

Bachelorarbeit

Konstruktion und Simulation von endlichen Automaten in Java mit besonderer Berücksichtigung universeller Abstandsautomaten

Konstantin Birker

04. Oktober 2012



Institut für Theoretische Informatik

betreut durch:

Dr. Jürgen Koslowski

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Braunschweig, 04. Oktober 2012

Zusammenfassung

Das Ziel dieser Arbeit war es ein didaktisches Werkzeug zu entwickeln, das in erster Linie Studenten hilft, die Funktionsweise von endlichen Automaten zu verstehen und spielerisch zu erlernen. Wert wird also auf eine gute visuelle Darstellung und ein komfortabel zu bedienendes Interface gelegt. Es geht nicht darum, ein Programm zu entwickeln, das es ermöglicht, Automaten schnell und effektiv zu simulieren.

Wichtige Aspekte sind dabei eine graphische und tabellarische Darstellung der Automaten, eine komfortable Möglichkeit Automaten im Programm zu konstruieren und dann Eingaben für den Automaten schrittweise zu simulieren. Auch eine Simulation der Konstruktion, also ein schrittweiser Aufbau eines vorgefertigten Automaten gehört dazu.

Der theoretische Teil der Arbeit beschäftigt sich mit universellen Abstandsautomaten. Diese sollen in das entwickelte Programm integriert werden und mit dessen Funktionen konstruiert, simuliert und erkundet werden. Derartige Abstandsautomaten stellen besondere Anforderungen an die Simulation. Benötigt werden Aspekte wie Nichtdeterminismus, spontane Übergänge, „Joker-Übergänge“, die für jeden Buchstaben benutzt werden können, und die Möglichkeit Übergangslabels einer Länge größer als eins zuzulassen.

Weiterhin sind einige Algorithmen für Automaten implementiert wie Epsilon-Elimination und Potenzmengenkonstruktion.

Inhaltsverzeichnis

Verzeichnis der Tabellen	ix
Verzeichnis der Abbildungen	x
Verzeichnis der Abkürzungen	xi
1 Features	1
1.1 Allgemeines	1
1.2 Automaten	2
1.3 Objektinspektor	4
1.3.1 Graph bzw. Automat	6
1.3.2 Knoten	8
1.3.3 Kanten	10
1.4 Graphische Ansicht	13
1.5 Tabellarische Ansicht	16
1.6 Simulationspanel	17
1.6.1 Konstruktion	18
1.6.2 Eingabe	18
1.6.3 Berechnung	18
2 Implementierung	19
2.1 Übersicht	19
2.1.1 Funktionale Klassen	19
2.1.2 GUI-Klassen	21
2.2 parseString	22
2.3 Tabellen	24
2.4 Shapes	26
2.5 Defaultwerte	28
2.6 Graph/Fsm-Erweiterung	28
2.7 Simulation	29
2.8 Algorithmen	30
2.8.1 Automaten optimieren	30
2.8.2 Dualer Automat	31
2.8.3 Epsilon-Elimination	31
2.8.4 Potenzmengenkonstruktion	32
2.8.5 Automateneigenschaften	33

3	Abstandsautomaten	35
3.1	Spezielle Abstandsautomaten	35
3.1.1	Hamming-Abstand	35
3.1.2	Levenshtein-Abstand	35
3.1.3	Damerau-Levenshtein	36
3.2	Motivation universeller Automaten	37
3.3	Ergebnisse	37
3.3.1	Hamming-Abstand	37
3.3.2	Levenshtein-Abstand	38
3.3.3	Damerau-Levenshtein	40
3.4	Vergleich zu Mihov/Schulz	42
4	Future Work	46
	Literatur	47

Verzeichnis der Tabellen

2.2	Übersicht über Typen und dazugehörige Editoren	27
3.2	Beispiel: Eingabe Majah bei dem Muster Maya für den Abstand 2. . .	40

Verzeichnis der Abbildungen

1.1	Übersicht über das Programm	1
1.2	Beispiel für die Umsetzung von Hoch- und Tiefstellung	2
1.3	Beispiele Determinismus	4
1.4	Der Objektinspektor	5
1.5	Die Automatenansicht des Objektinspektors	6
1.6	Die Knotenansicht des Objektinspektors	9
1.7	Die Kantenansicht des Objektinspektors	14
1.8	Die graphische Ansicht	14
1.9	Tabellarische Ansichten zu dem obigen Graphen: a) $Q \times Q$, b) $Q \times S$	16
1.10	Das Simulationspanel - Bis auf das Feld oben rechts ist die Ansicht für Konstruktion und Berechnung genauso.	17
2.1	Klassendiagramm funktionale Klassen mit den wichtigsten Attributen und Methoden	20
2.2	Klassendiagramm GUI-Klassen ohne Attribute und Methoden	21
2.3	Beispiele Potenzmengenkonstruktion Any und Else-Übergänge	33
3.1	Ein Hamming-Automat mit dem Abstand 2 für das Muster Maya	36
3.2	Ein Levenshtein-Automat mit dem Abstand 2 für das Muster Maya	36
3.3	Ein universeller Hamming-Automat mit dem Abstand 2	37
3.4	Ein universeller Levenshtein-Automat mit dem Abstand 2	38
3.5	Symbolische Dreiecke - erreichbare Zustände nach n verarbeiteten Buchstaben	39
3.6	Die deterministische Version des Levenshtein-Automaten für den Abstand 1	41
3.7	Die deterministische Version des Levenshtein-Automaten für den Abstand 1 mit geforderter 0 bzw. die minimierte Version.	41
3.8	Der Versuch eines universellen Damerau-Levenshtein-Automaten für den Abstand 3. Die rechteckigen Zustände sind die zusätzlichen Zustände zum Levenshtein-Automaten. Die farbigen Zustände sind die Zustände aus dem Levenshtein-Automaten (rot Löschen, blau Einfügen, grün Substitution). Die schwarzen Übergänge sind die zusätzlich benötigten Übergänge. Der waagerechte Übergang durch den zusätzlichen Zustand ist ein normaler Tausch, der steigende ein zusätzliches Einfügen, der fallende ein zusätzliches Löschen.	43
3.9	Deterministischer universeller Levenshtein-Automaten für den Abstand 1 nach Mihov/Schulz aus [1]	45

Verzeichnis der Abkürzungen

FSM	Finite state machine (endlicher Automat)
NEA	Nichtdeterministischer endlicher Automat
DEA	Deterministischer endlicher Automat
HCZ	Hotel California Zustand
GUI	Graphical User Interface (graphische Benutzeroberfläche)

1 Features

Dieses Kapitel beschreibt die Bedienung und die Möglichkeiten des Programms.

