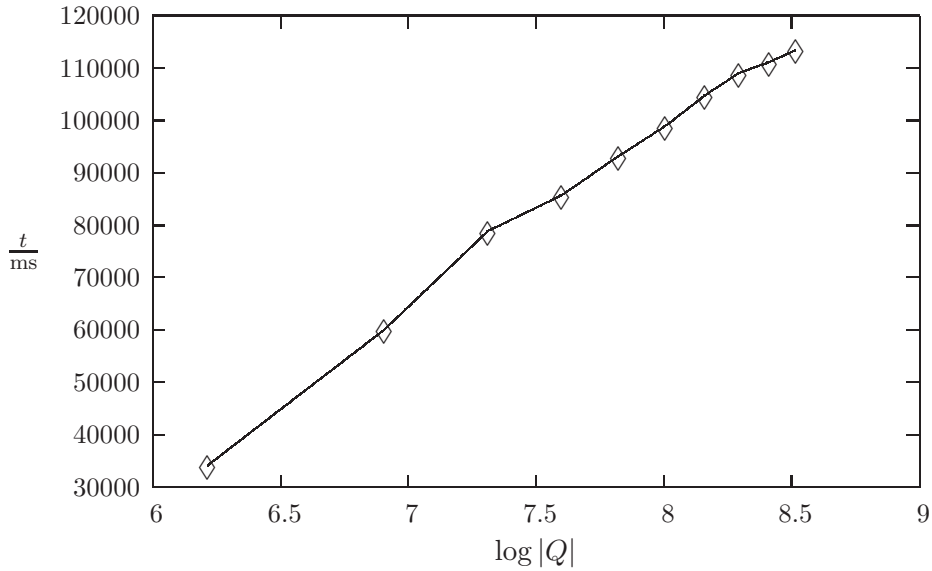


Abbildung 6.7.: Laufzeit der Automaten-Minimierung in Abhängigkeit von $\log |Q|$ für $|E| = 20000$, $|\Sigma| = |\Delta| = 26$ und $l_{\max} = 3$, , siehe Messreihe A.10

Die Untersuchung der Transducer-Minimierung kann an dieser Stelle übergangen werden, da diese lediglich eine Komposition der Quasi-Determinierung und der Automaten-Minimierung ist. Folglich ergibt sich die Laufzeit ungefähr als Summe der Laufzeiten dieser beiden Algorithmen.

6.3.3. Äquivalenztest

Der Äquivalenztest für Transducer ist als Algorithmus in der Klasse model.transducer.algorithm.AreEquivalent abgelegt. Da sich diese Implementierung, wie auch die der Quasi-Determinierung, eng an die in Kapitel 5 dargelegte Beschreibung anlehnt, seien hier in gebotener Kürze die beiden wichtigsten Methoden in den Listings 6.6 und 6.7 vorgestellt:

Die Methode execute(Transducer transducer1, Transducer transducer2) ist eine Implementierung der Funktion AreEquivalent. Die Methode

dualDFSvisit(State state1, State state2) bildet die Funktion AreIsomorph aus Listing 5.1 nach.

```
1  private boolean dualDFSvisit(State state1, State state2) {
2    boolean isIsomorph = true;
3    markedStates1.add(state1);
4    markedStates2.add(state2);
5    states1Tostates2.put(state1, state2);
6
7    if ((state1.isFinal() != state2.isFinal()) && (state1.getOutDegree() !=
         state2.getOutDegree())){
8      isIsomorph = false;
9    }else{
10     Map transitions1 = state1.getOutTransitions();
11     Map transitions2 = state2.getOutTransitions();
12     Iterator iter = transitions1.values().iterator();
13     while (iter.hasNext()) {
14       Transition t1 = (Transition) iter.next();
15       Transition t2 = null;
16       if (!transitions2.containsKey(t1.getInputLabel())){
17         isIsomorph = false;
18       }else{
19         t2 = (Transition) transitions2.get(t1.getInputLabel());
20         if (!t1.getOutputLabel().equals(t2.getOutputLabel())) {
21           isIsomorph = false;
22         }else{
23           State next1 = t1.getDestination();
24           State next2 = t2.getDestination();
25           if (markedStates1.contains(next1)!=markedStates2.contains(next2))
26             isIsomorph = false;
27           else if (markedStates1.contains(next1))
28             isIsomorph = isIsomorph && (states1Tostates2.get(next1) ==
                   next2);
29           else
30             isIsomorph = isIsomorph && dualDFSvisit(next1, next2);
31         }
32       }
33     }
34   }
35   return isIsomorph;
36 }
```

