Gewichtete Automatenfibel

Christian Wurm
cwurm@phil.hhu.de

March 13, 2014

1 Einleitung

Eine sehr kompakte Einführung in die Theorie der gwichteten Automaten. Das Ziel ist: 1. Motivationen zu geben für die formalen Konzepte, 2. die Definitionen genau und übersichtlich zu geben, und 3. auch die wichtigsten Ergebnisse darzustellen. Ich werde hier auf Beweise weitestgehend verzichten, da ich ohnehin nur Ergebnisse präsentiere, die auch anderswo nachgelesen werden können. Zu gewichteten Automaten im Allgemeinen (leider etwas *sehr* ausfhrlich) gibt es folgendes Handbuch: [1]. Da steht eigentlich alles drin; nur für Baumautomaten habe ich mich an Maletti, [2] gehalten.

2 Halbringe

Jede Theorie der gewichteten Automaten fängt bei Halbringen an. Warum? Weil wir unsere Gewichte immer einem Halbring entnehmen. Warum gerade einem Halbring? Dazu müssen wir erstmal schauen, was das ist.

Definition 1 *Ein Halbring ist eine algebraische Struktur* $(M, \oplus, \otimes, \overline{0}, \overline{1})$*, wobei* M *eine Menge ist mit* $\overline{0}, \overline{1} \in M$*,* $\oplus, \otimes : M \times M \to M$ *zwei binäre Operatoren sind, die folgende Gesetze erfüllen:*

1. *f.a.* $m, n, o \in M$*,* $m \oplus (n \oplus o) = (m \oplus n) \oplus o$ *(*$\oplus$*-assoziativ)*
2. *f.a.* $m, n, o \in M$*,* $m \oplus (n \otimes o) = (m \otimes n) \oplus o$ *(*$\otimes$*-assoziativ)*
3. *f.a.* $m \in M$*,* $\overline{0} \oplus m = m \oplus \overline{0} = m$ *(*$\overline{0}$ *ist* $\oplus$*-neutral)*
4. *f.a.* $m \in M$*,* $\overline{1} \otimes m = m \otimes \overline{1} = m$ *(*$\overline{1}$ *ist* $\otimes$*-neutral)*
5. *f.a.* $m \in M$*,* $\overline{0} \otimes m = m \otimes \overline{0} = \overline{0}$ *(*$\overline{0}$ *absorbiert* $\otimes$*)*
6. *f.a.* $m, n, o \in M$*,* $(m \oplus n) \otimes o = (m \otimes n) \oplus (n \otimes o)$ *(linkes Distributivgesetz)*
7. *f.a.* $m, n, o \in M$*,* $m \otimes (n \oplus o) = (m \otimes n) \oplus (m \otimes o)$ *(rechtes Distributivgesetz)*

Warum diese Definition? Zunächst folgendes: diese Definition ist eher abstrakt und allgemein; es ist also vielleicht besser zu denken: eine gewisse, konkrete Struktur ist ein Halbring, weil sie alle obigen Eigenschaften erfüllt, anstatt sich vorzustellen was ein "Halbring an sich" ist. Das ist natürlich eine interessante Frage, aber eher für Algebraiker, da wir uns immer nur mit sehr konkreten Halbringen beschäftigen werden. Wir gehen also direkt zu unserem ersten Beispiel.

Beispiel 1 Die Struktur $(\mathbb{N}_0, +, \cdot, 0, 1)$ ist ein Halbring, wobei $\mathbb{N}_0$ die natürlichen Zahlen mit 0 sind, und $+, \cdot$ Addition und Multiplikation wie in der Schule. Nennen wir diesen Halbring *natürlich*, da er über den natürlichen Zahlen operiert; weiterhin werden wir die arithmetische Addition und Multiplikation ebenfalls natürlich nennen, auch wenn wir jenseits der natürlichen Zahlen operieren. Es ist leicht zu überprüfen dass der natürliche Halbring ein Halbring ist. Was wichtiger ist, ist: es ist ein spezieller Halbring, der viele Eigenschaften hat die nicht jeder Halbring hat. Eine besondere wäre z.B.: $m+n = n+m$, $n \cdot m = m \cdot n$; es gilt also auch das Kommutativgesetz. Eine weitere interessante Eigenschaft ist folgende: wir haben $n \cdot m = n + .. + n$ (m mal) $= m + ... + m$ (n mal); diese Eigenschaft ist also: Multiplikation ist über iterierte Addition definiert.

Beispiel 2 ist nur eine leichte Verallgemeinerung von Beispiel 1: $(\mathbb{R}, +, \cdot, 0, 1)$ erfüllt ebenso alle Bedingungen des Halbringes. Dieser *reelle* Halbring hat eine besondere Eigenschaft, die der natürliche nicht hat: wir haben sog. *inverse Elemente*. D.h., wir haben für jedes $x \in \mathbb{R}$ ein Element $\overline{x} \in \mathbb{R}$, so dass $x + \overline{x} = \overline{x} + x = 0$; ebenso haben wir inverse Elemente für die Multiplikation, d.h. $x \in \mathbb{R}$ ein Element $x^{-1} \in \mathbb{R}$, so dass $x \cdot x^{-1} = x^{-1} \cdot x = 1$.

Beispiel 3 führt uns vor Augen, wie allgemein der Begriff des Halbrings ist. Sei Σ ein Alphabet; mit Σ^* bezeichnen wir die Menge aller endlichen Wörter über Σ; für eine Menge M bezeichnen wir mit $\wp(M)$ deren Potenzmenge, also die Menge aller ihrer Teilmengen. Mit $\cdot$ bezeichnen wir in diesem Kontext die einfache Konkatenation von Worten; wir "heben" diese Operation nun allerdings auf Mengen an: für $M, N \subseteq \Sigma^*$ definieren wir $M \cdot N := \{wv : w \in M, v \in N\}$, so dass also auch $M \cdot N \subseteq \Sigma^*$. Wir kommen nun zu unserem Halbring $(\wp(\Sigma^*), \cup, \cdot, \emptyset, \{\epsilon\})$. Zur Erklärung: unsere "Addition" ist nun Vereinigung, unsere "Multiplikation" ist Konkatenation. Das neutrale Element der Vereinigung ist $\emptyset$, und das neutrale Element der Konkatenation ist $\{\epsilon\}$ (nicht ϵ, denn wir haben ja nur Mengen und Konkatenation von Mengen von Wörtern). Das Distributivgesetz lässt sich relativ leicht verifizieren, ebenso Assoziativität. $\cup$ ist zusätzlich kommutativ, $\cdot$ allerdings nicht! Dieser Halbring hat eine weitere Eigenschaft: er ist **idempotent**, d.h. für jedes $M \in \wp(\Sigma^*)$ gilt: $M \cup M = M$. Eine wichtige Bemerkung zu diesem Halbring scheint angebracht: man darf niemals $\emptyset$ und $\{\epsilon\}$ verwechseln; $\{\epsilon\}$ ist nicht leer und nicht neutral für $\cup$. Umgekehrt ist $\emptyset$ absorbierend für $\cdot$: $M \cdot \emptyset = \emptyset \cdot M = \emptyset$. Dieser Halbring heißt übrigens aus Gründen, die wir wahrscheinlich nicht besprechen werden, der **freie idempotente Halbring**.

Wir können nun kurz darüber sprechen, warum wir gerade Halbringe betrachten. Wenn wir von Automaten ausgehen, dann haben wir zwei Phänomene, die wir interpretieren möchten: wir haben als Eingabe Worte, die auf gewis-

sen Pfaden durch unseren Automaten gehen. Dabei kann es – falls unser Automat nicht deterministisch ist, durchaus mehrere Pfade geben. Das sind die beiden grundlegenden Operationen der Automatentheorie: die *Abfolge* von Operationen (längs im Pfad) und die nichtdetermistische Auswahl von Operationen. Wenn wir Worten Gewichte zuweisen möchten, dann möchten wir beide automatentheoretische Operationen interpretieren, daher die zwei abstrakten Operationen $\otimes$ (entspricht der Abfolge im Pfad) und $\oplus$ (entspricht der nichtdeterministischen Auswahl). Die übrigen Gesetze sind in diesem Rahmen zu verstehen: die $\overline{0}$ repräsentiert die Unmöglichkeit eines Übergangs; daher soll sie den ganzen Pfad (interpretiert als $\otimes$) absorbieren. Die Assoziativitätsgesetze sind notwendig, da auch die lineare Abfolge von Operationen und die (nichtdetermistische) Auswahl assoziativ sind; ebenso ist das Distributivgesetz zu verstehen: ob ich "(a oder b) und dann c" mache ist dasselbe wie wenn ich "(a und dann c) oder (b und dann c) mache. Die Gesetze des Halbrings erklären sich also aus den automatentheoretischen Begebenheiten; gleichzeitig sind Halbringe die allgemeinsten Strukturen, die diese Anforderungen erfüllen. Der Grund der besonderen Auswahl von Halbringen ist also: wir möchten so allgemein wie möglich, aber so spezifisch wie nötig sein.

Wir sehen hier in Andeutungen auch schon den Grund, warum die Theorie der gewichteten Automaten so nützlich, allgemein und schön ist: wir trennen den Automaten selber von seiner Interpretation im Halbring, so dass wir beide Teile vollkommen separat behandeln können: wir können denselben Automaten in völlig verschiedenen Halbringen interpretieren, ohne ihn selbst verändern zu müssen. Wir kommen nun zu einigen weiteren wichtigen Halbringen.

Beispiel 4 ist der Boolesche Halbring $(\{0,1\}, max, min, 0, 1)$. Hier ist max das Maximum, min das Minimum. Er ist nach Boole benannt, da wir in (klassische) Logik die Formeln in diesem Halbring interpretieren; max ist die Interpretation von $\vee$ ("oder"), min die Interpretation von $\wedge$ ("und"). Allgemein gilt später: wenn wir einen gewichteten Automaten als einen ganz klassischen, ungewichteten Automaten auffassen wollen, dann müssen wir ihn einfach im Booleschen Halbring interpretieren.

Beispiel 5 Der Halbring der "Wahrscheinlichkeiten". Dieser Halbring ist mit Vorsicht zu genießen, da Wahrscheinlichkeiten ihre eigenen Axiome haben, die wir so im Halbring nicht darstellen können. Wir können also nur das algebraische Gerüst der Wahrscheinlichkeiten darstellen: $([0,1], \overline{+}, \cdot, 0, 1)$. $[0,1]$ ist das geschlossene Intervall auf den reellen Zahlen; $\cdot$ ist die natürliche Multiplikation; und für $x, y \in [0,1]$ definieren wir $x\overline{+}y = 1$, falls $x + y > 1$, und $x\overline{+}y = x + y$ andernfalls. Der Grund für diese Definition ist dass $[0,1]$ nicht abgeschlossen ist unter natürlicher Addition (aber sehr wohl unter natürlicher Multiplikation!)