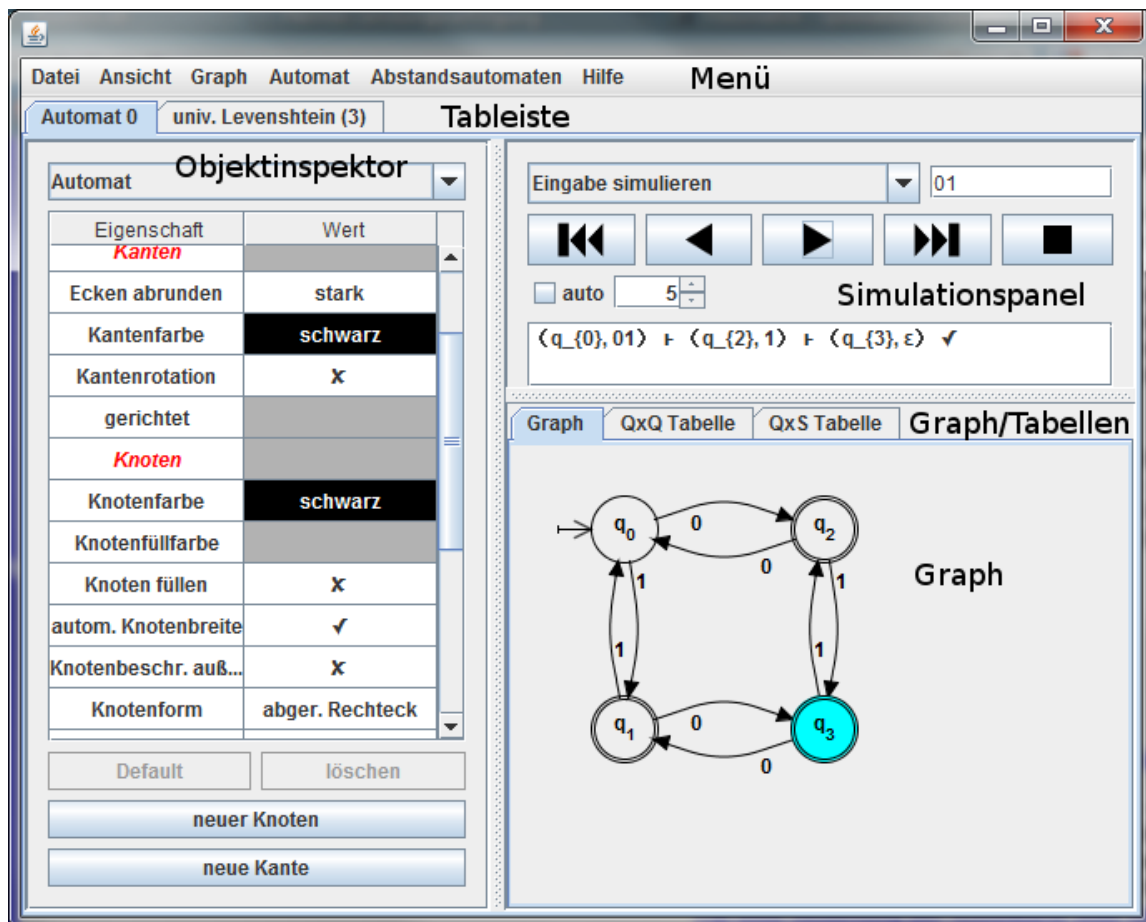


Abbildung 1.1: Übersicht über das Programm

1.1 Allgemeines

Das Programm dient in erster Linie dazu, Automaten zu konstruieren und simulieren. Da bei der Automatenkonstruktion und -darstellung eine Konstruktion und Darstellung für allgemeine Graphen enthalten ist, gibt es auch die Möglichkeit, Graphen

ohne Automateneigenschaften zu erstellen. Diese können nicht simuliert werden, haben keine $Q \times S$ -Tabellendarstellung und keine Übergangsfunktion. Dafür können für Kanten Gewichte eingestellt werden und es können ungerichtete Kanten benutzt werden.

Es können beliebig viele Graphen und Automaten geöffnet werden, welche dann in der Tableiste auswählbar sind. Über das *Datei*-Menü können neue Graphen oder Automaten erstellt werden.

Automaten und Graphen können gespeichert und geladen werden. Außerdem wird eine Liste zuletzt benutzter Dateien angezeigt, mit deren Hilfe Dateien schneller geladen werden können.

Bilder von Graphen können im PNG-Format exportiert werden. Dabei wird nicht der sichtbare Bereich, sondern der vollständige Graph ohne Ränder exportiert. Der Hintergrund ist transparent.

Im Menü *Ansicht* kann das Java-Look'n'Feel geändert werden.

Das Menü *Hilfe* beinhaltet eine Übersicht zur Steuerung des Graphen und Informationen zum Programm.

Namen und Kommentare können tief- und hochgestellte Teile enthalten. Dabei wird die in \LaTeX übliche Formatierung benutzt, also $_$ zum Tiefstellen und $^$ zum Hochstellen. Geschweifte Klammern dienen zur Blockbildung, ohne Block gilt das Formatierungszeichen nur für das nachfolgende Zeichen. Mit Backslashes kann die Funktion des Formatierungszeichens deaktiviert werden (Escapen). Um ein Backslash vor einem solchen Zeichen anzuzeigen, sind zwei Backslashes nötig.

$$\begin{array}{c} q_{\{01\}} q^{\{ab\}} q_{ab} q^{01} q_{\backslash 1} q_{\backslash\backslash}^1 \\ \quad \quad \quad \vee \\ q_{01} q^{ab} q_a b q^0 1 q_{\backslash 1} q_{\backslash}^1 \end{array}$$

Abbildung 1.2: Beispiel für die Umsetzung von Hoch- und Tiefstellung

1.2 Automaten

Die Automaten unterliegen keinen besonderen Einschränkungen. Ein Alphabet wird nicht vorgegeben, es ergibt sich automatisch aus den benutzten Übergangslabeln. Es dürfen beliebig viele Start- und Endzustände definiert werden. Es ist ebenfalls möglich, mehrere Kanten mit gleichen Start- und Zielknoten zu machen, was aber gleichwertig zu einer Kante mit mehreren Übergangslabeln ist. Kanten können optional mit einem Namen versehen werden. Knoten werden automatisch durchnummeriert (q_n bzw. v_n), dieser Name kann aber nach Belieben geändert werden. Mehrere Knoten mit gleichem Namen sind möglich, aber nicht empfehlenswert.

Die Simulation unterstützt Nichtdeterminismus, spontane Übergänge, Any-Übergänge (alle Symbole treten als Übergang auf), Else-Übergänge (alle Symbole, außer denen, für die ein expliziter Übergang definiert ist, treten als Übergang auf), Übergangslabel variabler Länge (Blöcke) und Symbolabkürzungen, wobei einzelne Zeichen gegen verschiedene andere Zeichen ersetzt werden können. Für Details bezüglich der Sonderzeichen siehe entsprechende Einträge bei den Automateneigenschaften im Abschnitt über den Objektinspektor.

Die Sonderfunktionen für die Simulation machen es sinnvoll, sich Gedanken über den Determinismusbegriff zu machen. Es ist klar, dass ein Automat mit mehreren Startzuständen nichtdeterministisch ist. Ein Automat wird ebenfalls nichtdeterministisch, wenn es von einem Zustand aus neben einem Any-Übergang noch Übergänge zu anderen Zuständen außer Else-Übergängen gibt. Else-Übergänge beeinflussen den Determinismus nur, wenn es von einem Zustand mehrere Else-Übergänge zu unterschiedlichen Zielzuständen gibt.