Listing 6.6: Methode AreEquivalent.dualDFSvisit(State, State)

Die Funktion dualDFSvisit kann nur $\min\{|Q_1|, |Q_2|\}$ mal aufgerufen werden, da bei jedem Aufruf in jedem Transducer ein weiterer Zustand markiert wird (Zeilen 3 und 4). Weiterhin ist durch die Auswahl der Kanten in den Zeilen 10 und 11 sichergestellt, dass jede Kante maximal einmal beschritten werden kann. Also geschieht der Vergleich von Markierungen in Zeile 20 pro Übergang ebenfalls nur einmal; die Laufzeit dieses Vergleichs ist linear in der Länge der kürzeren Ausgabemarkierung beschränkt. Da ein Durchlauf der Methode bis auf den Vergleich der Markierungen in konstanter Zeit geschehen kann, ergibt sich die Summe der Laufzeiten der dualDFSvisit-Methode analog zu Abschnitt 5.3 als

$$O(S_{\min} + \min\{|Q_1|, |Q_2|\} + \min\{|E_1|, |E_2|\}),$$

wobei $S_{\min}$ wie in Abschnitt 5.3 die Summe der Längen des jeweils Kürzeren von zwei korresponierenden Übergangsmarkierungen in den beiden Transducern ist.

```
1  public Object execute(Transducer transducer1, Transducer transducer2)
       throws AlgorithmException {
2    states1Tostates2.clear();
3    markedStates1.clear();
4    markedStates2.clear();
5
6    TransducerMinimize m = new TransducerMinimize();
7    Transducer minT1 = (Transducer) m.execute(transducer1);
8    Transducer minT2 = (Transducer) m.execute(transducer2);
9    State initial1 = minT1.getInitialState();
10   State initial2 = minT2.getInitialState();
11
12   if (!minT1.getInputAlphabet().equals(minT2.getInputAlphabet())
13       || !minT1.getOutputAlphabet().equals(minT2.getOutputAlphabet())
14       || ((initial1 != null) && (initial2 == null))
15       || ((initial1 == null) && (initial2 != null)))
16     return Boolean.FALSE;
17   else if ((initial1 == null) && (initial2 == null))
18     return Boolean.TRUE;
19   else
20     return new Boolean(minT1.getInitialOutput().equals(minT2.
         getInitialOutput()) && dualDFSvisit(initial1,initial2));
21 }
```

Listing 6.7: Die execute-Methode des Äquivalenztests

Die Laufzeit der execute-Methode wird ihrerseits nun durch die Minimierung der Transducer in den Zeilen 7 und 8 sowie den Vergleich der beiden minimierten Transducer in Zeile 20 geprägt:

Seien die beiden Eingabetransducer gegeben durch $T_1 = (Q_1, \Sigma_1, \Delta_1, \delta_1, \sigma_1, \lambda_1, q_{1,0}, F_1)$ und $T_2 = (Q_2, \Sigma_2, \Delta_2, \delta_2, \sigma_2, \lambda_2, q_{2,0}, F_2)$. Die Minimierung benötigt nach dem voran gegangenen Abschnitt 6.3.2 eine Laufzeit von $O(S_i + |Q_i| + |E_i| \cdot (P_{i,\max} + \log |Q_i|))$ für den Eingabetransducer T_i, $i = 1, 2$. Der anschließende Vergleich ist in Zeit $O(S_{\min} + \min\{|\Sigma_1|, |\Sigma_2|\} + \min\{|\Delta_1|, |\Delta_2|\} + \min\{|Q_1|, |Q_2|\} + \min\{|E_1|, |E_2|\})$ möglich: die Gleichheit der Ein- und Ausgabealphabete muss hier explizit geprüft werden. Somit folgt für den Äquivalenztest-Algorithmus eine Laufzeit von

$$O(S + |\Sigma| + |\Delta| + |Q| + |E| \cdot (P_{\max} + \log |Q|))$$

wobei $S = S_1 + S_2$, $|\Sigma| = |\Sigma_1| + |\Sigma_2|$, $|\Delta| = |\Delta_1| + |\Delta_2|$, $|Q| = |Q_1| + |Q_2|$, $|E| = |E_1| + |E_2|$ und $P_{\max} = P_{1,\max} + P_{2,\max}$.