Beispiel 6 ist ein Halbring, der sehr wichtig für uns ist, der sog. **tropische Halbring**: $(\mathbb{R}_+^\infty, min, +, \infty, 0)$. Hier $\mathbb{R}_+^\infty$ ist die Menge der nicht-negativen reellen Zahlen zusammen mit ∞, welches größer ist als jede reelle Zahl. Daher ist ∞ neutral für die Operation min (Minimum); Addition mit 0 ist klar. Beachten Sie, dass hier die Addition unsere "Multiplikation" ist! Es ist leicht zu verifizieren dass der tropische Halbring unsere übrigen Gesetze erfüllt; wir

demonstrieren das für das linke Distributivgesetz. Wir haben $x + min(y, z) = min(x + y, z + y)$, da $x + y \leq x + z$ genau dann wenn $y \leq z$. Auch der tropische Halbring ist idempotent: $min(x, x) = x$. Er hat noch eine weitere schöne Eigenschaft: wir haben immer $min(x, y) \in \{x, y\}$; d.h. die "Addition" liefert uns als Wert den Wert eines ihrer Argumente.

Als **Beispiel 7** betrachten wir einige Varianten des tropischen Halbrings. Zunächst den Halbring $(\mathbb{N}_+^\infty, min, +, \infty, 0)$. Dieser Halbring ist dem tropischen sehr ähnlich und wird meist genauso benannt. Da er allerdings weniger flexibel ist, bevorzugen wir den "klassischen" tropischen Halbring. Der praktische Nachteil ist folgender: wir schätzen die Gewichte im tropischen Halbring als $-log(\hat{p})$, wobei $\hat{p}$ eine geschätzte Wahrscheinlichkeit ist. Ich muss nicht erklären dass es sehr unwahrscheinlich ist dass $-log(\hat{p}) \in \mathbb{N}$. Eine weitere Variante ist der sog. **arktische Halbring** $(\mathbb{N}_-^\infty, max, +, -\infty, 0)$. Er ist soz. das duale Gegenstück vom tropischen und *isomorph*, d.h. strukturgleich. Aus genau diesem Grunde ist er aber auch eigentlich uninteressant, da er in allen relevanten Eigenschaften mit dem tropischen Halbring zusammenfällt.

Das sollte zunächst reichen; wir wenden uns jetzt Sprachen, Relationen und Automaten zu.

3 Sprachen und Relation

Für eine allgemeine Einführung in Sprachen, Relationen und Automaten verweise ich auf mein Skript http://user.phil.uni-duesseldorf.de/∼cwurm/FSM/fibeldruck.pdf. Wir gehen daher direkt zu gewichteten Sprachen und Automaten. Mit **Gewichten** meinen wir normalerweise reelle Zahlen; wir werden also, solange nichts explizit anderes gesagt wird, davon ausgehen, dass unsere Gewichte eine Teilmenge von $\mathbb{R}$ sind (ich sage aber nicht dass es immer immer so sein muss). Damit sind wir schon fertig für unsere allgemeinen Definitionen.

Definition 2 *Sei Σ ein Alphabet. Eine **gewichtete Sprache** ist eine Funktion $G : \Sigma^* \to \mathbb{R}$. Für zwei Mengen M, N ist eine gewichtete **Relation** über M, N eine Funktion $G : M \times N \to \mathbb{R}$.*

Wenn wir also von Sprachen zu gewichteten Sprachen gehen, dann gehen wir von einer Menge zu einer Funktion von dieser Menge nach $\mathbb{R}$. Dasselbe gilt für Relationen; da uns an dieser Stelle eigentlich nur mengentheoretische Eigenschaften interessieren und Sprachen ebenso wie Relationen Mengen sind, meine ich wenn ich Sprache sage immer auch Relation. Allgemein kann man jede Menge mit einer solchen Funktion charakterisieren, der sog. **charakteristischen Funktion** einer Menge. Sei M eine Menge; dann definieren wir χ_M als Funktion wie folgt:

$$\chi_M(x) = \begin{cases} 1, \text{ falls } x \in M \\ 0 \text{ andernfalls} \end{cases} \tag{1}$$

(Das kommt mit einigen Problemen in der axiomatischen Mengenlehre, die sind hier aber uninteressant.) Es ist klar dass jede Menge ihre charakteristische Funktion eindeutig bestimmt und umgekehrt. Wir können also jede Sprache als gewichtete Sprache auffassen, indem wir die Menge als ihre charakteristische Funktion auffassen. Insofern sind gewichtete Sprachen eine Generalisierung von Sprachen, da wir mehr als nur die Werte 0,1 zur Verfügung haben.

Für Relationen ist folgendes zu beachten: die Relationen, die wir betrachten sind Teilmengen von $\Sigma^* \times \Sigma^*$; das bedeutet, die gewichteten Relationen die wir betrachten haben die Form $G : \Sigma^* \times \Sigma^* \to \mathbb{R}$.

4 Gewichtete Automaten

Gewichtete Sprachen sind Funktionen; ein gewichteter Automat soll eine solche Funktion induzieren.

Ein gewichteter Automat ist ein Tupel $\mathfrak{A} = (Q, \Sigma, \pi, \delta, \tau)$, wobei Q eine endliche Menge von Zuständen ist und Σ ein endliches Eingabealphabet ist. δ entspricht der klassischen Übergangsfunktion; es ist aber nun eine Funktion in eine Menge M, die die **Trägermenge** eines Halbringes ist. Wir nehmen also an M ist die Trägermenge eines Halbringes $(M, \oplus, \otimes, \overline{0}, \overline{1})$. Dann ist $\delta : Q \times \Sigma \times Q \to M$ eine Funktion, die jedem Übergang ein Gewicht zuweist; falls wir also $\delta(q, a, q') = x$ haben, bedeutet das soviel wie: falls $\mathfrak{A}$ in Zustand q ist und liest eine Eingabe a liest, geht er mit Gewicht x in den Zustand q'. Weiterhin haben wir eine Funktion $\pi : Q \to M$, welches den Startzustand ersetzt: π liefert uns ein Gewicht dafür, dass wir in einem Gewissen Zustand anfangen; $\tau : Q \to M$ macht dasselbe mit den Endzuständen.

Das Kernstück der Theorie der gewichteten Automaten ist die Definition, wie ein Automat eine Sprache induziert. Diese Definition ist gar nicht einfach und braucht einige Begriffe. Ein **Pfad** durch einen Automaten ist eine beliebige (endliche) Folge $(q_0, a_0, q_1 ..., a_{i-1}, q_i) : q_0, ..., q_{i-1} \in Q$, die illustriert, wie wir uns durch einen Automaten bewegen, indem wir ein gewisses Wort lesen. NB: die Indizes an Zuständen und Buchstaben stellen nur die Reihenfolge dar und sagen nichts über die Identität verschiedener Zustände! Ein Wort $\vec{w} = a_0 ... a_{i-1}$ der Länge i induziert eine Menge von Pfaden $(q_0, a_0, q_1, a_1, q_2, ..., a_{i-1}, q_i)$ in einem Automaten $\mathfrak{A}$, die wir mit $\pi_w(\mathfrak{A})$ denotieren. Wir können nun jedem Pfad $\overline{p}$ ein Gewicht zuweisen wie folgt:

$$
\begin{aligned}
& G_{\mathfrak{A}}(q_0, a_0, q_1, .., q_{i-1}, a_{i-1}, q_i) \\
&= \pi(q_0) \otimes \delta(q_0, a_0, q_1) \otimes ... \otimes \delta(q_{i-1}, a_{i-1}, q_i) \otimes \tau(q_i) \\
&= \pi(q_0)\tau(q_i) \bigotimes_{j=0}^{i-1} \delta(q_j, a_j, q_{j+1}
\end{aligned} \tag{2}
$$

Hier benutzen wir $\bigotimes$ um ein beliebig langes Produkt darzustellen; $\bigotimes$ verhält sich also zu $\otimes$ wie $\prod$ zu $\cdot$, $\sum$ zu $+$; dasselbe gilt für $\bigoplus$ und $\oplus$. In Worten: wir multiplizieren das Gewicht dass wir in q_0 anfangen, in q_i aufhören, und

das Gewicht jedes einzelnen Übergangs. Von hier ist es nur ein kleiner Schritt zum Gewicht eines Wortes. Wir denotieren die probabilistische Sprache, die $\mathfrak{A}$ induziert, mit $G(\mathfrak{A})$:

$$(3) \quad G(\mathfrak{A})(\vec{w}) = \bigoplus_{\overline{p} \in \pi_w(\mathfrak{A})} G_{\mathfrak{A}}(\overline{p})$$

Wir summieren also über alle Pfade, die ein Wort in einem Automaten induziert. Das war es schon; ich habe aber 2 wichtige Anmerkungen:

1. Der Automat selber spezifiziert noch nicht den Halbring, nur die Gewichte selbst. Wir können also denselben Automaten in ganz verschiedenen Halbringen interpretieren, sofern die Gewichte es zulassen!

2. Eine ganz wichtige Konvention ist folgende: wir nehmen *immer* an, dass unsere Automaten *total*, also alle Übergänge mit allen Buchstaben definiert sind. In der Praxis sind selten alle Übergange ausbuchstabiert; daher nehmen wir an, dass diejenigen, die nicht ausbuchstabiert sind, das Gewicht $\overline{0}$ haben: sie spielen damit für das Gewicht eines Wortes keine Rolle, weil die $\overline{0}$ die Multiplikation absorbiert, und für Addition neutral ist!

Der zweite Punkt relativiert den ersten: nehmen wir einen Automaten $\mathfrak{A}$ mit Gewichten in $\mathbb{R}_0^+$; den können wir im reellen und im tropischen Halbring interpretieren. Aber je nach Wahl müssen wir die undefinierten Übergänge als 0 oder ∞ auffassen; das heißt also insbesondere: falls wir einen Übergang mit 0 haben, bedeutet diese 0 genau das Gegenteil je nachdem ob reell oder tropisch interpretiert! Man muss sich also meist schon Gedanken über den Halbring machen, wenn man einen Automaten schreibt, da wir im reellen Halbring immer das maximale Gewicht suchen, im tropischen Halbring das minimale.

Diese Definition finden ebenso Anwendungen auf Relationen $R \subseteq \Sigma^* \times \Sigma^*$, bzw. gewichtete Relationen $G : \Sigma^* \times \Sigma^* \to \mathbb{R}$; der einzige Unterschied ist, dass die Übergänge mit Paaren (a, b) von Buchstaben beschriftet sind, und dass Konkatenation nach dem Muster $(a, b) \cdot (c, d) = (ac, bd)$ erfolgt (näheres im oben angeführten Skript).

5 Operationen auf Automaten/Transduktoren

Die wichtigsten Operationen auf gewichteten Automaten/Transduktoren sind die folgenden. Der Übersicht halbe folgende Konvention: wo es keinen Unterschied macht, ob Automat oder Transduktor, steht ein Automat; wo es nur Transduktoren gibt, ein Transduktor, und wo beides geht und es einen signifikanten Unterschied macht, steht beides.

Gewichtete Vereinigung

$$(4) \quad G(\mathfrak{A}_1 \cup \mathfrak{A}_2)(w) = G(\mathfrak{A})_1(w) \oplus G(\mathfrak{A}_2)(w)$$

Wir addieren also die Gewichte von w.

Gewichteter Schnitt

(5) $G(\mathfrak{A}_1 \cap \mathfrak{A}_2)(w) = G(\mathfrak{A}_1)(w) \otimes G(\mathfrak{A}_2)(w)$

Multiplikation statt Addition.

Komposition

Nur für Transduktoren!