Spontane Übergänge bringen einen gewissen Nichtdeterminismus mit, weil ggf. in einem Schritt entschieden werden muss, ob ein normaler Übergang oder ein spontaner Übergang erfolgen soll. Man kann aber einen Determinismus definieren, dass es eindeutig sein muss, ob für eine erfolgreiche Weiterverarbeitung der Eingabe ein spontaner Übergang erfolgen muss oder nicht darf. Das bedeutet, es darf nur von maximal einem Epsilon-erreichbaren Zustand ein Übergang mit einem Symbol x möglich sein. Das erfordert einen Lookahead. Diese Definition von Determinismus soll hier gelten.

Lässt man nur Blöcke fester Länge n zu, braucht man für Determinismus keine besonderen Einschränkungen. Eine mögliche Interpretation wäre, den Block nicht als n konkatenierte Symbole aus dem Alphabet S , sondern als ein Symbol der Länge n aus dem Alphabet S^n zu betrachten. Problematisch wird es bei Blöcken variabler Länge. Wenn man als einzige Forderung stellt, dass die Übergangsrelation eine Funktion ist, wäre es möglich, dass von einem Zustand zwei Übergänge existieren, wobei der eine Übergang ein echtes Präfix des anderen ist und trotzdem jede Berechnung eindeutig ist. Diese Eindeutigkeit ist schwer feststellbar. Daher soll hier die restriktivere Variante gelten, dass in einem Zustand kein Übergang Präfix eines anderen sein darf (siehe auch Beispiele). Die spontanen Übergänge, die echtes Präfix eines jeden nicht spontanen Überganges sind, bilden hier eine Ausnahme.

Im Menü *Automat* finden sich einige Algorithmen. Unter *Elimination spontaner Übergänge* können spontane Übergänge von links, von rechts oder beidseitig durch Absorption von normalen Übergängen eliminiert werden.

Im Submenü *Minimierung* können unerreichbare Zustände, unproduktive Zustände und unnötige Kanten entfernt werden. *Unnötige Kanten entfernen* bedeutet, dass Mehrfachkanten mit gleichem Start- und Zielknoten zusammengefasst werden und Kanten ohne Übergänge entfernt werden. Minimierungsalgorithmen für DEAs oder gar NEAs sind noch nicht implementiert.

Die Potenzmengenkonstruktion macht aus einem NEA einen gleichwertigen DEA. Dies liefert einen neuen Automaten, der ursprüngliche Automat bleibt also erhalten.

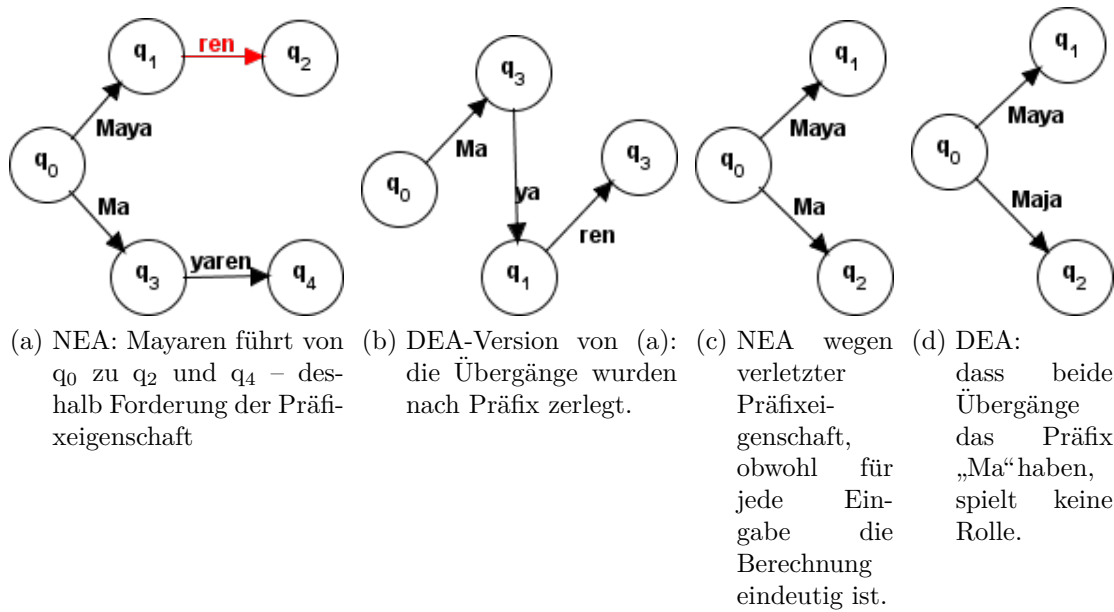


Abbildung 1.3: Beispiele Determinismus

Transformation in den dualen Automaten bedeutet das Austauschen von Start- und Zielzuständen und die Umkehrung aller Kanten.

Hotel California Zustand hinzufügen fügt dem Automaten einen Zustand hinzu, der nicht zur erkannten Sprache beiträgt, aber den Automaten vollständig macht; es führen also alle nicht definierten Übergänge in diesen Zustand.

Der Kleene-Algorithmus zur Identifikation der Sprache durch einen regulären Ausdruck ist noch nicht implementiert.

Automat untersuchen zeigt ein Fenster, in dem steht, ob der Automat vollständig ist oder nicht (also ob es von jedem Zustand mit jedem Symbol des Alphabetes einen Übergang gibt), ob es sich um einen NEA oder DEA handelt und ob der Automat spontane Übergänge benutzt. Außerdem werden alle Zustände aufgelistet und deren Anzahl angegeben. Es wird das minimale Alphabet bestimmt und Start- und Endzustandsmenge angezeigt.

Im Menü *Abstandsautomaten* sind Funktionen zur Konstruktion spezieller und universeller Abstandsautomaten zu finden. Außerdem kann hier das Tool zur Kodierung der Eingabe für universelle Abstandsautomaten aufgerufen werden. Zur Funktionsweise der Automaten siehe Kapitel 3 über Abstandsautomaten.

1.3 Objektinspektor

Der Objektinspektor ist die Leiste bzw. Tabelle am linken Fensterrand und dient dazu – je nach Auswahl – Eigenschaften vom Graphen bzw. Automaten, Knoten bzw.

Automat	
Eigenschaft	Wert
Kanten	
Ecken abrunden	stark
Kantenfarbe	schwarz
Kantenrotation	x
gerichtet	
Knoten	
Knotenfarbe	schwarz
Knotenfüllfarbe	
Knoten füllen	x
autom. Knotenbreite	✓
Knotenbeschr. auß...	x
Knotenform	abger. Rechteck
Default löschen	
neuer Knoten	
neue Kante	

Abbildung 1.4: Der Objektinspektor

Zustand oder Kante bzw. Übergang zu ändern. Über der Tabelle ist eine Combo-Box zur Auswahl des gewünschten Elementes. Unter der Tabelle sind Buttons zum Hinzufügen von Knoten und Kanten und zum Löschen und Zurücksetzen der Standardeinstellungen für Kanten und Knoten.