Laufzeitverhalten

Die Messung eines realen Laufzeitverhaltens der Äquivalenztest-Algorithmus ist auf Basis der zufällig generierten Transducer nur schwer möglich, da allein die Wahrscheinlichkeit für gleiche Ausgangsgerade der Startzustände $q_{1,0}$ und $q_{2,0}$ durch

$$P(|Trans[q_{1,0}]| = |Trans[q_{2,0}|) = \sum_{i=1}^{|E|} \left(\binom{|E|}{i} \left(\frac{1}{|Q|} \right)^i \left(\frac{|Q| - 1}{|Q|} \right)^{|E|-i} \right)^2$$

gegeben ist und für einen steigenden Quotienten $\frac{|E|}{|Q|}$ gegen 0 geht. Zum Beispiel ergibt sich für Werte $|E| = 3000$ und $|Q| = 1000$ eine Wahrscheinlichkeit von ca. $0,16$. Folglich ist die in Diagramm 6.8 dargestellt Laufzeit des Äquivalenztest-Algorithmus hauptsächlich durch die Laufzeit der Transducer-Minimierung der beiden Eingabetransducer geprägt.

Zur Erklärung der Formel: Für einen zufällig generierten Transducer ist die Wahrscheinlichkeit, dass ein bestimmter Zustand (hier: der Startzustand) Ausgangsgrad i hat binomial verteilt mit Parametern $|E|$ und $\frac{1}{|Q|}$. Die Wahrscheinlichkeit, dass die Startzustände zweier stochastisch unabhängig generierter Transducer beide Ausgangsgrad i besitzen ist das

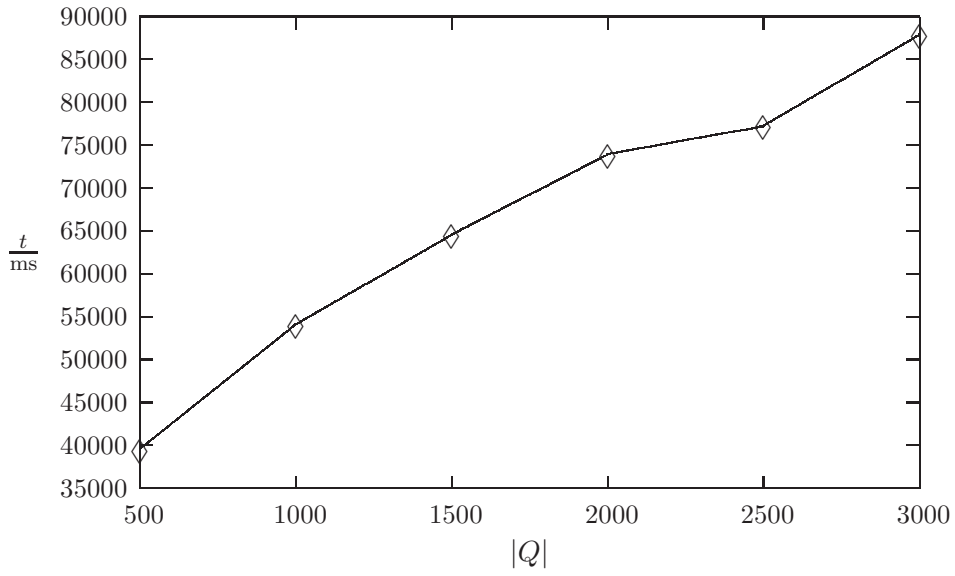


Abbildung 6.8.: Laufzeit des Äquivalenztests für $|\Sigma| = |\Delta| = 26$, $|E| = 10000$ und $l_{\max} = 3$, siehe Messreihe A.11

Quadrat dieses Werts:

$$\left(\binom{|E|}{i}\left(\frac{1}{|Q|}\right)^{i}\left(1-\frac{1}{|Q|}\right)^{|E|-i}\right)^{2}$$

Da die Startzustände Ausgangsgrade $i, 0 \leq i \leq |E|$ besitzen können erhält man die Wahrscheinlichkeit für gleiche Ausgangsgrade zwei bestimmter Zustände durch die Summe über all dieser Wahrscheinlichkeiten.

7. Bewertung, Ausblick

In dieser Bachelorarbeit wurde ein Algorithmus zur Minimierung und ein Äquivalenztest von Transducern vorgestellt.