(6) $G(\mathfrak{T}_1 \circ \mathfrak{T}_2)(w, v) = \bigoplus_u G(\mathfrak{T}_1)(w, u) \otimes G(\mathfrak{T}_2)(u, v)$

Also eine große Summe über kleine Produkte! Die Summe $\bigoplus$ ist implizit über alle $u \in \Sigma^*$ definiert; aber das Alphabet ist nicht explizit definiert, daher die etwas missverständliche Schreibweise.

Konkatenation

Erstmal für einfache Automaten, um es übersichtlicher zu machen:

(7) $G(\mathfrak{A}_1 \cdot \mathfrak{A}_2)(w) = \bigoplus_{uv=w} G(\mathfrak{A}_1)(u) \otimes G(\mathfrak{A}_2)(v)$

Wir dekomponieren also w zu uv und multiplizieren $\mathfrak{A}_1(u), \mathfrak{A}_2(v)$; und das machen wir für *alle Dekompositionen*; die Ergebnisse werden aufsummiert. Nun kommen wir zu Transduktoren; das Schema ist dasselbe, sieht nur komplizierter aus: für Paare (u_1, v_1), (u_2, v_2) bilden wir die einfache, ungewichtete Konkatenation mit $(u_1, v_1) \cdot (u_2, v_2) = (u_1 u_2, v_1 v_2)$; die gewichtete Variante ist:

(8) $G(\mathfrak{T}_1 \cdot \mathfrak{T}_2)(w, v) = \bigoplus_{y_1 y_2 = w, z_1 z_2 = v,} G(\mathfrak{T}_1)(y_1, z_1) \otimes G(\mathfrak{T}_2)(y_2, z_2)$

Also dasselbe, aber es ist leicht zu sehen dass die Anzahl der Dekomposition nun quadratisch wächst mit der Wortlänge!

Stern

Der Stern ist sowohl für Automaten als auch Transduktoren leicht definiert, wenn man Konkatenation erstmal hat:

(9) $G(\mathfrak{A})^*(w) = G(\mathfrak{A}^0)(w) \oplus G(\mathfrak{A})(w) \oplus G(\mathfrak{A} \cdot \mathfrak{A})(w) \oplus G(\mathfrak{A} \cdot \mathfrak{A} \cdot \mathfrak{A})(w) \oplus ...$

Hier ist $\mathfrak{A}^0$ ein besonderer Automat, der ϵ den Wert $\overline{1}$ zuweist, und allen anderen Ketten den Wert $\overline{0}$.

Projektion

Projektion ist eine einfache Operation auf normalen Transduktoren; sei $R \subseteq \Sigma^* \times \Sigma^*$ eine Relation; man definiert $\pi_1(R) = \{w : (w, v) \in R\}$, π_2 parallel. Gewichtet sieht das wie folgt aus:

$$(10) \quad G(\pi_1(\mathfrak{T})) = \bigoplus_w G(\mathfrak{T})(v, w)$$

Wir bilden also die Summe über alle möglichen zweiten Komponenten; π_2 ist parallel definiert.

6 Wir bauen einen Spell-Checker

Was ist ein spell-checker? Hier gibt es bereits verschiedene mögliche Antworten, die verschiedene technische Konsequenzen haben. Ich sehe drei Antworten, bzw. drei Aufgaben für S, den spell-checker:

1. S prüft, ob ein Wort bekannt ($\cong$ richtig) ist oder nicht, und gibt also eine binäre Ausgabe.
2. S liefert zu jedem unbekannten ($\cong$ falschen) Wort ein bekanntes Wort, das maximale Plausibilität hat.
3. S liefert zu *jedem* Wort eine Alternative, die maximale Plausibilität hat.

Folgende Anmerkungen:

- Für S bedeutet falsch immer dasselbe wie unbekannt. Das ist einfach eine Konvention, die notwendig ist, wenn man Umsetzbarkeit will.
- Es ist unklar was Plausibilität heißen soll, aber es ist klar das verschiedene Faktoren eine Rolle spielen können:
  1. Wie ähnlich ist das neue Wort dem gegebenen?
  2. Wie häufig/wahrscheinlich ist das neue Wort allgemein?
  3. Wie häufig/wahrscheinlich ist das neue Wort im gegebenen Kontext?

  All das kann Plausibilität beeinflussen; allgemein werden wir diese 3 Faktoren in aufsteigender Reihenfolge berücksichtigen.
- Die Frage ob wir zu jedem Wort eine Alternative präsentieren sollen, hängt von der Anwendung ab: ein klassischer spell-checker tut das nicht; Google aber beispielsweise schon (da es Trefferzahl maximieren will).

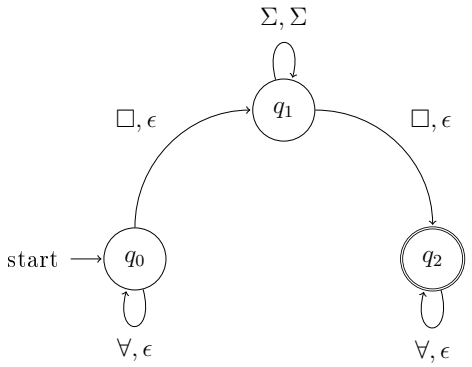
Wir werden für das erste in beiden Nummerierungen nur bis Punkt 2 gehen: wir möchten bekannte Wörter einfach akzeptieren, und die Plausibilität möchten wir unabhängig vom Kontext betrachten.

6.1 Ein binärer Spell-Checker

Wir konstruieren nun den einfachsten spell-checker: er nimmt Worte zur Eingabe und gibt richtig oder falsch aus, etwa wie in MSWord, wo falsche Worte einfach rot unterstrichen werden. Es ist klar, dass diese Konstruktion auf einem einfachen Automaten basiert, der nur unser Lexikon akzeptiert. Allerdings gibt es auch hier einige Probleme:

1. Wo kriegen wir unser Lexikon her?
2. Wir möchten ja nicht nur einzelne Worte eingeben, sondern Texte, und möchten dass darin die falschen Worte markiert werden.

Wir gehen zunächst auf Frage 1. Die Lösung hierzu ist recht einfach: wir können beliebige Texte nehmen, von denen wir sicher sind, dass sie nur korrekte Worte enthalten. Diese Texte müssen wir *tokenisieren*; ich gehe hier nicht auf diese Prozedur ein, und erkläre nur kurz was ein tokenisierter Text ist. Wir nehmen ein Symbol $\square$, das nicht im Text selber vorkommt, und setzen dieses Symbol zwischen alle *token* die im Text vorkommen (was als token zählt eine praktische Frage, die uns hier nicht interessiert). Nehmen wir an, wir haben einen Automaten $\mathfrak{A}_{tok}$, der genau den tokenisierten Text erkennt. Was wir jetzt machen müssen ist diesen Text zu *typisieren*: damit meine ich, aus dem Automaten, der den tokenierten Text erkennt, bauen wir einen Automaten, der genau die Worte erkennt, die also token im Text vorkommen. Das läuft wie folgt: wir bilden die Komposition mit den Typisierungsautomat $\mathfrak{T}_{typ}$, und dann die zweite Projektion π_2. $\mathfrak{T}_{typ}$ sieht wie folgt aus. Hier lassen wir $\forall$ als Variable für alle Eingaben stehen, Σ nur für die Buchstaben im (untokenisierten!) Text.



Wir definieren $Lex := \pi_2(\mathfrak{A}_{tok} \circ \mathfrak{T}_{typ})$. Damit haben wir ein Lexikon, d.h. ein Automaten, der jedes Wort in unserem Text erkennt. Wie können wir prüfen, ob in einem Text jedes Wort korrekt ist? Ganz einfach: wir bilden die $*$-Hülle der Sprache (des Lexikons), das er erkennt. Der resultierende Automat erkennt jede Verkettung von bekannten Wörtern (mit Leerzeichen dazwischen). Wenn wir nun einen Text als Eingabe liefern, dann wird der Text also entweder akzeptiert – alle Worte sind korrekt – oder abgelehent: es gibt ein falsches Wort darin.

Wir sehen dass das unbefriedigend ist: wenn wir ein Dokument von 10 Seiten bei unserem Automaten abliefern, und er sagt uns es enthält (mindestens) einen Fehler, werden wir uns bedanken.

Was wir brauchen ist etwas anderes: er soll uns den Eingabetext ausgeben, aber die Wörter, die er nicht kennt, auf eine eindeutige Art und Weise markieren. Nehmen wir dafür die Klammern $\langle, \rangle$. Wenn wir nun eine Eingabe haben wie

(1) Hunde esssen Hundefutter

dann möchten wir, unter realistischen Annahmen, die Ausgabe

(2) Hunde $\langle$ esssen $\rangle$ Hundefutter

Wie machen wir das? Nehmen wir an, *Lex* ist unser Lexikon; $Lex \subseteq \Sigma^*$, $\langle, \rangle \notin \Sigma$. Wir nutzen nun die Abschlusseigenschaften von regulären Sprachen und Relationen: da wir Abschluss unter Komplement haben, bilden wir die Sprache $\Sigma^* - Lex$; weiterhin gibt es eine rationale Relation $R_{\langle\rangle} := \{(w, \langle w \rangle\}$. Was wir nun bilden ist die Relation

(11) $((\Sigma^* - Lex) \circ R_{\langle\rangle}) \cup Lex$.

Diese Relation berechnet die Identität auf Wörtern in *Lex*, und für $w \notin Lex$ berechnet es $w \mapsto \langle w \rangle$. Jetzt müssen wir nur noch die $*$-Hülle bilden, dann sind wir fertig:

(12) $S_1 := ((\Sigma^* - Lex) \circ R_{\langle\rangle}) \cup Lex\square)^*$.

$\square$ steht für das Leerzeichen; wir machen hier die vereinfachende Annahme, dass hinter jedem Wort der Eingabe ein Leerzeichen steht. Dieser Automat berechnet genau was wir möchten: es markiert Worte, die nicht bekannt sind, auf eine eindeutige Art und Weise, und lässt alles andere unverändert.

6.2 Ein Spell-Checker mit automatischer Korrektur

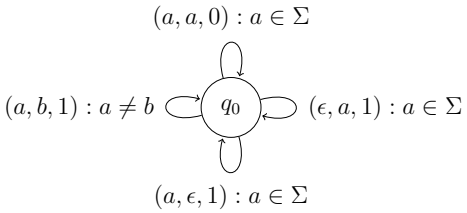
Wir kommen nun zum zweiten Punkt: wir möchten einen spell-checker S_2, der uns nicht nur die unbekannten Worte markiert, sondern gleichzeitig durch das ähnlichste bekannte Wort ersetzt. Hierzu können wir S_1 benutzen; gleichzeitig müssen wir die *minimum edit distance* zwischen einen beliebigen $w \in \Sigma^* - Lex$ und einem Wort in *Lex* finden; genauer gesagt:

Definition 3 *Die minimum edit distance zwischen (w, v) ist definiert als die minimale Anzahl von Schritten von Substitution, Eliminierung und Einfügung, mit denen man w in v transformieren kann; wir schreiben also $MED : \Sigma^* \times \Sigma^* \to \mathbb{N}$.*

Was uns interessiert ist also nicht $MED(w, v)$, sondern uns interessiert, für jedes Wort des Eingabetextes; uns interssiert für jedes Wort $w \in \Sigma^* - Lex$,

(13) $argmin_{v \in Lex} MED(w, v)$,

wobei argmin das duale Gegenstück zu argmax ist. Wie berechnen wir diese Funktion? Zunächst nehmen wir den gewichteten Automaten mit einem Zustand, der für jedes Wortpaar (w, v) alle möglichen edit-Distanzen berechnet; wir bekommen die beste Lösung, $MED(w, v)$, dann dadurch, dass wir im tropischen Halbring das Minimum nehmen:



Nennen wir diese Relation MED. Als nächstes bilden wir die Komposition mit Lex:

(14) $MED \circ Lex$

Der resultierende Automat ist ein gewichteter Automat: wir gehen einfach, wie immer, davon aus, dass jede Kante im ungewichteten Lex das Gewicht $\overline{1}$ hat, was in unserem Fall (tropischer Halbring) 0 ist. Dieser Automat ist eine Funktion $\Sigma^* \times Lex \to \mathbb{R}$: jede Eingabe wird mit einem Wort in Lex in Relation gesetzt mit einem gewissen Gewicht. Was uns interessiert ist, für eine jede Eingabe w,

(15) $argmin_{v \in Lex} MED \circ Lex(w, v)$.