1.3.1 Graph bzw. Automat

Eigenschaft	Wert
Name	Automat 0
Kommentar	
<u>Default-Werte</u>	
<i>Kanten</i>	
Ecken abrunden	stark
Kantenfarbe	schwarz
Kantenrotation	✕
gerichtet	
<i>Knoten</i>	
Knotenfarbe	schwarz
Knotenfüllfarbe	
Knoten füllen	✕
autom. Knotenbreite	✓
Knotenbeschr. außen	✕
Knotenform	abger. Rechteck
Knotenbreite	40
Knotenhöhe	40
Knoten - Bogenbreite	40
Knoten - Bogenhöhe	40
<u>Automat</u>	
Any-Symbol	?
Else-Symbol	!
spontanes Symbol	#
Symbolabkürzungen	

Abbildung 1.5: Die Automatenansicht des Objektinspektors

Wert	Beschreibung	Beispiele
Name	Der Name hat keine besondere Bedeutung. Er wird beim Speichern vorgeschlagen und nach dem Laden als Bezeichnung des Tabs angezeigt.	Automat n, Graph n

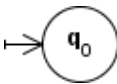

Wert	Beschreibung	Beispiele
Kommentar	Im Kommentar kann z.B. die Funktion des Graphen bzw. Automaten beschrieben werden. Diese wird dann, wenn der Graph ausgewählt ist, oben in der graphischen Ansicht angezeigt.	
Default-Werte	Für die Beschreibung der Default-Werte für Knoten und Kanten, siehe entsprechende Werte in den Unterkapiteln.	

Automaten haben noch einige zusätzliche Eigenschaften.

Wert	Beschreibung	Beispiele
Any-Symbol	Das Any-Symbol steht für einen beliebigen Übergang. Dieser Übergang kann für jedes beliebige Zeichen der Eingabe benutzt werden. Zu beachten ist, dass es nur für ein einzelnes Symbol steht und nicht in Blöcken benutzt werden kann.	?
Else-Symbol	Das Else-Symbol kann genau dann als Übergang benutzt werden, wenn es für das aktuelle Zeichen der Eingabe keinen anderen möglichen Übergang gibt. Es ist ähnlich wie das Any-Symbol, jedoch bleibt Determinismus erhalten. Zu Beachten: Spontane Übergänge beeinflussen Else-Übergänge nicht. Gibt es mehrere Else-Übergänge sind alle gleichwertig. Gibt es für jedes Zeichen einen anderen Übergang (z.B. durch ein Any-Übergang), ist der Else-Übergang leer.	!
Spontanes Symbol	Das spontane Symbol (typischerweise ein ϵ - aufgrund der einfacheren Eingabe per Default ein #) ist ein Übergang, der benutzt werden kann ohne ein Zeichen der Eingabe zu verbrauchen. Es können mehrere spontane Übergänge hintereinander ausgeführt werden.	#

Wert	Beschreibung	Beispiele
Symbol- abkürzungen	Symbolabkürzungen sind eine Menge von Abbildungen eines einzelnen Zeichens auf eine Menge von Zeichen. Hier kann z.B. definiert werden, dass ein „-“ für eine 0 oder eine 1 stehen kann. Gibt es bei der Simulation einen Übergang - kann dieser bei einer 0 oder einer 1 in der Eingabe benutzt werden. Symbolabkürzungen können auch in Blöcken benutzt werden. (Aber es können nicht einzelne Symbole auf Symbolblöcke abgebildet werden.)	





1.3.2 Knoten

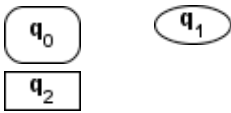
Wert	Beschreibung	Beispiele
Name	Der Name ist die Bezeichnung und Beschriftung des Knotens. Außerdem werden Knoten in Listen durch diesen Namen angezeigt.	q_n für Automaten, v_n für Graphen
Kommentar	Bei Auswahl des Knotens, wird der Kommentar oben in der grafischen Ansicht angezeigt. Dieser kann z.B. eine Beschreibung der Bedeutung des Knotens enthalten.	
Index	Der Index ist wichtig bei der Simulation der Konstruktion. Dabei wird schrittweise der Index erhöht und jeweils nur Knoten und Kanten angezeigt, deren Index kleiner oder gleich dem aktuellen Index ist.	0
Startzustand	Ist dieser Zustand Startzustand? Dies sind Zustände, in denen die Simulation einer Eingabe beginnen kann. Grafische Darstellung: ein eingehender Pfeil am linken Rand. In der Tabelle in der Zeile/Spalte I (für Initial) aufgeführt.	
Endzustand	Ist dieser Zustand Endzustand? Endet die Simulation einer Eingabe in einem Endzustand, wird die Eingabe akzeptiert. Grafische Darstellung: Zustand ist doppelt umrandet. In der Tabelle in der Zeile/Spalte F (für Final) aufgeführt.	

Eigenschaft	Wert
Name	q_{0}
Kommentar	
Index	0
Startzustand	✓
Endzustand	x
grafische Eigenschaft	
Farbe	schwarz
Füllfarbe	
füllen	x
autom. Breite	✓
Beschriftung außen	x
Beschriftungspositi...	
Form	abger. Rechteck
Position	48, 30
Breite	40
Höhe	40
Bogenbreite	40
Bogenhöhe	40

Abbildung 1.6: Die Knotenansicht des Objektinspektors

Grafische Eigenschaften

Wert	Beschreibung	Beispiele
Farbe	Die Farbe der Umrandung und Beschriftung.	
Füllfarbe	Die Farbe, in der der Knoten gefüllt wird. Nur auswählbar, wenn Füllen aktiviert ist.	
Füllen	Wenn nicht aktiviert, ist der Knoten nicht gefüllt, sonst mit der unter Füllfarbe festgelegten Farbe.	
Automatische Breite	Wenn aktiviert, wird der Knoten automatisch verbreitert, wenn die Beschriftung breiter als der Knoten ist.	 

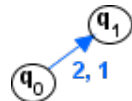
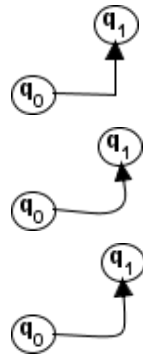
Wert	Beschreibung	Beispiele
Beschriftung außen	Soll die Beschriftung des Knotens frei außerhalb des Knotens platzierbar sein. Wenn nicht aktiviert, ist die Beschriftung immer an der gleichen Stelle und genauso groß wie der Knoten.	q_0
Beschriftungsposition	Die Position der Beschriftung in Pixeln. Nur auswählbar, wenn <i>Beschriftung außen</i> aktiviert ist.	
Form	Form des Knotens: abgerundetes Rechteck, Ellipse oder Rechteck. Bemerkung: Kreis und Quadrat sind Spezialfälle von Ellipse und Rechteck (oder auch vom abgerundeten Rechteck).	
Position	Die Position des Knotens in Pixeln.	
Breite	Breite des Knotens in Pixeln.	40
Höhe	Höhe des Knotens in Pixeln.	40
Bogenbreite	Nur für abgerundetes Rechteck: Wie viele Pixel in x-Richtung werden abgerundet.	40
Bogenhöhe	Nur für abgerundetes Rechteck: Wie viele Pixel in y-Richtung werden abgerundet.	40

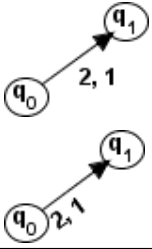
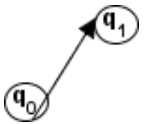
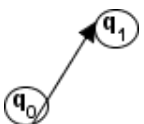
1.3.3 Kanten

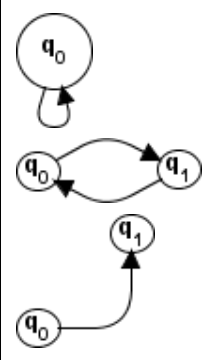
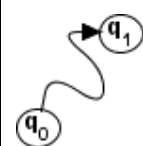
Wert	Beschreibung	Beispiele
Name	Der Name wird der Kantenbeschriftung vorangestellt.	
Kommentar	Bei Auswahl des Knotens, wird der Kommentar oben in der grafischen Ansicht angezeigt. Dieser kann z.B. eine Beschreibung der Bedeutung der Kante enthalten.	
Index	Der Index ist wichtig bei der Simulation der Konstruktion. Dabei wird schrittweise der Index erhöht und jeweils nur Knoten und Kanten angezeigt, deren Index kleiner oder gleich dem aktuellen Index ist.	0
Start	Der Zustand, von dem die Kante losgeht.	
Ziel	Der Zustand, zu dem die Kante hingehet.	