Der Minimierungs-Algorithmus arbeitet in Zeit $O(S+|Q|+|E|\cdot(P_{\max}+\log|Q|))$ (siehe Abschnitt 4.4) und ist damit effizient. Der erste Schritt dieser Minimierung, die Quasi-Determinierung, kann dabei auch für die Lösung des verwandten single-source shortest-path Problem verwendet werden. Dies ist jedoch hier nicht näher erläutert worden. Ebenso wurde das Gebiet der p-subsequentiellen Transducer nur angeschnitten.

Der hier vorgestellte Äquivalenztest setzt auf dem Minimierungs-Algorithmus für Transducer auf und arbeitet in Zeit $O(S_1 + S_2 + |Q_1| + |Q_2| + (|E_1| + |E_2|) \cdot (P_{\max} + \log(|Q_1| + |Q_2|)))$ (siehe Abschnitt 5.3).

Die Algorithmen wurden in Java mitsamt einer graphischen Benutzeroberfläche implementiert und zeigten in Messungen für Transducer mit bis zu $|E| = 20.000$ Übergängen ein Verhalten nahe den theoretischen Angaben. Aufgrund der Dauer der Messungen war mir die Verwendung größerer Transducer für diese leider nicht möglich. Darüber hinaus habe ich die Algorithmen objekt-orientiert implementiert, was gut lesbaren Quellcode führt. Dies resultiert jedoch einer langsameren Laufzeit, als sie z. B. für eine Implementierung zu erreichen wäre, die die Objekte auf Zahlen abbildet.

A. Anhang

A.1. Trimmen von endlichen Automaten

Das Trimmen von endlichen Automaten bezeichnet ein Verfahren, welches alle Zustände entfernt, die entweder nicht vom Startzustand aus erreichbar sind oder keinen Pfad in einen Endzustand besitzen. Dieses Verfahren lässt sich problemlos auf endliche Transducer übertragen und funktioniert wie folgt:

1. Man durchläuft dazu den Transducer zunächst in einem vollständigen Suchverfahren (d. h. eines, welches alle möglichen Ergebnisse liefert; hier: alle möglichen Zustände besucht) vom Startzustand aus und markiert alle besuchten Zustände grün

2. Im nächsten Schritt startet man eine parallele Suche von allen im ersten Schritt markierten Endzuständen des Transducers und markiert jeden besuchten Zustand rot.

3. Nun entfernt man alle nicht rot markierten Zustände aus dem endlichen Automaten/Transducer.

Die Zustände des so erhaltenen endlichen Automaten sind alle vom Startzustand aus erreichbar (Schritt 1) und besitzen einen Pfad in einen Endzustand (Schritt 2). Folglich arbeitet der Algorithmus korrekt und benötigt lineare Zeit $O(|Q| + |E|)$.

Eine Implementierung dieses Algorithmus in Java ist im Paket model.transducer.algorithm als Subklasse der abstrakten Klasse TransducerAlgorithm zu finden.

A.2. Algorithmen für gerichtete Graphen

Die in den folgenden Unterabschnitten erwähnten Algorithmen basieren allesamt auf der Tiefensuche in Multigraphen. Eine nähere Erläuterung der Tiefensuche befindet sich in [Tar74]). Um redundanten Code in den

```
1  protected void dfsVisit(Vertex v) {
2    startVisit (v);
3    mark(v);
4    Iterator iter = graph.getOutEdges(v).iterator();
5    while (iter.hasNext()) {
6      DirectedEdge e = (DirectedEdge) iter.next();
7      if (!isMarked(e)) {
8        mark(e);
9        Vertex w = graph.getDestination(e);
10       if (!isMarked(w)) {
11         traverseDiscovery(e);
12         if (!isDone()) {
13           dfsVisit (w);
14         }
15         afterTraverseDiscovery(e);
16       }else{
17         traverseBack(e);
18       }
19     }
20   }
21   finishVisit (v);
22 }
```

Listing A.1: Klasse DFS, Methode **void** dfsVisit(Vertex)

drei Algorithmen zu vermeiden, habe ich die Tiefensuche als Method Template Pattern nach [GT98] in der Klasse model.graph.algorithm.DFS umgesetzt. Der Kern dieser Klasse ist in Listing A.1 dargestellt.

Dabei sind die Methoden **void** mark(Vertex), **void** mark(State), **boolean** isMarked(Vertex) und **boolean** isMarked(State) in der Klasse DFS selbst implementiert. Alle weiteren Methodenaufrufe beziehen sich auf abstrakt definierte Methoden, die die erbenden Klassen implementieren müssen.