Leider weiß ich nicht, ob und wie man einen derartigen *argmin*-Automaten berechnen kann, nicht einmal für den tropischen Halbring (für andere Halbringe ist es leicht zu zeigen, dass das nicht geht). Wir müssen uns daher auf eine andere Lösung verlegen. Bevor wir allerdings weiter machen, müssen wir noch ein weiteres Zwischenproblem lösen: wir hätten gerne, dass die Änderungen, die unser Automat vornimmt, markiert sind, und zwar wieder durch die Klammern $\langle, \rangle$. Das lässt sich sehr einfach konstruieren; zunächst aber müssen wir eine Unterscheidung treffen zwischen Worten in Lex und außerhalb. Insbesondere haben wir das Problem, dass wir dafür sorgen müssen, dass es immer besser ist, ein Wort nicht einzuklammern als es einzuklammern. Deswegen führen wir nun die Relation $Q_{\langle\rangle}$ ein:

(16) $Lex_{\langle\rangle} := \{\langle, \langle\} \cdot (Lex) \cdot \{\rangle, \rangle\}$

Wir haben also eine Relation, die nur eingeklammerte Worte enthält. Der Grund dafür wir gleich sichtbar werden; der entscheidende Schritt ist die Relation

(17) $S_2' := (MED \circ Lex_{\langle\rangle}) \cup Lex$

Diese Relation berechnet die Identität auf *Lex*, und zwar mit Gewicht 0 (Standardgewichtung). Außerdem berechnet sie für jede Eingabe (inklusive *Lex*) ein Wort in $Lex_{\langle\rangle}$, mit der dazugehörigen MED. Der Trick ist nun folgender: da nach unserer Annahme $\langle, \rangle$ in gar keiner Eingabe vorkommen, ist die MED zwischen Eingabe und Ausgabe in $Lex_{\langle\rangle}$ immer *mindestens* 2. Das heißt: wenn wir am Ende den kürzesten Pfad suchen, ist für Wörter in *Lex* die Identität immer am günstigsten, während alle Wörter nicht in *Lex* zwangsläufig eingeklammert werden. Damit haben wir einen gewichteten Transduktor, der einzelne Worte transduziert. Das ist natürlich zu wenig; wir bilden also die $*$-Hülle.

(18) $S_2 = (S_2'\Box)^*$

Das ist nun leider noch nicht genug: was wir brauchen ist ja die eine Ausgabe, die uns die MED liefert! Ich weiß leider nicht, wie und ob das allgemein funktioniert; wir können das also nur für einen gewissen Text berechnen: sei T ein Text; mit $\mathfrak{A}_T$ bezeichnen wir den linearen Automaten, der genau diesen Text erkennt. Was wir nun bilden ist

(19) $\mathfrak{A}_T \circ S_2$

Der kürzeste Pfad durch diesen Automaten kann in linearer Zeit berechnet werden, und liefert uns die Ausgabe mit der minimum edit distance, die wir suchen.

6.3 Ein Spell-Checker mit Wortwahrscheinlichkeiten

Mit Wortwahrscheinlichkeiten meinen wir folgendes: wir schätzen Wahrscheinlichkeiten von Worten, und möchten dass diese Wahrscheinlichkeiten in die Korrektur einfließen: je wahrscheinlicher ein Wort, desto eher möchten wir es als Korrektur haben. Natürlich ist das kein Ersatz für MED, wir möchten ja nicht jedes unbekannte Wort als "der" korrigieren (das häufigste Wort). Außerdem ist klar, dass dieser Ansatz oft mit MED in Konkurrenz und Konflikt steht. Daher ist die Frage, ob solch eine Erweiterung sinnvoll ist, eine empirische und hängt von der Anwendung ab. Daher möchten wir hier nur zeigen, inwiefern eine solche Erweiterung machbar ist.

Mit den üblichen Methoden (Zählung der Vorkommen eines Wortes dividiert durch Anzahl der Worte im Text) können wir die Wahrscheinlichkeit eines

Wortes schätzen. Die ist für uns noch nicht benutzbar, wir müssen sie erst in ein Gewicht für den tropischen Halbring transformieren; wir machen das wie immer mit:

(20) $gew(w) = -log(p(w))$,

wobei $p(w)$ die geschätzte Wahrscheinlichkeit ist; das ganze ist invers korreliert: je kleiner $p(w)$, desto größer $gew(w)$. Wir nennen die gewichtete Sprache, die jedem Wort (in *Lex*) sein Gewicht zuweist, *Gew*. Wir können nun einfach folgendes machen: wir bilden die Komposition

(21) $S_2' \circ Gew$

Nach der Definition von gewichteter Komposition gilt:

(22) $(S_2' \circ Gew)(w, v) = S_2'(w, v) \otimes Gew(v) = S_2'(w, v) + Gew(v)$

Wir addieren also damit einfach das Gewicht. Wir können von der neuen Relation wiederum $*$-Hülle und kürzesten Pfad berechnen.

7 Sprachmodelle mit Gewichteten Automaten

Sei Σ ein Alphabet (oder Lexikon). Ein Sprachmodell ist eine Abbildung $\mathcal{M} : \Sigma^* \to \mathbb{R}$, die jedem Wort/Satz ein Gewicht zuweist. Was das beste ist, hängt vom Halbring ab: im reellen Halbring ist je größer je besser, im tropischen Halbring umgekehrt. Es ist klar dass jeder gewichtete Automat ein Sprachmodell darstellt; die Frage ist: wie lernen wir das Sprachmodell mit den Daten? Sprachmodelle, die man von Daten lernt, sind gewöhnlich n-Grame, da in diesen alle Faktoren overt sind und daher die verlässliche Lernbarkeit gegeben ist.

n-Grame basieren auf Markov-Ketten. Deren wichtigste Eigenschaft ist folgende: Ereignisse beeinflussen die Wahrscheinlichkeiten von Nachfolgeereignissen, aber nur einem begrenzten Abstand. Wir werden nun eine formale Definition liefern.

Definition 4 *Sei $S_n : n \in \mathbb{N}$ eine (endliche oder unendliche) Reihe von Ergebnissen, $X_n : n \in \mathbb{N}$ Reihe von Zufallsvariablen auf S_n. $X_n : n \in \mathbb{N}$ ist eine Markov-Kette $m - ter$ Ordnung, falls $P(X_{t+1} = x_{t+1}|X_t = x_t, X_{t-1} = x_{t-1}, ..., X_1 = x_1) = P(X_{t+1} = x_{t+1}|X_t = x_t, X_{t-1} = x_{t-1}, ..., X_{t-m} = x_{t-m})$*

Das bedeutet soviel wie: nur die letzten m Ergebnisse beeinflussen die Wahrscheinlichkeit von $P(X_t = x_t)$, alle anderen sind irrelevant. Beachten Sie, dass Ketten von Ereignissen der Form: der n-te Wurf ist Zahl, der $n+1$-te Wurf ist Kopf etc., wo alle Ereignisse unabhängig sind, Markov Ketten 0-ter Ordnung sind.

Wir können effektiv und effizient Parameter aus Daten schätzen mit der Maximum-Likelihood Schätzung. Wenn wir also annehmen, dass Buchstaben in Texten zufällig verteilt sind, dann können wir, gegeben einen Text der groß genug ist um einigermaßen zuverlässig zu sein, effektiv schätzen was die zugrundeliegenden Parameter sind.

Sei T ein Text. Wir bezeichnen mit $\mathtt{a}(\mathsf{T})$ die Anzahl von $\mathtt{a}$s in T etc., mit $|\mathsf{T}|$ bezeichnen wir die Anzahl der Zeichen in T. Wir haben gesehen dass wir die Maximum Likelihood für $p(\mathtt{a})$ effktiv schätzen können mit $\frac{\mathtt{a}(\mathsf{T})}{|\mathsf{T}|}$. Wir erweitern unsere Schätzung nun für Markov-Ketten. Einfachheit halber nehmen wir zunächst eine Markov-Kette erster Ordnung als Modell, obwohl das natürlich inadäquat ist, wie wir gesehen haben. Zunächst machen wir folgende Annahme: wir erweitern unser Alphabet Σ, das bereits das Leerzeichen enthält, um die Zeichen $\#_a, \#_e \notin \Sigma$. Wir nehmen an, dass $\#_a$ (nur) am Anfang jedes Textes steht, $\#_e$ nur am Ende. Uns interessiert natürlich nicht die Wahrscheinlichkeit von # selbst, sondern die Wahrscheinlichkeit, dass ein Buchstabe am Anfang eines Textes steht. Diesen Fall müssen wir natürlich gesondert betrachten, denn in diesem Fall haben wir keine Vorgängerzustände, auf die wir uns berufen können. Wir müssen dabei beachten, dass wir für verlässliche Schätzungen für $P(\mathtt{a}|\#_e)$ eine Vielzahl von Texten betrachten müssen, da wir pro Text nur eine solche Folge haben.

Wir möchten zunächst die Wahrscheinlichkeit von $P(\mathtt{a}|\mathtt{x})$ für alle $\mathtt{x} \in \Sigma$ berechnen. Wir tun das auf eine denkbar einfache Art und Weise: wir erweitern unsere Notation $\mathtt{a}(\mathsf{T})$ auf Worte, so dass $\mathtt{abc...}(\mathsf{T})$ die Anzahl aller Vorkommen von $\mathtt{abc...}$ in T ist. Wir sagen nun:

$$\hat{P}(\mathtt{a}|\mathtt{b}) := \frac{\mathtt{ba}(\mathsf{T})}{\mathtt{b}(\mathsf{T})}, \tag{23}$$

wobei $\hat{P}$ die von uns geschätzte Wahrscheinlichkeit bezeichnet. Diese Schätzung erlaubt es uns, für alle $\mathtt{a},\mathtt{b} \in \Sigma$ die Wahrscheinlichkeit $\hat{P}(\mathtt{a}|\mathtt{b})$ zu schätzen.