Wert	Beschreibung	Beispiele
Gewicht	In der Graphentheorie haben gewichtete Kanten Bedeutung. Für Graphen wird das Kantengewicht (sofern nicht NaN) neben dem Namen der Kante als Beschriftung angezeigt. Für Automaten hat das Kantengewicht keine Bedeutung und wird nicht angezeigt.	none
gerichtet	Nur einstellbar für Graphen. Ist die Kante gerichtet (mit Pfeil) oder nicht. Kanten von Automaten sind immer gerichtet.	true
Übergänge	Nur für Automaten. Mit welchen Symbolen bzw. Symbolblöcken kann diese Kante als Übergang in den Zielzustand benutzt werden.	

Grafische Eigenschaften

Wert	Beschreibung	Beispiele
Farbe	Die Farbe der Kante und der Beschriftung.	
Ecken abrunden	Durch Hilfspunkte können Ecken/Kurven in Kanten eingebaut werden. Ob diese Ecken abgerundet werden oder nicht kann hier ausgewählt werden. Für nähere Informationen siehe entsprechendes Kapitel in der Implementierung.	
Beschriftungsposition	Die Position in Pixeln, an der die Beschriftung ist. Bei Neuberechnung der Kante wird die Position neu berechnet (und muss ggf. erneut angepasst werden).	

Wert	Beschreibung	Beispiele
Beschriftungs-rotation	Wenn aktiviert, wird die Beschriftung am Beginn der Kante positioniert und in dem ausgehenden Winkel der Kante gedreht. Wenn deaktiviert, wird die Beschriftung horizontal in der Mitte zwischen Start und erstem Hilfspunkt (bzw. Ziel) positioniert.	
Beschriftungswinkel	Bei Neuberechnung der Kante wird der Winkel neu berechnet (und muss ggf. erneut angepasst werden).	
ausgehender Winkel	Sofern nicht anders festgelegt, wird der ausgehende Winkel automatisch berechnet. Hiermit kann festgelegt werden, an welcher Stelle des Knotens die Kante verankert ist. Hinweis: Die Kante tritt nicht orthogonal aus dem Knoten, dies kann durch zusätzliche Hilfspunkte erreicht werden (der ausgehende Winkel allerdings auch). 0° ist am rechten Rand und vertikal in der Mitte des Knotens.	
eingehender Winkel	Sofern nicht anders festgelegt, wird der eingehende Winkel automatisch berechnet. Hiermit kann festgelegt werden, an welcher Stelle des Knotens die Kante verankert ist. Hinweis: Die Kante tritt nicht orthogonal in den Knoten, dies kann durch zusätzliche Hilfspunkte erreicht werden (der ausgehende Winkel allerdings auch). 0° ist am rechten Rand und vertikal in der Mitte des Knotens.	

Wert	Beschreibung	Beispiele
automatische Hilfspunkte	Wenn aktiviert, werden für Schleifen und direkten Hin- und Rückkanten automatisch Hilfspunkte angelegt, sodass eine Schleife als solche erkennbar ist und die Kanten nicht übereinander liegen. Wenn der Menüpunkt rechtwinklige Kanten im Menü Graph aktiviert ist, werden Hilfspunkte derart angelegt, dass Kanten nur vertikal und horizontal verlaufen. Die Hilfspunkte werden bei jeder Neuberechnung der Kante neu angelegt und ggf. vorhandene Hilfspunkte werden gelöscht. Werden die Hilfspunkte verändert, wird dies automatisch deaktiviert.	
Hilfspunkte	Eine Liste von Punkten, an denen die Kante entlanggeleitet wird.	

1.4 Graphische Ansicht

Die graphische Ansicht ist die Hauptansicht für Graphen und Automaten. Automaten werden hier mit Knoten für Zustände und Kanten für Übergänge dargestellt.

Über den Menüpunkt *Graph am Raster ausrichten* im Menü *Graph* können die Knoten so positioniert werden, dass nur an diskreten Stellen Knoten sind. Dabei wird auch beachtet, dass keine Knoten übereinander liegen. Dabei werden die Knoten der Reihe nach an den nächst besten freien Platz positioniert und in keiner Weise intelligent angeordnet.

Der Menüpunkt *Spring Embedder* ist experimentell. Hierbei sollen die Knoten sinnvoll angeordnet werden, sodass verbundene Knoten beieinander liegen und zwischen Knoten ein gewisser Abstand ist. Das funktioniert bisher nur teilweise.

Mit dem Menüpunkt *rechtwinklige Kanten* lässt sich einstellen, ob automatisch Hilfspunkte angelegt werden sollen, damit Kanten nur horizontal und vertikal verlaufen können.

Alle wichtigen Konstruktionsaufgaben und Einstellungsmöglichkeiten können mit Hilfe von Tastatur- und Mauseingaben direkt in dem Graphen vollzogen werden. Die folgende Übersicht zur Steuerung kann auch über den Punkt *Bedienung Graph* im Menü *Hilfe* angezeigt werden. Grundgedanke der Steuerung ist Linksklick zum

Eigenschaft	Wert
Name	
Kommentar	
Index	0
Start	q_0
Ziel	q_2
Gewicht	0.0
Übergänge	0
grafische Eigenschaft	
Farbe	schwarz
Ecken abrunden	stark
Beschriftungspositi...	108, 38
Beschriftungsrotati...	\times
Beschriftungswinkel	0.0
ausgehender Winkel	0.0
eingehender Winkel	0.0
automatische Hilfsp...	✓
Hilfspunkte	128, 30

Abbildung 1.7: Die Kantenansicht des Objektinspektors

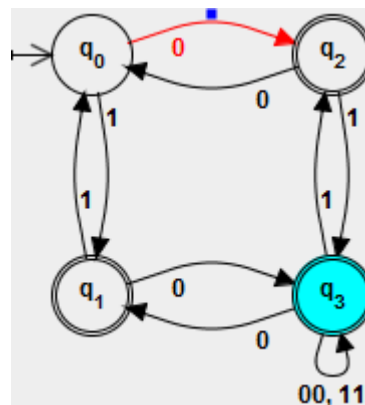


Abbildung 1.8: Die graphische Ansicht

Ziehen, Rechtsklick/Doppelklick zum Erstellen, Links Drag zum Verschieben und Mittelklick für andere Aufgaben.