A.2.1. Test auf Azyklität

Ein gerichteter Multigraph $G = (V, E)$ ist azyklisch genau dann, wenn kein Pfad existiert, im selben Knoten beginnt und endet. Stößt man während einer Tiefensuche auf eine Kante, dessen Zielknoten $v \in V$ bereits markiert wurde, so ist G zyklisch: der Pfad, den die Tiefensuche seit dem Markieren

von v beschritten hat ist beginnt und endet in v.

Folglich durchsuche der Algorithmus G in Tiefensuche und markiere alle besuchten Knoten. Stößt er dabei auf einen bereits markierten Knoten, bevor er alle Knoten besucht hat, so gebe **false** zurück. Andernfalls **true**.

Eine Implementation dieses Algorithmus, der allerdings die Zyklität, also das genaue Gegenteil, eines Multigraphen prüft, ist in der Klasse model.graph.algorithm.Cyclic zu finden.

A.2.2. Berechnen starker Zusammenhangskomponenten

Die Berechnung der starken Zusammenhangskomponenten eines gerichteten Multigraphen $G = (V, E)$ ist anhand eines Algorithmus von [Tar72] in linearer zeit $O(|E| + |V|)$ möglich.

Der Algorithmus basiert auf einem in dem Artikel [Tar72] bewiesenen Korollar, das besagt, dass die starken Zusammenhangskomponenten von G einen Teilbaum des Suchbaums bilden, den eine Tiefensuche auf G aufspannt. Daher reduziert sich das Problem auf die Bestimmung der Wurzeln dieser Teilbäume.

Nummeriert man die während einer Tiefensuche erreichten Knoten aufsteigend von 0 auf durch, so kann die Wurzel eines solchen Teilbaums mittels der Funktion LOWLINK : $V \to \mathbb{N}$ bestimmt werden. Diese Funktion ordnet jedem Knoten von G den Knoten mit der kleinsten Nummer zu, der in der gleichen starken Zusammenhangskomponente liegt und über beliebig viele Übergänge des Suchbaums gefolgt von einer Kante, welche im Suchbaum aufwärts oder in einen anderen Teilbaum führt:

$$\text{LOWLINK}(v) = \min\{\{v\} \cup \{w | v \xrightarrow{\star} \dashrightarrow w \wedge \exists u(u \xrightarrow{\star} v \wedge u \xrightarrow{\star} w \wedge u \text{ und } w \text{ liegen in der gleichen starken Zusammenhangskomponente})\}\},$$

wobei $u, v, w \in V$, sowie $\xrightarrow{\star}$ einen Pfad innerhalb des durch die Tiefensuche auf G aufgespannten Suchbaums und $\dashrightarrow$ eine Kante, die im besagten Suchbaumes zu einem Vorgänger oder in einen anderen Teilbaum führt, bezeichne. Ein Knoten ist die Wurzel einer starken Zusammenhangskomponente genau dann, wenn die einem Knoten bei der Entdeckung durch die Tiefensuche zugeordnete Nummer gleich LOWLINK(v) ist.

Die Werte der Funktion LOWLINK können während einer rekursiv Tiefensuche berechnet werden. Die dabei erreichten Knoten, die noch keiner starken Zusammenhangskomponente zugeordnet sind, werden auf einen Stack gelegt.

```
1  // initialize variables
2  i ← 0;
3  sccs.clear();
4  S ← ∅
5  for each (v ∈ V) LOWLINK[v] ← NUMBER[v] ← −1;
6
7  void strongconnect(v){
8    LOWLINK[v] ← NUMBER[v] ← i ← i+1;
9    S.push(v);
10   for each (u ∈ Trans[v]){
11     if (NUMBER[u] = −1){
12       // new Vertex u discovered
13       strongconnect(u);
14       LOWLINK[v] ← min{LOWLINK[v], LOWLINK[u]};
15     }else if (NUMBER[v] < NUMBER[u]){
16       // u already visited, correct LOWLINK
17       LOWLINK[v] ← min{LOWLINK[v], LOWLINK[u]};
18     }
19   }
20   if (LOWLINK[v] = NUMBER[v]){
21     // v is the root of strongly connected component
22     while(NUMBER[S.peek()] ≥ NUMBER[v]{
23       sccs.add(S.peek(), LOWLINK[S.peek()];
24     }
25   }
26 }
27
28 // call the recursive function strongconnect
29 for each (v ∈ V){
30   if (NUMBER[v] = −1) strongconnect(v);
31 }
```