Diese Methode lässt sich leicht auf beliebige Markov-Ketten n-ter Ordnung verallgemeinern: sei $\vec{\mathtt{w}}$ ein Wort mit $|\vec{\mathtt{w}}| = n$; dann ist

$$\hat{P}(\mathtt{a}|\vec{\mathtt{w}}) := \frac{\vec{\mathtt{wa}}(\mathsf{T})}{\mathtt{a}(\mathsf{T})}. \tag{24}$$

Mit diesen bedingten Wahrscheinlichkeiten können wir nicht ohne weiteres zu den unbedingten Wahrscheinlichkeiten zurückkommen: wir haben zwar die bekannten Regeln zur bedingten Wahrscheinlichkeit und Partitionen, und bekommen:

$$\hat{P}(\mathtt{a}) := \sum_{|\vec{\mathtt{w}}|=n} \hat{P}(\mathtt{a}|\vec{\mathtt{w}})\hat{P}(\vec{\mathtt{w}}), \tag{25}$$

Allgemeiner ausgedrückt, für eine Markov-Kette n-ter Ordnung, $|\vec{\mathtt{w}}| \leq n$, haben wir

$$\hat{P}(\mathtt{a}|\vec{\mathtt{w}}) := \sum_{|\mathtt{x}\vec{\mathtt{w}}|=n} \hat{P}(\mathtt{a}|\mathtt{x}\vec{\mathtt{w}})\hat{P}(\vec{\mathtt{x}}), \tag{26}$$

Aber um Wahrscheinlichkeiten zu berechnen, brauchen wir Wahrscheinlichkeit von Wörtern $\hat{P}(\vec{\mathtt{w}})$! Wahrscheinlichkeit von Wörtern berechnet sich wie folgt: sei $\vec{\mathtt{w}} = \mathtt{a_1a_2}...\mathtt{a_i}$. Dann ist

$$\begin{aligned}\hat{P}(\mathtt{a_1a_2a_3}...\mathtt{a_i}) &= \hat{P}(\mathtt{a_1})\hat{P}(\mathtt{a_2}|\mathtt{a_1})\hat{P}(\mathtt{a_3}|\mathtt{a_1a_2})...\hat{P}(\mathtt{a_n}|\mathtt{a_1}...\mathtt{a_{n-1}}) \\ &= \prod_{i=1}^{n} \hat{P}(\mathtt{a_i}|\mathtt{a_1}...\mathtt{a_{i-1}})\end{aligned} \tag{27}$$

Wir müssen also, für eine Markov-Kette n-ter Ordnung, alle Wahrscheinlichkeiten $\hat{P}(\mathtt{a}|\vec{\mathtt{w}}) : 0 \leq |\vec{\mathtt{w}}| \leq n$ schätzen. Mit diesem Wissen und einiger Mühe lässt sich natürlich zeigen:

$$\hat{P}(\mathtt{a}) := \sum_{|\vec{\mathtt{w}}|=n} \hat{P}(\mathtt{a}|\vec{\mathtt{w}})\hat{P}(\vec{\mathtt{w}}) = \frac{\mathtt{a}(\mathsf{T})}{|\mathsf{T}|}, \tag{28}$$

wie wir das erwarten. Wie benutzen wir also gewichtete Automaten, um n-Grame zu modellieren? Ein n-Gram Modell ist ein ganz einfacher Automat: da die Wahrscheinlichkeiten immer nur von den letzten n Zuständen anhängen, reicht es, sich eben diese zu merken. Wir setzen also $Q \subseteq \Sigma^n$, unsere Übergänge haben die Form $a\overline{w}b, c, \overline{w}bc, \alpha$, wobei $\alpha = P(c|a\overline{w}b)$. Mit dieser einfachen Methode können wir Markov-Ketten simulieren. So bekommen wir einen gewichteten Automaten $\mathfrak{A}$. Wir können also – dank der Modularität unserer Methode – diese Markov Kette in unseren Spell-checker einbauen: sei S_2 unser Spell-checker; wir konstruieren dann einfach:

$$S_2 \circ \mathfrak{A} \tag{29}$$

Natürlich gelten hier dieselben Erwägungen wie vorhin, so dass die Komposition einfach für jede Eingabe von S_2 die Gewichte von Eingabe/Ausgabe mit $\mathfrak{A}(x)$ (x die Ausgabe) multipliziert.

Eine Gegenfrage ist folgende: wir können natürlich mit Automaten *viel* mehr als mit Markov-Ketten. Kann man irgendwie die zusätzliche Ausdruckstärke benutzen? Das Problem ist, dass man zuviele Möglichkeiten hat, als dass man Wahrscheinlichkeiten sinnvoll schätzen könnte. Es gibt allerdings einige Möglichkeiten: z.B. können wir Wahrscheinlichkeiten schätzen, dass ein b auftritt, gegeben dass *irgendwo* vorher ein $\overline{w}$ auftritt. Wiederum muss man die Länge von $\overline{w}$ beschränken um sinnvolle Schätzungen zu bekommen, aber sonst spricht nichts gegen diesen Ansatz.

Wir können die Wahrscheinlichkeiten nach dem alten Muster schätzen; um das ganze auf gewichtete Automaten zu übertragen, muss man nur wiederum Zustände haben, die sich die Vorkommen von vorherigen Worten merken. Wenn die Länge dieser Worte beschränkt ist, dann bleibt auch die Anzahl der Zustände beschränkt.

8 Maschinelle Übersetzung

Es gibt drei große Probleme in der maschinellen Übersetzung: 1. das Erstellen von Modellen, 2. das Training von Modellen und 3. das Dekodieren von Modellen.

Wir gehen zunächst an Aufgabe 1. Ein MT-Modell nimmt eine Eingabe in einer "Sprache" und liefert eine Menge von Ausgaben in einer anderen "Sprache", jede mit einem gewissen Gewicht. Wir sind wieder ganz abstrakt, sagen wir haben zwei Alphabete Σ, T und $MT : \Sigma^* \times T^* \to \mathbb{R}$ ist ein Funktion.

Wir werden zunächst diese Funktion beschreiben. Normalerweise haben wir (in den einfachsten Modellen) zwei Komponenten:

1. Eine lexikalische Komponente $lex : \Sigma \times T \to \mathbb{R}$; sie gibt das Gewicht jeder einzelnen Übersetzung an.
2. Eine Umordnungskomponente $ord : \wp(\mathbb{N} \times \mathbb{N}_0) \to \mathbb{R}$.
3. Ein Sprachmodell, das das Gewicht des Ausgabesatzes berechnet.

Das Sprachmodell ist ein separates Modul, dass wir zunächst ausklammern. Der zweite Punkt Bedarf einiger Erklärung: mit $\wp(\mathbb{N} \times \mathbb{N}_0)$ meinen wir die Menge aller Teilmengen von $\mathbb{N} \times \mathbb{N}_0$, kurz gesagt die Menge der Relationen über natürlichen Zahlen. *ord* weist also jedem Element $R \subseteq \mathbb{N} \times \mathbb{N}_0$ ein Gewicht zu. Das repräsentiert die sog. Alinierung: das erste Wort im Eingabesatz wird repräsentiert als das $j$te Wort im Ausgabesatz, falls $(1, j) \in R$. NB: Wörter können mehrfach aliniert werden (mit mehreren Wörtern übersetzt), oder gar nicht: in diesem Fall haben wir die Alinierung $(i, 0)$. Wir müssen aber über jedes Eingabewort Rechnung ablegen – nicht aber für jedes Ausgabewort! Es können einfach Ausgaben dazu kommen.

Wie das Sprachmodell geschätzt wird, haben wir bereits gesprochen. Lexikalische Gewichte und Alinierungsgewichte werden einfach von wortalinierten Korpora geschätzt nach der Maximum Likelihood Methode. (Da kann man noch viel zu sagen, ich verweise auf meine anderen Skripte).

Was man am Ende berechnet ist folgendes:

$$MT(\overline{w}, \overline{v}) = \Big(\prod_{i=1,(i,j)\in R}^{|\overline{w}|} lex(w_i, v_j) \Big) ord(R) \tag{30}$$

wobei w_i, v_j der $i$te Buchstabe von $\overline{w}$ bzw. der $j$te Buchstabe von $\overline{v}$ ist. Ein wichtiges Problem gibt es hier allerdings: wir können $MT(\overline{w}, \overline{v})$ auf verschiedene Arten und Weisen berechnen je nachdem wie wir R wählen, und jede

gibt wahrscheinlich ein anderes Gewicht. Während wir in obiger Formel die (abstrakte) Multiplikation haben, brauchen wir nun die (abstrakte) Addition, um das Gesamtergebnis zu finden. Eine wichtige Vereinfachung, die normalerweise vorgenommen wird, ist folgende: wir vereinfachen *ord* zu *ord'*, welches nur einzelnen Paaren Gewichte zuweist, und für das Gesamtgewicht nehmen wir das Produkt:

$$(31) \quad ord(R) = \prod_{(i,j)\in R} ord(i,j)$$

Dadurch "vereinfacht" sich unsere obige Formel zu

$$(32) \quad MT(\overline{w},\overline{v}) = \prod_{i=1,(i,j)\in R}^{|\overline{w}|} lex(w_i,v_j)\, ord(i,j)$$

Da unsere Alinierung R immer von Daten geschätzt ist, ist sie immer ein endliche Menge. Daraus folgt rein mathematisch und mit roher Gewalt, dass wir MT in genau dieser Form auch als gewichteten endlichen Automaten implementieren können. Das ist aber kein sonderlich interessantes und aufschlussreiches Unterfangen. Stattdessen werden wir fragen, ob es eine genuin automatentheoretische Perspektive auf MT gibt.

Nehmen wir z.B. ein Satzpaar (ab, cd). Nehmen wir an, die plausibelste Übersetzung von a ist d, und von b c. Man kann das mit der Alinierung $\{(1,2),(2,1)\}$ einfach darstellen. Man kann aber auch sagen: Das Gewicht von (a,c) ist hoch, vorausgesetzt, dass danach (b,d) übersetzt wird. Wir können also unsere Zustände so kreieren, dass sie speichern:

1. die Worte, die wir noch übersetzen müssen, und
2. Worte, die wir schon übersetzt aber noch gar nicht gelesen haben.

Wir können Worte löschen aus beiden Listen, und das ist was die Kosten verursacht. Damit der Automat endlich bleibt, müssen wir natürlich eine Obergrenze für die Länge der Liste festlegen. Unsere Übergänge haben also die Form:

$$(33) \quad (M_1, (w_i, v_i, \alpha), M_2)$$

wobei M_1, M_2 Mengen sind von Paaren in $\Sigma \times \mathbb{N}$ und $\mathbb{N}_0 \times T$. Es gibt nun verschiedene Möglichkeiten:

1. $\alpha = 0$, und $M_2 = M_1 \cup \{(w_i, i), (i, v_i)\}$
2. $M_1 = M_2$ und $\alpha = lex(w_i, v_i) \otimes ord(i,i)$;
3. $M_2 = (M_1 - \{(j, v_j)\}) \cup \{i, v_i\}$, und $\alpha = lex(w_i, v_j) \otimes ord(i,j)$;

4. $M_2 = (M_1 - \{(j, v_j)\}) \cup \{(w_i, i)\}$, und $\alpha = \mathit{lex}(w_j, v_i) \otimes \mathit{ord}(j, i)$;

5. $M_2 = (M_1 - \{(j, v_j)\}) - \{(w_l, l)\}$, und $\alpha = \mathit{lex}(w_i, v_j) \otimes \mathit{ord}(i, j) \otimes \mathit{lex}(w_l, v_i) \otimes \mathit{ord}(i, l)$.