Hinweis: Drag bedeutet klicken, nicht los lassen und ziehen

- Zustände
 - auswählen: Linksklick
 - neu: Rechtsklick oder Doppelklick mit links ins Leere
 - löschen: Auswählen, Entf drücken
 - verschieben: Links Drag
 - Start-/Endzustand durchschalten: Mittelklick
 - beschriften: Auswählen, Tastatureingabe (Backspace/Rücktaste zum Löschen)
- Kanten
 - auswählen: Linksklick
 - neu: Startknoten auswählen, Rechtsklick oder Doppelklick mit links auf den Zielknoten
 - löschen: Auswählen, Entf drücken
 - Ziel ändern: Auswählen, Rechtsklick oder Doppelklick mit links auf neuen Zielknoten
 - Übergang/Beschriftung ändern: Auswählen, Tastatureingabe (Backspace/Rücktaste zum Löschen)
 - Beschriftung verschieben: Links Drag auf Beschriftung
 - Beschriftungsrotation (de)aktivieren: Mittelklick auf Beschriftung
 - Hilfspunkte
 - * neu: Links Drag an gewünschter Stelle
 - * löschen: Kante auswählen, Mittelklick auf Hilfspunkt
 - * verschieben: Links Drag auf Hilfspunkt
 - * Abrundung der Ecken: Mittelklick auf die Kante (gar nicht, leicht, stark)
- sonstiges
 - neue Kante mit Knoten: Rechts Drag vom Startknoten zum Zielknoten; existiert am Ziel keiner, wird er erstellt.
 - nichts auswählen: Linksklick ins Leere

Graph	QxQ Tabelle		QxS Tabelle			
	q_0	q_1	q_2	q_3	I	F
q_0	-	1	0	-	✓	✗
q_1	1	-	-	0	✗	✓
q_2	0	-	-	1	✗	✓
q_3	-	0	1	00, 11	✗	✓
I	✓	✗	✗	✗		
F	✗	✓	✓	✓		

(a) $Q \times Q$

Graph	QxQ Tabelle		QxS Tabelle			
	1	0	00	11	I	F
q_0	q_1	q_2			✓	✗
q_1	q_0	q_3			✗	✓
q_2	q_3	q_0			✗	✓
q_3	q_2	q_1	q_3	q_3	✗	✓

(b) $Q \times S$ Abbildung 1.9: Tabellarische Ansichten zu dem obigen Graphen: a) $Q \times Q$, b) $Q \times S$

- alles verschieben: Rechts Drag im Leeren

1.5 Tabellarische Ansicht

Die Übergangsrelation von Automaten lässt sich als Teilmenge von $Q \times Q \times S$ auffassen. Da das kartesische Produkt kommutativ ist, lässt dies mehrere Darstellungen von Tabellen zu. Für Automaten sind, $Q \times Q$, wobei in den Zellen Symbole aus S stehen und $Q \times S$, wobei in den Zellen die Zielzustände aus Q stehen. Für Graphen gibt es nur die $Q \times Q$ -Darstellung, wobei der Inhalt der Zellen zeigt, ob eine Kante von dem Startzustand (Zeile) zum entsprechenden Zielzustand (Spalte) existiert.

Bei $Q \times Q$ sind in Zeilen und Spalten die Zustände bzw. Knoten angeordnet. An den Kreuzungspunkten stehen die Kante mit den Übergängen, sofern eine existiert. Zwei zusätzliche Zeilen und Spalten zeigen an, ob der Zustand Startzustand (I) oder Endzustand (F) ist.

Durch Klick auf einen Zustand kann der Name des Zustandes geändert werden. Durch Klicks auf Startzustand- bzw. Endzustand-Felder kann der entsprechende Status gewechselt werden. Die Kanten werden textuell bearbeitet, wobei die einzelnen Übergänge durch „ „ getrennt werden müssen. Ein Name kann vorangestellt werden, abgetrennt durch „:“. Zustände bzw. Knoten sollten durch den Button unter dem Objektinspektor hinzugefügt werden.

Bei $Q \times S$ sind in Zeilen die Zustände angeordnet und in den Spalten alle vorkommenden Übergänge. An den Kreuzungspunkten steht eine Liste von Zuständen, die vom entsprechenden Zustand mit dem entsprechenden Übergang erreichbar sind. Zwei zusätzliche Spalten zeigen an, ob der Zustand Startzustand (I) oder Endzustand (F) ist.

Durch Klick auf einen Zustand, kann der Name des Zustandes geändert werden. Durch Klicks auf Startzustand- bzw. Endzustand-Felder kann der entsprechende Status gewechselt werden. Die erreichbaren Zustände werden durch den CollectionEditor bearbeitet, in dem Zustände hinzugefügt, gelöscht oder ausgetauscht werden können. Übergänge können mit dem Eingabefeld und den Buttons unter der Tabelle hinzugefügt oder entfernt werden. Zustände bzw. Knoten sollten durch den Button unter dem Objektinspektor hinzugefügt werden.

Für die Tabellen sollte noch eine Darstellung der Simulation folgen und eine Änderung der Auswahl.

1.6 Simulationspanel

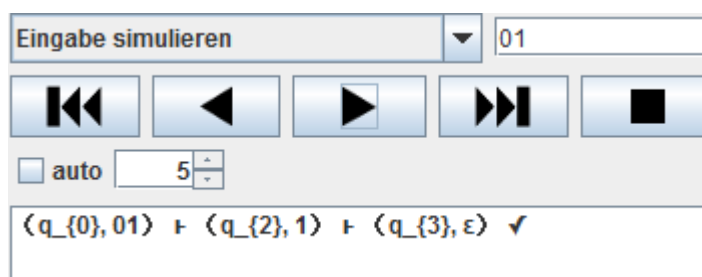


Abbildung 1.10: Das Simulationspanel - Bis auf das Feld oben rechts ist die Ansicht für Konstruktion und Berechnung genauso.

Das Simulationspanel gibt es nur für Automaten.

Hinweis: Die Simulation funktioniert bisher nur mit der graphischen Ansicht.

Das Simulationspanel kann genutzt werden um die Konstruktion, eine Eingabe oder eine spezielle Berechnung zu simulieren. Die Aufgabe kann mit der ComboBox oben links ausgewählt werden.

Mit kann die Simulation gestartet bzw. zurück auf den Anfang gesetzt werden. Mit und kann schrittweise vor und zurück gegangen werden. beendet die Simulation, geht also zum letzten Schritt. bricht die Simulation ab.

ist so eingerichtet, dass es bei Bedarf fragt, ob die Simulation gestartet werden soll, wenn die Simulation noch nicht gestartet wurde. Wenn die CheckBox *auto* aktiviert ist, folgt der nächste Schritt automatisch nach der im Spinner festgelegten Anzahl von Sekunden.

1.6.1 Konstruktion

Bei der Simulation der Konstruktion wird schrittweise ein Zähler erhöht. Der aktuelle Zählerstand kann oben rechts gesehen und verändert werden. In jedem Simulationsschritt werden nur Elemente angezeigt, deren Index kleiner oder gleich dem aktuellen Zählerstand ist.