Listing A.2: Der Algorithmus zur Bestimmung starker Zusammenhangskomponenten

Der Algorithmus ist in Listing A.2 in Pseudocode-Notation angegeben und in der Klasse model.graph.algorithm.StrongConnect als Subklasse von DFS implementiert. Die Variable S bildet hierbei den oben erwähnten Stack, die Nummerierung der Knoten in der Reihenfolge der Entdeckung durch die Tiefensuche wird in NUMBER gespeichert. Die Zuordnung der Knoten zu den starken Zusammenhangskomponenten geschieht über eine Tabelle sccs, welche den Knoten einer starken Zusammenhangskomponente gleiche Zahlen zuordnet (den LOWLINK Wert der Wurzel).

A.2.3. Berechnen einer topologischen Sortierung

Eine topologische Sortierung auf einem azyklischen gerichteten Multigraphen ist eine Anordnung der Knoten, so dass alle Nachfolger eines Knotens v hinter v stehen. (Nachfolger seien all die Knoten, die von v aus erreichbar sind.) Eine topologische Sortierung existiert für jeden solchen Graphen, muß jedoch nicht eindeutig sein.

Die Bestimmung einer solchen topologischen Sortierung für einen gerichteten azyklischen Multigraphen $G = (V, E)$ ist anhand einer Tiefensuche wie folgt möglich:

1. Sei $i \leftarrow 0$
2. Durchsuche G in Tiefensuche.
3. Sind dabei alle Nachfolger eines Knotens $v \in V$ besucht, so markiere v mit i und setze anschließend $i \leftarrow i + 1$.
4. Wenn alle Knoten besucht sind, ergibt sich aus der absteigenden Reihenfolge der i eine topologische Sortierung der Knoten von G.

Dieser Algorithmus bestimmt eine topologische Sortierung: Sei B der von der Tiefensuche während des Suchens aufgespannte Suchbaum. Sei $u \in V$ der derzeit untersuche Knoten in G und $(u, v) \in E$ eine davon ausgehende Kante. Dann kann (u, v) entweder zu einem (direkten oder indirekten) Nachfolger von u führen, d. h. ein Knoten im Teilbaum von B, dessen Wurzel u ist. In diesem Fall ordnet der Algorithmus u, sofern noch nicht geschehen, eine höhere Zahl i zu und v steht korrekt hinter u in der topologischen Sortierung.

Oder v ist kein Nachfolger von u, also v in einem anderen Teilbaum von B. In diesem Fall muß in einer topologischen Sortierung v ebenfalls

hinter u stehen, da v bereits von der Tiefensuche besucht und in den B eingeordnet wurde. Da v eine niedrigere Zahl i zugeordnet bekommen haben muß ist dies auch der Fall. Folglich arbeitet der Algorithmus korrekt und in linearer Zeit $O(|V| + |E|)$.

Eine Implementierung dieses Algorithmus befindet sich in der Klasse model.graph.algorithm.TopologicalSort.

A.3. Methodenlaufzeiten der implementierenden Klassen

Für die in den Interfaces deklarierten Methoden ergeben sich unter den Annahmen des Abschnitts 6.1 und unter Verwendung der implementierenden Klassen des Pakets model.transducer.implementation die in den folgenden Tabellen A.1–A.6 aufgelisteten Laufzeiten. Die hierbei verwendeten Bezeichnungen sind

- $S = \sum_{t \in E} |l_{\text{out}}(t)|$ für die Summe der Längen aller Ausgaben
- Q und E für die Mengen der Zustände und Übergänge
- Σ und Δ für das Ein- bzw. Ausgabealphabet

Methode	Laufzeit
Vertex getDestinationVertex()	$O(1)$
Vertex getSourceVertex()	$O(1)$
Transition toTransition()	$O(1)$

Tabelle A.1.: Interface DirectedEdge

Methode	Laufzeit
Collection getVertices()	$O(1)$
Collection getEdges()	$O(1)$
int getNumberOfVertices()	$O(1)$
int getNumberOfEdges()	$O(1)$
Transducer toTransducer()	$O(1)$