Zusätzlich müssen wir noch ϵ-Übergänge berücksichtigen, wir lassen die hier aber weg, um die Sache nicht unnötig kompliziert zu machen. Die akzeptierenden Zustände sind einfach die, in denen keine Elemente der Form (w, k) enthalten sind.

Das liefert uns eine getreue Übersetzung der normalen IBM-Modelle (leicht generalisiert), und die Gewichte können von den klassischen Schätzungen übernommen werden.

Ich kann aber auch unmittelbarer an das Problem gehen, und die Gewichte für die Übergänge direkt aus wortalinierten Korpora schätzen. Hier ergibt sich ein wichtiger Unterschied: in klassischen Ansätzen werden Alinierungswahrscheinlichkeiten unabhängig von Wortwahrscheinlichkeiten geschätzt. Für unseren Automaten ist diese Beschränkung künstlich und unnötig, denn die Alinierung ist natürlich *nicht* unabhängig von den Wörtern die aliniert werden! Das ist sehr plausibel und beseitigt eine große Schwäche. Andererseits gibt es damit viel mehr Daten, von denen wir einen viel kleineren Einblick haben. Von daher ist es fragwürdig, ob wir damit tatsächlich bessere Ergebnisse erzielen. Aber immerhin gibt es eine genuin automatentheoretische Perspektive auf MT-Modelle.

9 Weitere Module

Das Sprachmodell kann an unser Übersetzungsmodell angehängt werden mit Komposition. Dekodierung besteht darin, von allen Übersetzungen die beste zu finden. Wir tun das auf die gewohnte Art und Weise mit den *shortest path* Algorithmus gegeben die Komposition mit einer Eingabe. Was das Lernen angeht: wir haben bereits einzelne Worte darüber verloren. Einer der wichtigsten Schritte in der maschinellen Übersetzung ist das automatische erstellen von wortalinierten Korpora aus satzalinierten Korpora. Für diese Aufgabe spielen gewichtete Automaten aber (noch) keine Rolle.

10 Baumautomaten

10.1 Bäume als Strukturen

(Endliche) Automaten können über verschiedenen Datenstrukturen operieren, und der Fall wo sie auf Zeichenketten operieren ist tatsächlich nur ein Spezialfall. Wir werden hier nur eine Erweiterung betrachten, nämlich die Erweiterung auf Bäume. Das ist tatsächlich eine Generalisierung, denn wir können Ketten als einen Spezialfall von Bäumen auffassen: stellen wir uns einfach einen Baum vor, in dem jeder Knoten nur eine Tochter hat, dann haben wir eine Kette.

Zunächst fragen wir uns, was Bäume sind. Bäume sind recht allgemeine Strukturen; die allgemeine Definition lautet: ein Baum ist eine partielle Ordnung, in der für jedes Objekt die Menge seiner Vorgänger linear geordnet sind, und der ein minimales Element enthält. Bäume sind also unter anderem die Strukturen $([0,1],\leq)$ und $(\Sigma^*, \mathit{pref})$, wobei *pref* die Präfixrelation ist. Uns interessieren aber nicht Bäume im allgemeinen, sondern nur ganz bestimmte Bäume:

1. Bäume, die endlich sind (oder etwas weiter gefasst: wohlfundiert);
2. Bäume, deren Knoten *Namen* haben.
3. Bäume, deren Knoten, die sich nicht dominieren, vollständig geordnet sind.

Der erste Punkt sollte klar sein was Endlichkeit angeht. Man kann Automaten auch auf unendlichen Bäumen benutzen, allerdings müssen sie wohlfundiert sein, d.h.: für jeden Knoten ist die Menge aller Vorgängerknoten endlich. Das schließt $([0,1],\leq)$ aus, nicht aber $(\Sigma^*, \mathit{pref})$. Der zweite Punkt ist folgender: in einem Baum sind die Knoten eindeutig definierte Objekte. Wir möchten, dass sie zusätzlich noch Namen haben, die für zwei verschiedene Knoten gleich sein können. Da das Objekt des Knotens (im Gegensatz zum Namen) die einzige Funktion hat, den Knoten zu identifizieren, ist es egal welches Objekt das konkret ist; wichtig ist nur, dass es nicht mit anderen Knoten zusammenfällt. Technisch gesprochen: uns interessieren Bäume nur bis auf Isomorphie. Der dritte Punkt ist folgender: ein Baum im klassischen Sinne spezifiziert keine Präzedenz zwischen Knoten, die einander nicht dominieren. Wir möchten, dass jede Menge von Knoten, das nicht durch Dominanz verbunden ist (sog. Anitketten), durch eine weitere Relation linear geordnet wird, die sog. Präzedenz. Außerdem soll Präzedenz die Dominanz respektieren: Wann immer a b präzediert, a c dominiert und b d, dann präzediert c auch d.

Uns interessieren nur die Bäume, die alle diese Kriterien erfüllen; man nennt diese Bäume auch vollständig geordnete endliche/wohlfundierte Bäume. Diese Bäume haben eine schöne Eigenschaft: wir können die Knoten einfach weglassen und durch deren Namen ersetzen, und sie dann als **Terme** schreiben. Ein Baum ist für unsere Zwecke einfach ein Term

(34) $a_0(a_{00}(a_{000},...,a_{00i}),...,a_{0i}(...))$,

wobei $a_{\vec{x}} : \vec{x} \in \mathbb{N}^*$ die Namen der Knoten sind; die Indizes hingegen sollen hier die Knoten selber identifizieren. Wenn wir die Namen aber ausschreiben, dann fallen die Indizes weg. Bäume sind z.B. $a(b,a)$ oder $a(b,c(a,b))$. Wir können in dieser Form, ebenso wie wir mit Σ^* die endlichen Ketten über Σ definieren, auch die endlichen Terme über Σ definieren: $\mathit{term}(\Sigma)$ ist die kleinste Menge so dass

1. falls $a \in \Sigma$, dann $a \in \mathit{term}(\Sigma)$, und
2. falls $t_1,...,t_i \in \mathit{term}(\Sigma)$, $a \in \Sigma$, dann ist $a(t_1,...,t_i) \in \mathit{term}(\Sigma)$.

10.2 Baumautomaten: Definition

Wir werden nun definieren, wie wir Automaten Terme einlesen lassen. Überlegen wir zunächst, wie es aussieht, wenn wir eine Zeichenkette $a_1...a_i$ als einen unär verzweigenden Baum auffassen, als Term $a_1(...(a_i)...)$. Der Automat ist im Startzustand, liest die Wurzel, geht in einen neuen Zustand, liest die (einzige) Tochter der Wurzel, geht in den nächsten Zustand usw. Wenn er an einem Blatt angekommen ist, muss er in einem akzeptierenden Zustand sein. Für Baumautomaten müssen wir diese Prozedur generalisieren. Nehmen wir einen Baum $a_1(a_2(..), a_3(..))$. Hier liest der Baum die Wurzel a_1 im Startzustand. Als nächstes *spaltet er sich auf*, d.h. er geht in zwei Zustände, und in dem einen Zustand prüft er, ob er $a_2(...)$ akzeptiert, und in dem anderen prüft er, ob er $a_3(...)$ akzeptiert. Am Ende werden wir, so der Baum endlich ist, eine positive oder negative Antwort erhalten für jedes Blatt des Baumes. Was dann aber noch fehlt ist folgenes: wir müssen die Antworten zu einer einzigen Antwort zusammensetzen. Damit haben wir das Prinzip des einfachen Baumautomaten beschrieben; wir kommen nun zu den Definitionen.

Definition 5 *Ein top-down Baumautomat ist ein Tupel* $\mathfrak{A} = (Q, q_0, \delta, F, \Sigma)$*, wobei alles wie bei endlichen Automaten ist, bis auf* $\delta \subseteq Q \times \Sigma \times (\bigcup_{n\in\mathbb{N}} Q^n)$.

Was mit der veränderten Übergangsfunktion ausgedrückt wird ist folgendes: eine Eingabe hat eben nicht nur *einen* Nachfolger, sondern eine beliebige endliche Anzahl davon, und jede bekommt einen Zustand zugewiesen. Die eigentlich entscheidende Definition ist die eines **Laufes** eines Automaten auf einem Baum in $term(\Sigma)$. Wir definieren die Funktion δ^* wie folgt: falls $t(t_1, ..., t_i) \in term(\Sigma)$, $q \in Q$, $(q, a, q_1, ..., q_i) \in \delta$, dann ist $(\delta^*(q_1, t_1), ..., \delta^*(q_i, t_1)) \in \delta^*(q, a(t_1, ..., t_i))$. Eine wichtige Rolle für die Akzeptanz spielt der **leere Baum**, den wir manchmal mit () repräsentieren. Falls wir einen atomaren Term der Form a haben, und $(q, a, q') \in \delta$, dann sagen wir $\delta^*(q, a) = q'$; wir lassen den leeren Term also weg. Wir sagen, dass $\mathfrak{A}$ einen Term t akzeptiert, falls es $(q_1, q_2, ..., q_i) \in \delta(q_0, t)$ gibt, so dass $q_1, ..., q_i \in F$. Wir müssen also auf jedem Blatt in einem akzeptierenden Zustand angekommen sein. Zusammenfassend kann man sagen, ein Baumautomat verhält sich wie ein normaler Automat auf allen Pfaden durch den Baum, nur dass er zusätzlich noch berücksichtigt, wieviele Töchter (Nachfolger) ein Knoten hat, und auf dem wievielten Nachfolger er seinen Pfad fortsetzt.

Es gibt nicht nur *top-down* Baumautomaten, sondern auch sog. *bottom-up* Automaten, die einen Baum von unten nach oben erkennen.

Definition 6 *Ein bottom-up Baumautomat ist ein Tupel* $\mathfrak{A} = (Q, \Sigma, F, \delta)$*, wobei alles wie gehabt bis auf* δ*:* δ *ist eine Familie von Funktionen* $\delta_a : a \in \Sigma$*,* $\delta_\sigma : (\bigcup_{n\in\mathbb{N}} Q^n) \to Q$.

Wir definieren nun den Begriff der Akzeptanz: wir erweitern wieder δ zu δ^*, so dass $\delta^*(a(t_1, ..., t_i)) = \delta_a(\delta^*(t_1), ..., \delta^*(t_i))$ f.a. $a \in \Sigma, t_1, ..., t_i \in term(\Sigma)$. Besondere Aufmerksamkeit verdient der Fall, wo $a = a()$ ein atomarer Baum ist; hier muss $\delta_b(q, a)$ definiert sein. Auf diese Art und Weise ist δ^* eine Abbildung

$\delta^* : term(\Sigma) \to Q$. Die Menge von Bäumen, die $\mathfrak{A}$ erkennt, ist definiert als $L(\mathfrak{A}) = \{t \in term(\Sigma) : \delta^*(t) \in F\}$.