1.6.2 Eingabe

Die zu simulierende Eingabe kann im Textfeld oben rechts eingegeben werden. Für jeden Simulationsschritt wird die zuletzt benutzte Kante und die aktiven Zustände markiert. Parallel werden alle möglichen Berechnungen zu der Eingabe in die Liste eingetragen. Zu Beachten ist, dass durch unterschiedliche Längen der Übergangslabel (inklusive spontaner Übergang, der kein Zeichen verbraucht), verschiedene Berechnungen unterschiedlich weit in der Verarbeitung der Eingabe sein können. Sind alle Berechnungen fertig berechnet, zeigt der nächste Schritt, ob der Automat die Eingabe akzeptiert oder nicht, also ob zu den aktiven Zuständen ein Endzustand gehört.

1.6.3 Berechnung

Hier kann eine bestimmte Berechnung einzeln simuliert werden. Dafür muss sie vorher mit der Eingabesimulation berechnet worden sein. Ist die Berechnung nicht vollständig, ist diese Simulation auch nicht vollständig. Nach der Berechnung kann die gewünschte Berechnung aus der Liste ausgewählt und wie die Eingabesimulation nachvollzogen werden. Dies kann nützlich sein, wenn durch Nichtdeterminismus sehr viele Konfigurationen parallel existieren. Oben rechts wird die verbleibende Eingabe angezeigt.

2 Implementierung

Dieses Kapitel gibt eine Übersicht über den grundsätzlichen Aufbau des Programms und beleuchtet dann einige Aspekte und Probleme näher.

2.1 Übersicht

Die Klassen des Projektes lassen sich in die funktionalen Klassen im Hintergrund und die GUI-Klassen zur Darstellung und Interaktion mit dem User unterteilen. Das Projekt nutzt keine externen Pakete. Die Oberfläche ist aus Swing-Komponenten gebaut. Das Projekt nutzt das Observer-Pattern. Nach Änderungen am Graphen o.ä. werden alle Observer benachrichtigt, welche alle darstellenden Komponenten sein sollten, die sich dann entsprechend aktualisieren können.

Ein grundsätzliches Problem bei der Implementierung ist, dass Java es nicht erlaubt, konstante Pointer benutzen, also ein Read-Only-Zugriff auf Objekte. Z.B. muss auf die Listen der Kanten und Knoten zum Zeichnen zugegriffen werden. Das bedeutet aber auch, dass durch diesen Zugriff die Objekte verändert werden könnten, was zu inkonsistenten Zuständen führen könnte (z.B. müssen die Kanten sowohl in die globale Liste, als auch die lokale Liste des Startzustandes eingetragen werden). Man könnte dies umgehen, indem man eine Wrapper-Klasse erstellt, die nur Getter-Methoden öffentlich zur Verfügung stellt. Das ist aber ein unverhältnismäßiger Aufwand.

2.1.1 Funktionale Klassen

Die Hauptklasse ist die Klasse Graph, welche einen grundlegenden Eigenschaften eines Graphen implementiert und die Knoten und Kanten hält.

Die Klasse Fsm erweitert die Klasse Graph um automaten-spezifische Aspekte, das beinhaltet den Austausch der Kanten gegen eine für Automaten erweiterte Version, Methoden zur Simulation von Eingaben und die Algorithmen auf Automaten.

Knoten werden durch die Klasse Vertex repräsentiert, Kanten durch die Klasse Edge, bzw. EdgeFsm, welche die Kanten von Graphen um Übergänge erweitert. Diese Klassen implementieren das leere Interface Element, damit z.B. für eine Auswahl beide Klassen gleichartig gespeichert werden können.

2 Implementierung

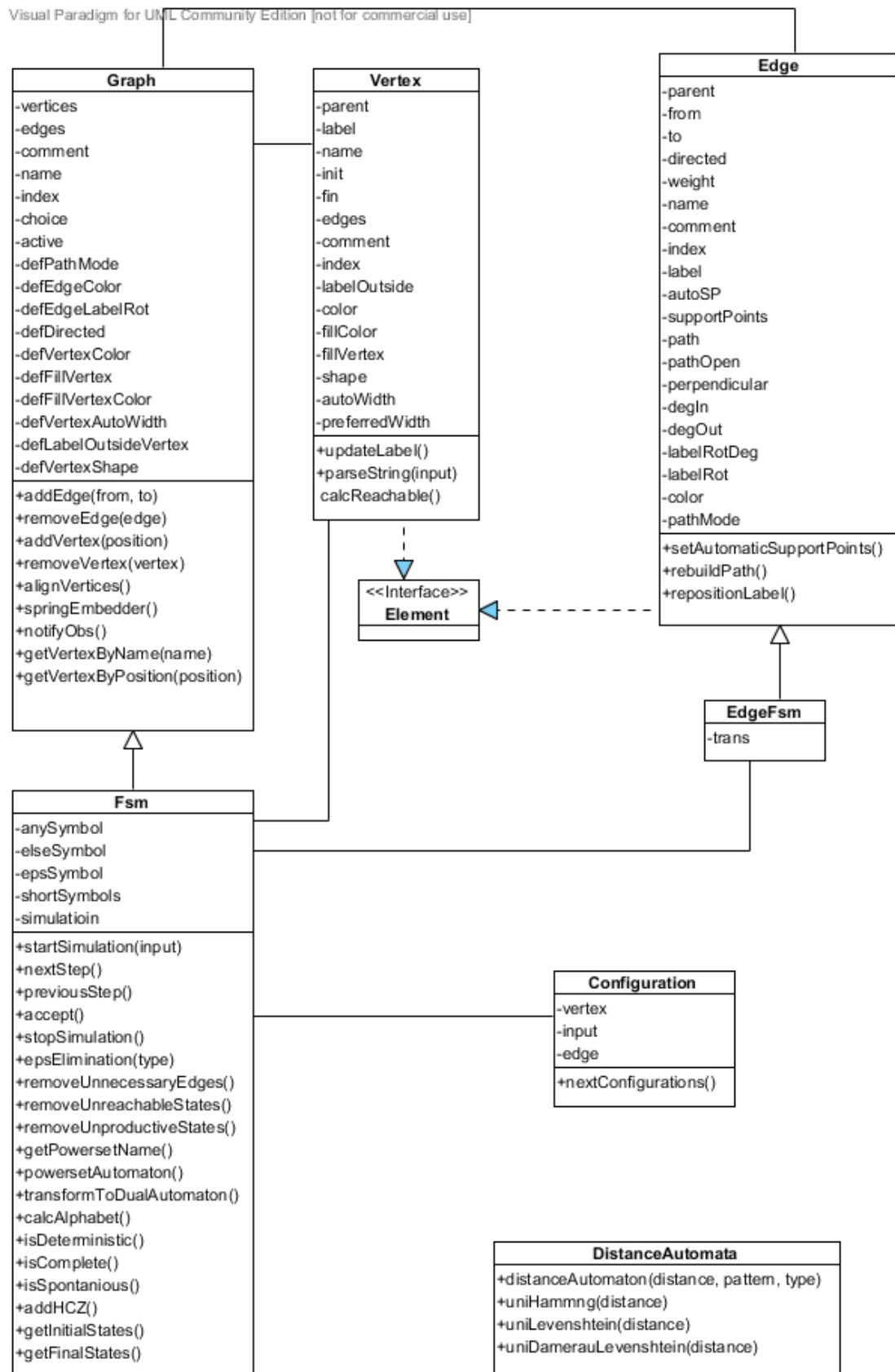


Abbildung 2.1: Klassendiagramm funktionale Klassen mit den wichtigsten Attributen und Methoden

Für die Simulation wird die Klasse Configuration benutzt. Sie repräsentiert einen Berechnungsschritt während der Simulation, also den aktuellen Zustand mit der Resteingabe. Außerdem enthält die Klasse eine Funktion um eine Menge von Folgekonfigurationen zu berechnen.