Tabelle A.2.: Interface DirectedGraph

Methode	Laufzeit
void addInTransition(Transition t)	$O(1)$
void addOutTransition(Transition t)	$O(1)$
int getId()	$O(1)$
Collection getInTransitions()	$O(1)$
Map getOutTransitions()	$O(1)$
Transducer getTransducer()	$O(1)$
boolean isFinal()	$O(1)$
void removeInTransition(Transition t)	$O(1)$
void removeOutTransition(Transition t)	$O(1)$
void setFinal(boolean isFinal)	$O(1)$
int getOutDegree()	$O(1)$
int getInDegree()	$O(1)$
Vertex toVertex()	$O(1)$

Tabelle A.3.: Interface State

Methode	Laufzeit
State addState(boolean isFinal)	$O(1)$
void appendToInitialOutput(String suffix)	$O(\|\text{suffix}\|)$
Transition addTransition(...)	$O(\|\text{outputLabel}\|)$
Collection getStates()	$O(1)$
Collection getTransitions()	$O(1)$
Alphabet getInputAlphabet()	$O(1)$
Alphabet getOutputAlphabet()	$O(1)$
void addToInputAlphabet(char c)	$O(1)$
void addToOutputAlphabet(char c)	$O(1)$
void removeFromInputAlphabet(char c)	$O(\|E\|)$
void removeFromOutputAlphabet(char c)	$O(S)$
void setInitialState(State s)	$O(1)$
String getInitialOutput()	$O(1)$
State getInitialState()	$O(1)$
int getMaxStateId()	$O(1)$
void setInitialOutput(String initialOutput)	$O(1)$
void removeTransition(Transition t)	$O(1)$
void removeState(State s)	$O(\|Trans(\text{s}) \cup Trans^T(\text{s})\|)$
int getNumberOfTransitions()	$O(1)$
int getNumberOfStates()	$O(1)$
DirectedGraph toGraph()	$O(1)$
Object clone()	$O(\|\Sigma\| + \|\Delta\| + \|E\| + \|Q\| + S)$

Tabelle A.4.: Interface Transducer

Methode	Laufzeit
void appendToOutput(String suffix)	$O(\|\text{suffix}\|)$
void deleteOutputLabelSuffix(int length)	$O(\|\text{length}\|)$
State getDestination()	$O(1)$
Character getInputLabel()	$O(1)$
String getOutputLabel()	$O(1)$
State getSource()	$O(1)$
void setDestination(State s)	$O(1)$
void setInputLabel(Character inputLabel)	$O(1)$
void setOutputLabel(String outputLabel)	$O(\|\text{outputLabel}\|)$
void setSource(State s)	$O(1)$
Transducer getTransducer()	$O(1)$
String getAutomatonLabel()	$O(1)$
DirectedEdge toDirectedEdge()	$O(1)$

Tabelle A.5.: Interface Transition

Methode	Laufzeit
Collection getInEdges()	$O(1)$
Collection getOutEdges()	$O(1)$
State toState()	$O(1)$

Tabelle A.6.: Interface Vertex

A.4. Tabellen der Laufzeitmessungen

	Laufzeit in ms		
$\|E\|$	$\|Q\| = 1000$	$\|Q\| = 2000$	$\|Q\| = 3000$
1000	91,6	41,6	76,6
2000	449,6	62,6	91,2
3000	677,4	1050,8	112,6
4000	1342,0	2958,0	934,6
5000	1506,6	2924,0	6218,4
6000	1646,6	5861,4	10001,0
7000	1948,6	4898,4	11682,4
8000	2170,6	5372,6	13275,0
9000	2360,4	5581,6	14173,4
10000	2585,8	5819,2	13167,6
11000	2807,8	6055,2	15031,8
12000	3014,2	6460,2	14278,6
13000	3223,6	6897,4	13147,0
14000	3426,4	7329,6	12357,0
15000	3620,4	7747,0	13248,4
16000	3798,6	8028,6	14784,8
17000	4118,0	8437,6	13830,0
18000	4212,8	8764,4	15078,2
19000	4403,8	9367,4	14336,0
20000	4858,2	9663,6	15152,6

Tabelle A.7.: Quasi-Determinierung: $|\Sigma| = |\Delta| = 26$, $l_{\max} = 3$

	Laufzeit in ms		
$\|l_{\max}\|$	$\|E\| = 3000$	$\|E\| = 4000$	$\|E\| = 5000$
1	1125,8	1226,8	1490,2
2	1953,0	2279,0	2939,2
3	2765,2	3554,8	4346,2
4	3467,8	4810,2	5790,4
5	4493,6	6035,0	7241,4
6	5619,6	7066,0	8711,4
7	6678,8	8316,8	10182,6
8	7718,2	9610,0	11641,4
9	8831,8	10670,4	13117,6
10	9637,4	11927,4	14564,8