Ein weiterer wichtiger Begriff in Zusammenhang mit Baumautomaten ist der der **regulären Baumgrammatik**. Eine reguläre Baumgrammatik ist ein Tupel $G = (\mathcal{N}, \Sigma, S, R)$, wobei $\mathcal{N}, \Sigma$ Mengen sind von Nichtterminalen und Terminalsymbolen, $S \in \mathcal{N}$ das Startsymbol, und R eine Menge von Regeln ist der Form

(35) $N \to a(\alpha)$,

wobei $N \in \mathcal{N}, a \in \Sigma$, und α ist ein regulärer Ausdruck über Symbole in $\mathcal{N}$. Reguläre Baumgrammatiken generieren Bäume; wir definieren die Ableitung wie üblich: Sei $\alpha[N] \in term(\Sigma \cup \mathcal{N})$. Damit meinen wir: wir haben einen Term, in dem N als ein Blatt vorkommt. Wir schreiben $t[N] \vdash_G t[a(t)]$, wobei $t[a(t)]$ das Ergebnis der Ersetzung von N durch $a(t)$ ist, gdw. $N \to a(\alpha) \in R$ und t eine Instanz von α ist (NB: α ist ja ein regulärer Ausdruck!). Mit $\vdash_G^*$ bezeichnen wir die reflexive, transitive Hülle von $\vdash_G$, und bekommen so $L(G) = \{t \in term(\Sigma) : S \vdash_G^* t\}$. Also alles wir bei kontextfreien Grammatiken, nur das wir Bäume generieren. Im folgenden Abschnitt werde ich die wichtigsten Ergebnisse zusammenfassen.

10.3 Die wichtigsten Ergebnisse in Kürze

Wir machen es kurz:

Lemma 7 *Sei $L \subseteq term(\Sigma)$ eine Menge von Bäumen. Die folgenden drei Aussagen sind äquivalent:*

1. *Es gibt eine reguläre Baumgrammatik G so dass $L = L(G)$.*
2. *Es gibt einen bottom-up Baumautomaten $\mathfrak{A}$ so dass $L = L(\mathfrak{A})$.*
3. *Es gibt einen top-down Baumautomaten $\mathfrak{A}$ so dass $L = L(\mathfrak{A})$.*

Eine Menge von Bäumen nennt man auch einfach eine Baumsprache. Die drei Charakterisierungen sind also äquivalent, und charakterisieren jeweils dieselbe Klasse von Baumsprachen. Wir nennen diese Klasse auch die Klasse der **regulären Baumsprachen**. Einen wichtigen Unterschied gibt es zwischen top-down und bottom-up Automaten. Wir sagen, ein Automat ist **deterministisch**, falls seine Übergangsrelation δ eine Funktion ist. Man beachte, dass wir bottom-up Automaten deterministisch definiert haben, top-down Automaten nicht; der Grund dafür ist folgender:

Lemma 8 *Für jeden nichtdeterministischen bottom-up Baumautomaten $\mathfrak{A}$ gibt es einen deterministischen bottom-up Baumautomaten $\mathfrak{A}'$, so dass $L(\mathfrak{A}) = L(\mathfrak{A}')$.*

Der Beweis ist wie üblich die Potenzmengenkonstruktion über Zustände. Für die top-down Methode funktioniert diese Konstruktion nicht:

Lemma 9 *Es gibt eine Baumsprache L, so dass es einen nichtdetermistischen top-down Automaten $\mathfrak{A}$ gibt mit $L = L(\mathfrak{A})$, aber keinen determistischen top-down Automaten $\mathfrak{A}'$ mit $L = L(\mathfrak{A}')$.*

Der Grund hierfür ist folgender: Falls $\mathfrak{A}'$ deterministisch ist, die Bäume $a(b,c)$, $a(c,b)$ akzeptiert, dann geht ist also $\delta^*(q_0, a) = (q, q')$; $q(b), q(c), q'(b), q'(c) \in F$. Folglich akzeptiert der Automat auch den Baum $a(b,b)$. Also gibt es keinen deterministischen top-down Automaten, der $\{a(b,c), a(c,b)\}$ erkennt. Diese Baummenge ist aber regulär:

Lemma 10 *Jede endliche Menge von Bäumen ist eine reguläre Baummenge.*

Das zu beweisen ist recht einfach: wir nehmen einfach für jeden auftretenden Fall einen Zustand im Baumautomaten; so bleiben wir immer mit endlich vielen Zuständen. Als nächstes sollten wir kurz besprechen, wir reguläre Baummengen mit kontextfreien Sprachen zusammenhängen. Sei t ein Baum. Mit $b(t)$ meinen wir die eine Zeichenkette w, die entsteht, wenn man die Label der Blätter von t hintereinanderschreibt. Man kann b auffassen als Funktion $b : term(\Sigma) \to \Sigma^*$. Wir heben diese Funktion auf die kanonische Art und Weise auf Mengen an.

Lemma 11 *Falls L eine reguläre Baumsprache, dann ist $b(L)$ eine kontextfreie Sprache; und falls L eine kontextfreie Sprache ist, dann ist gibt es eine reguläre Baumsprache L_B, so dass $L = b(L_B)$.*

Der Beweis ist in beide Richtungen einfach; wir können tatsächlich kontextfreie Grammatiken umformen in reguläre Baumgrammatiken, die deren Ableitungsbäume beschreiben: wir transformieren einfach alle Regeln der Form $N \to \alpha$ zu Regeln $\overline{N} \to N(\alpha)$; die Terminale der neuen Grammatik sind $\mathcal{N} \cup \Sigma$, die Nichtterminale sind die Menge $\{\overline{N} : N \in \mathcal{N}\}$. Wenn wir nun allerdings diese Bäume betrachten, dann sehen wir, dass Klasse der kontextfreien Ableitungsbäume, die wir auf diese Art und Weise erhalten, echt kleiner ist als die Klasse der regulären Baumsprachen. Ein einfaches Beispiel ist folgende Baumsprache: $\{a, a(a)\}$. Die ist endlich, also regulär, aber nicht kontextfrei, da wir auf kontextfreie Art und Weise mindestens noch $a(a(a)), a(a(a(a)))...$ ableiten können. Diese kontextfreien Ableitungssprachen heißen auch **lokale Baumsprachen**, und verhalten sich zu den regulären Baumsprachen genau so, wie sich die streng lokalen Sprachen zu den regulären Sprachen verhalten (dazu sage ich einiges im oben angeführten Automaten-Skript). Dazu gehört u.a. folgender Satz. Ein **Baumhomomorphismus** ist eine Abbildung $h : \Sigma \to T$, die wie folgt auf Terme erweitert wird: $h(a(t_1, ..., t_i) = h(a)(h(t_1), ..., h(t_i))$.

Lemma 12 *Für jede reguläre Baummenge L gibt es eine lokale Baummenge L' und einen Homomorphismus h, so dass gilt: $L = h[L']$. Umgekehrt, falls L' lokal ist, h ein Homomorphismus, dann ist $h[L']$ regulär.*

Der Beweis des ersten Teiles geht so: wir konstruieren ein Alphabet das aus Paaren in $\Sigma \times Q$ besteht; dadurch können wir den Automaten mit einfachen

Buchstaben simulieren. Der zweite Teil ist offensichtlich; wir können z.B. den Homomorphismus einfach in der Baumgrammatik einsetzen und bekommen das gewünschte Ergebnis.

Reguläre Baumsprachen haben sehr viele Eigenschaften mit regulären Sprachen gemeinsam: neben der Charakterisierung durch Automaten, Grammatiken auch die Charakterisierung durch eine Logik; und außerdem die äußerst wichtigen Abschlusseigenschaften:

Lemma 13 *Seien $L_1, L_2 \subseteq term(\Sigma)$ reguläre Baummengen. Dann sind $L_1 \cup L_2, L_1 \cap L_2, term(\Sigma) - L_1$ ebenfalls reguläre Baummengen.*

Wir haben also Abschluss unter Vereinigung unter Vereinigung, Schnitt (man zeigt dass mit den Standardkonstruktionen für Automaten) und Komplement (das folgt nur, weil wir die bottom-up Automaten deterministisch halten können!). Dieses Ergebnis ist eigentlich überraschend, denn die kontextfreien Sprachen sind nicht abgeschlossen unter Schnitt und Komplement, und die regulären Baumsprachen sind in gewissem Sinne äquivalent. Der Unterschied ist dass wir bei Baumsprachen die mengentheoretischen Operationen über die Bäume definieren, nicht über die Ketten. Wenn wir $b[L_1] \cap b[L_2]$ schneiden, dann gibt es im allgemeinen Fall keine reguläre Baumsprache L_3, so dass $b[_3] = b[L_1] \cap b[L_2]$.

11 Baumtransduktoren

11.1 Allgemeines

Das war soz. der einfache Teil. Richtig kompliziert wird es, wenn man endliche Baumtransduktoren verwendet, denn hier werden viele Äquivalenzen hinfällig, wie etwa bottom-up≡top-down; ebenso sind reguläre Baummengen nicht unbedingt abgeschlossen unter Transduktionen (im Gegensatz zu regulären Sprachen. Es gibt also nicht mehr *den* Begriff des endlichen Baumtransduktors, sondern einige Begriffe, die man sorgsam auseinanderhalten muss. Eine sehr einfache Klasse von "Transduktoren" haben wir bereits kennengelernt, nämlich die Baumhomomorphismen, die in einer einfachen Form von Transduktor mit nur einem Zustand entsprechen. Wir werden jetzt aber weit über dieses Modell hinausgehen.

Zunächst brauchen wir einige Begriffe. *Var* ist eine abzählbare Menge von Variablen $\{x_1, x_2...\}$. Mit $term_{Var}(\Sigma)$ denotieren wir folgende Menge:

1. falls $x \in \Sigma \cup Var$, dann ist $x \in term_{Var}(\Sigma)$;
2. falls $a \in \Sigma$, $t_1, ..., t_i \in term_{Var}(\Sigma)$, dann ist $a(t_1, ..., t_i) \in term_{Var}(\Sigma)$.
3. sonst ist nix in $term_{Var}(\Sigma)$.

Wir lassen also Variablen nur als Blätter stehen, sonst nirends. Wir nehmen folgende Konvention an: mit $t[x_1, ...x_i]$ bezeichnen wir einen Term in $term_{Var}(\Sigma)$,

der genau die Variablen $x_1, ...x_i$ enthält. Die Nummerierung der Variablen ist hierbei sehr wichtig (wir identifizieren sie darüber), aber sie gibt *nicht* die Reihenfolge an, in der die Variablen in $t[x_1, ...x_i]$ auftreten. Mit $term^i_{Var}(\Sigma)$ denotieren wir die Menge aller Terme in $term_{Var}(\Sigma)$ der Form $t[x_1, ...x_i]$, die *nur* die Variablen $x_1, ...x_i$ enthalten.

Unsere Transduktoren basieren nun auf Funktionen $f_i : \Sigma \to term^i_{Var}(\Sigma)$. Um unser einfachstes Transduktorenmodell möglichst einfach zu präsentieren, machen wir nun folgende Annahme (die man oft allgemein für Baumautomaten macht): wir nehmen an unser Alphabet Σ ist **sortiert**, d.h. jeden $a \in \Sigma$ wird ein **Rang** $r(a) \in \mathbb{N}$ zugewiesen, oder anders gesagt, es gibt eine Funktion $r : \Sigma \to \mathbb{N}$. Wenn Σ sortiert ist, dann ist $term(\Sigma)$ wie folgt definiert: falls $a \in \Sigma$, $r(a) = 0$, dann ist $a \in term(\Sigma)$; und falls $t_1, ..., t_i \in term(\Sigma)$, $r(a) = i$, dann ist $a(t_1, ..., t_i) \in term(\Sigma)$; sonst nix.

11.2 Top-down Modelle

Wir wissen also, wenn wir ein label sehen, wieviele Kinder wir darunter zu erwarten haben; das ist insbesondere wichtig für top-down Ansätze. Wir kommen nun zu unserem ersten Transduktoren-Modell. Einfachheit halber nehmen wir an, dass Eingabe und Ausgabealphabet identisch sind.

Definition 14 *Ein $T1$-Transduktor ist ein Tupel $\mathfrak{T} = (Q, q_0, \Sigma, \delta)$, wobei Σ sortiert ist, $\delta \subseteq Q \times \Sigma \times \Sigma \times (\bigcup_{n\in\mathbb{N}} Q^n)$.*

Die Definition ist unschön; einfacher gesag: unsere Übergänge haben die Form: $(q, a, b, q^{r(b)})$; d.h. wir brauchen exakt soviele Zustände, wie der Rang von a beträgt. Die Definition ist klar: es ist alles wie beim top-down Automaten, nur dass der Buchstabe, den wir gelesen haben, nicht gelöscht sondern ersetzt wird; und am Ende haben wir keine Folge von (akzeptierenden) Zustände da stehen, sondern einen neuen Baum. Das ist der Grund, warum es kein F gibt. Wir können die akzeptierenden Zustände auch noch hinzufügen, das verkompliziert aber das Verständnis und fügt nichts essentiell neues (im Vergleich zu Transduktoren auf Zeichenketten und einfachen Baumautomaten) hinzu. $T1$-Transduktoren sind zugegebenermaßen langweilig: sie ändern nicht die Struktur eines Baumes, nur die labels. Formaler gesagt: sei t ein Baum, $\mathfrak{T}$ ein $T1$-Transduktor. Dann gibt es 2 Baumhomomorphismen h_1, h_2 so dass $h_1(t) = h_2(\mathfrak{T}(t))$. Wir führen also eine Erweiterung ein:

Definition 15 *Ein $T2$-Transduktor ist ein Tupel $\mathfrak{T} = (Q, q_0, \mathcal{F}, \Sigma, \delta)$, wobei $\mathcal{F}$ eine Menge von Funktionen ist der Form $f : \Sigma \to term_{Var}(\Sigma)$, so dass $f : a \mapsto term^{r(a)}_{Var}(\Sigma)$, und $\delta \subseteq Q \times \Sigma \times \mathcal{F} \times (\bigcup_{n\in\mathbb{N}} Q^n)$.*

Etwas einfacher gesagt: unsere Übergänge haben die Form $(q, a, f, q^{r(a)})$, wobei $f : a \mapsto term^{r(a)}_{Var}(\Sigma)$. Der Gedanke hinter dieser Definition ist folgender: das label a wird durch einen vollständigen (komplexen) Term ersetzt; die Kinder von a werden mithilfe der Variablen indiziert und darin eingebaut. Wir

werden hier nun die Transduktion definieren. Nimm an, $\mathfrak{T}$ ist in Zustand q; wir definieren

(36) $\mathfrak{T}_q(a(t_1, ..., t_i)) = f(a)[(\mathfrak{T}_{q_1}(t_1))/x_1, ..., (\mathfrak{T}_{q_i} t_i)/x_i]$

für $(q, a, f, (q_1, ...q_i)) \in \delta$. Diese Gleichung bedarf einiger Erklärung. Sei t ein Baum. Mit $t[x_1, .., x_i]$ meinen wir: es gibt Blätter, die mit Variablen gelabelt sind, und das sind genau die Variablen $x_1, ..., x_i$. Z.B. $t[x_1, x_2]$ kann eine Abkürzung sein von $a(b(x_2), x_1)$. Angenommen, $t_1, ...t_i$ sind ebenfalls Bäume. Dann ist $t[t_1/x_1, ..., t_i/x_i]$ der Baum, der resultiert falls wir den Baum t_1 an die Stelle von x_1 setzen, ..., und den Baum t_i an die Stelle von x_i setzen. Also sei $t_1 = a(b, c)$, $t_2 = b(c)$, dann wird in unserem Beispiel $t[t_1/x_1, t_2/x_2] = a(b(b(c)), a(b, c))$. NB: es ist nicht die Reihenfolge, in der Variablen im Term auftreten, die über Substitution entscheidet, sondern ihr Name! In der Notation $t[x_1, .., x_i]$ sagen wir *nicht* über die Reihenfolge, in der die Variablen im Term auftreten!

Was wir also oben tun ist: wir ersetzen a durch einen Baum mit ebensovielen Variablen wie a Kinder hat, und die Kinder von a werden wiederum für die Variablen eingesetzt. Allerdings hat die Sache einen Haken: wir setzen nicht einfach die Kinder ein, sondern die Kinder werden weiter modifiziert durch den Automaten. Das ist die induktive Bedingung, durch die sichergestellt wird dass der ganze Baum durchlaufen wird. Falls ein Knoten keine Kinder hat, dann wird er einfach durch einen Baum ersetzt und fertig.

Dieses Modell ist natürlich *viel* mächtiger als $T1$; wir werden kurz die Möglichkeiten besprechen, die es bietet. 1. Wir können Knoten durch ganze Teilbäume ersetzen; das ist offensichtlich. 2. Wenn ein Knoten mit label a zwei Kinder hat, dann können wir die Reihenfolge der Kinder *beliebig verändern*: wir können also den gesamten Teilbaum, der vom ersten Kind dominiert wird, vertauschen mit dem Teilbaum, der vom i-ten Kind dominiert wird. Das ist ebenfalls klar, wenn wir bedenken, dass die Notation $t[x_1, ..., x_i]$ nichts über die Reihenfolge von $x_1, ..., x_i$ aussagt. Soweit sogut – es gibt aber noch viel mehr Möglichkeiten: wir können 3. einen kompletten Teilbaum *löschen*: denn $term^i_{Var}(\Sigma)$ sagt nur, dass die Variablen in $\{x_1, ..., x_i\}$ enthalten sind, nicht aber das jede Variable vorkommen muss! Wenn wir also $r(a) = 3$, $f(a) = a(x_1, x_3)$ haben, dann bedeutet dass, dass der Teilbaum, den das zweite Kind dominert, einfach ersatzlos gestrichen wird! Wir können aber auch 4. Teilbäume *klonen*, d.h. wir erstellen mehrere Kopien von ihnen. Denn wir haben nirgends festgelegt, dass eine gewissen Variable nur *einmal* in einem Term auftreten darf. Wir können also, für $r(a) = 1; r(b) = 2$ haben: $f(a) = b(x_1, x_1)$. Damit wird also $a(t) \mapsto b(t, t)$.

Wir sind also bereits außerordentlich mächtig, und haben mehr als wir normalerweise möchten. Wir sagen daher eine Abbildung $f : a \mapsto t[x_1, ..., x_{r(a)}$ ist *nicht-löschend*, falls jede Variable $x_1, ..., x_{r(a)}$ in dem Term vorkommt; eine Abbildung ist *linear*, falls jede Variable höchstens einmal vorkommt. Entsprechend können wir sagen, dass $T2$-Modelle nicht-löschend bzw. linear sind, falls alle Funktionen nicht-löschend bzw. linear sind.

Das sind wichtige und interessante Beschränkungen; es gibt aber noch eine wichtige Erweiterung unseres Modelles (die wir aber nur informell besprechen werden): bislang haben wir auf der Eingabeseite immer nur einen einzelnen Buchstaben spezifiziert. Das lässt sich ändern: wir können die Funktionen in $\mathcal{F}$ erweitern zu Funktion $f : t[x_1, ...x_i] \to t'[x_1, ..., x_i]$. (35) wird dann zu:

(37) $\mathfrak{T}_q(t\langle t_1, ..., t_i\rangle) = f(t)[(\mathfrak{T}_{q_1}(t_1))/x_1, ..., (\mathfrak{T}_{q_i}t_i)/x_i]$

Hier meinen wir mit $t\langle t_1, ..., t_i\rangle$ dass t die Teilbäume $t_1, ..., t_i$ dominiert, und *sonst nichts.*

11.3 Bottom-up Modelle

Wir betrachten nun einige bottom-up Modelle. Ähnlich wie sich top-down Baumtransduktoren natürlich aus den Automaten ergeben, sind bottom-up Transduktoren eine natürliche Erweiterung aus bottom-up Automaten. Wir werden aber auch in diesem Fall die akzeptierenden Zustände weglassen, da sie das Konzept nur verkomplizieren.

Definition 16 *Ein T3-Transduktor ist ein Tupel $(Q, \Sigma, (\delta_a)_{a\in\Sigma})$, wobei Σ geordnet ist, und f.a. $a \in \Sigma$ $\delta_a \subseteq \bigcup_{n\in\mathbb{N}}(Q^n \times term^n_{Var}(\Sigma) \times Q$*

Das bedeutet, $\delta_a(q_1, ..., q_i) \subseteq term^n_{Var}(\Sigma) \times Q$. Das Prozedere ist wie bei normalen bottom-up Automaten: wir definieren $\delta^*(a(t_1, ..., t_i)) = \delta_a(\delta^*(t_1), ..., \delta^*(t_i))$. So bekommen wir zunächst einen Term $\delta_a(...\delta_{a_0}(\delta_{a_1}(), ..., \delta_{a_i}())...)$. Jeder der Terme $\delta_{a_1}()$ liefert uns ein Paar (t, q), mit $t \in term(\Sigma)$, $q \in Q$. Wir kriegen also jetzt einen Term $\delta_a(...\delta_{a_0}((t1, q_1), ..., (t_i, q_i))...)$ Jetzt passiert folgendes: für $(t[x_1, ..., x_i], q) \in \delta_{a_0}(q_1, ..., q_i)$ wird

(38) $\delta_{a_0}((t1, q_1), ..., (t_i, q_i)) = (t[t_1/x_1, ..., t_i/x_i], q)$

Wir bauen also nun – nachdem wir den alten Baum komplett in eine Funktion umgewandelt haben – wieder einen neuen Baum auf. Das ganze geht dann von unten nach oben weiter, so dass schlussendlich $\delta^*(t) = t' \in term(\Sigma)$. Im Prinzip gibt es hier dieselben Parameter zur Komplexität wie bei den top-down Automaten, so dass wir nicht mehr darüber reden müssen. Ein wichtiges Ergebnis ist aber folgendes: die Äquivalenz von top-down und bottom up, wie wir sie bei Automaten gesehen haben, bricht für Transduktoren zusammen; und es gibt jeweils eine Vielzahl von Transduktorenmodellen, die alle nichtäquivalente Relationen charakterisieren.

References

[1] Manfred Droste, Werner Kuich, and Heiko Vogler. *Handbook of Weighted Automata.* Springer, ?, 2009.

[2] Andreas Maletti, Jonathan Graehl, Mark Hopkins, and Kevin Knight. The power of extended top-down tree transducers. *SIAM J. Comput.*, 39:410–430.