Tabelle A.8.: Quasi-Determinierung: $|\Sigma| = |\Delta| = 26$, $|Q| = 1000$

	Laufzeit in ms		
$\|E\|$	$\|Q\| = 1000$	$\|Q\| = 2000$	$\|Q\| = 3000$
1000	26,6	38,8	73,4
2000	1048,0	59,6	88,0
3000	3874,0	806,8	106,0
4000	6722,2	4223,6	1075,2
5000	9321,2	8741,2	4346,0
6000	11915,6	13330,4	8299,6
7000	15308,0	17568,4	15793,6
8000	18142,2	21585,0	20306,4
9000	22587,6	26297,4	27148,2
10000	25109,4	31720,8	32539,8
11000	28450,2	36837,6	38511,4
12000	32125,0	41549,2	44657,0
13000	35357,2	46060,8	51028,4
14000	39335,6	51913,6	56711,8
15000	44225,2	57329,4	63511,6
16000	45732,8	63234,6	70332,8
17000	49895,0	67338,6	75443,8
18000	52476,6	76442,4	85535,8
19000	54860,6	76160,2	88547,6
20000	58395,0	84153,2	96834,8

Tabelle A.9.: Automaten-Minimierung: $|\Sigma| = |\Delta| = 26$, $l_{\max} = 3$

$\|Q\|$	Laufzeit in ms
500	34168,0
1000	60010,6
1500	78874,8
2000	85716,8
2500	93085,6
3000	98876,0
3500	104721,6
4000	108997,6
4500	111127,4
5000	113383,2

Tabelle A.10.: Automaten-Minimierung: $|\Sigma| = |\Delta| = 26$, $l_{\max} = 3$, $|E| = 20000$

$\|Q\|$	Laufzeit in ms
500	39542.4
1000	54134.2
1500	64557.8
2000	73979.8
2500	77236.4
3000	87902.8

Tabelle A.11.: Äquivalenztest: $|\Sigma| = |\Delta| = 26$, $l_{\max} = 3$, $|E| = 10000$

Literaturverzeichnis

[AHU74] Aho, A. V., J. E. Hopcroft und J. D. Ullman: *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, Reading, MA, 1974.

[Brz62] Brzozowski, J. A.: *Canonical regular expressions and minimal state graphs for definite events*. In: *Mathematical Theory of Automata*, MRI Symposien Serie, Seiten 529–561, Polytechnic Institute of Brooklyn, N.Y., 1962. Polytechnic Press.

[GT98] Goodrich, Michael T. und Roberto Tamassia: *Data Structures and Algorithms*. John Wiley & Sons Inc., New York, NY, 2. Auflage Auflage, 1998.

[HU00] Hopcroft, J. E. und J. D. Ullman: *Einführung in Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Oldenbourg Wissenschaftsverlag GmbH, München, 4. Auflage Auflage, 2000.

[Huf54] Huffmann, D. A.: *The Synthesis of Sequential Switching Circuits*. Journal of Franklin Institute, 257(3):161–190, 1954.

[Moh00] Mohri, Mehryar: *Minimization algorithms for sequential transducers*. Theoretical Computer Science, 234:177–201, 2000.

[Moo56] Moore, E. F.: *Gedanken-experiments on sequential machines*. In: C. E. Shannon und J. McCarthy (Herausgeber): *Ann. Math. Studies*, Seiten 129–153, Princeton University Press, 1956.

[Rev92] Revuz, D.: *Minimisation of acyclic deterministic automata in linear time*. Theoretical Computer Science, 92(1):181–189, 1992.

[Spe03] Specht, J.: *Compiler-Konstruktion I*. Fachgebiet Programmiersprachen und Übersetzer, Universität Hannover, Institut für Informationssysteme, WS 2002/2003.

[Tar72] Tarjan, Robert E.: *Depth first search and linear graph algortihms*. SIAM Journal on Computing, 1 (2):146–160, 1972.

[Tar74] Tarjan, Robert E.: *Finding dominators in directed graphs*. SIAM Journal on Computing, 3:62–89, 1974.

[VW02] Vossen, Gottfried und Kurt-Ulrich Witt: *Grundlagen der Theoretischen Informatik mit Anwendungen*. Friedrich Vieweg & Sohn Verlagsgesellschaft mbH, Braunschweig, 2. Auflage Auflage, 2002.