Die Klasse DistanceAutomata enthält ausschließlich statische Methoden zur Erzeugung verschiedener Abstandsautomaten (universell und speziell).

2.1.2 GUI-Klassen

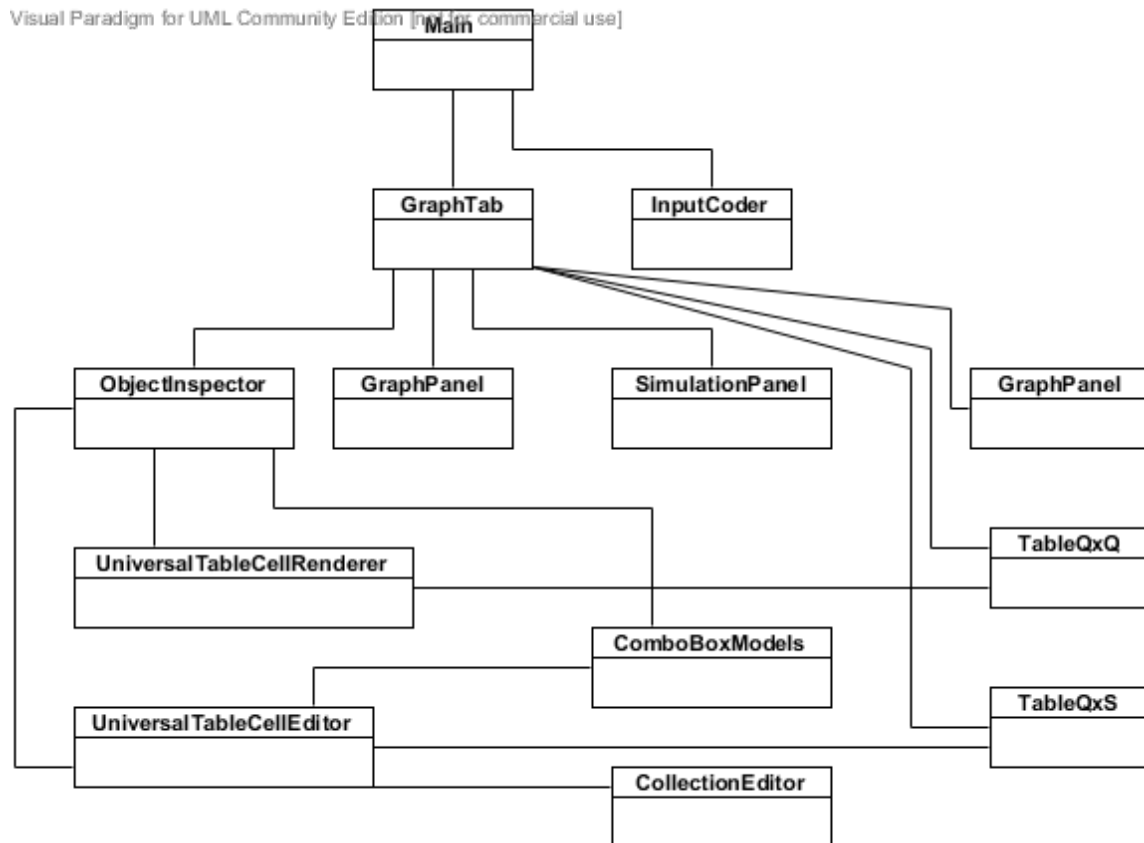


Abbildung 2.2: Klassendiagramm GUI-Klassen ohne Attribute und Methoden

Die Hauptklasse ist die Klasse Main. Sie ist ausführbar und erstellt ein JFrame mit einer Menüleiste und einer JTabbedPane. Hierin werden beliebig viele Automaten-/Grapheneinheiten repräsentiert durch die Klasse GraphTab. Hierin sind der Objektinspektor, die graphische Visualisierung, QxQ- und QxS-Tabelle und die Simulationssteuerung mit den entsprechenden Klassen ObjectInspector, GraphPanel, TableQxQ, TableQxS und SimulationPanel angeordnet. Alle Tabellen (also QxQ, QxS und Objektinspektor) benutzen als Renderer zur Darstellung den UniversalTableCellRenderer und als Editor den UniversalTableCellEditor. Die Modelle, die den Tabellen die Daten liefern, sind Unterklassen der entsprechenden Komponente.

Die Klasse `ComboBoxModels` enthält Modelle und Renderer für `ComboBoxen` (mit Knoten oder mit Elementen).

Die Klasse `CollectionEditor` wird von dem `UniversalCellEditor` benutzt und zeigt ein Fenster zur Bearbeitung von Listen, Sets und Maps.

`InputCoder` ist ein Fenster, welches dazu da ist, die Eingabe für universelle Abstandsautomaten zu kodieren.

2.2 parseString

Bei den Beschriftungen von Knoten und Kanten ist Hoch- und Tiefstellen von Text möglich. Dafür sind zwei Implementierungen denkbar. Eine Möglichkeit wären `Attributed Strings`. Im folgenden Beispiel sind die Zahlen 5-7 hochgestellt.

Listing 2.1: Java-Code

```
1 AttributedString as1 = new AttributedString("1234567890");
2 as1.addAttribute(TextAttribute.SUPERScript, TextAttribute.
   SUPERScript.SUPER, 5, 7);
3 g2d.drawString(as1.getIterator(), 15, 60);
```

Die Alternative beruht darauf, dass (fast) alle Swing-Komponenten HTML darstellen können. Das folgende Beispiel stellt ebenfalls die Zahlen 5-7 hoch.

Listing 2.2: Java-Code

```
1 label1.setText("<html>1234<sup>567</sup>890</html>");
```

Umgesetzt ist die zweite Methode. Beim setzen von Beschriftungen wird der String nach den formatierenden Symbolen (`_`, `^` und `\`) gesucht und dann in entsprechenden HTML-Code umgesetzt. Dabei ist die Reihenfolge der Auswertung wichtig.

Listing 2.3: Java-Code

```
1         public static String parseString(String s) {
2             StringBuilder t = new StringBuilder();
3             Stack<String> open = new Stack<String>();
4             boolean esc = false;
5             boolean ins = false;
6             boolean html = false;
7             for (int i = 0; i < s.length(); i++) {
8                 final char c = s.charAt(i);
9                 if (esc) {
10                     t.append(c);
11                     esc = false;
12                     if (ins) {
```

```

13         t.append(open.pop());
14         ins = false;
15     }
16 } else if (c == '\\') {
17     esc = true;
18 } else if (ins && c != '{') {
19     t.append(c).append(open.pop());
20     ins = false;
21 } else if (c == '_') {
22     t.append("<sub>");
23     open.add("</sub>");
24     ins = true;
25     html = true;
26 } else if (c == '^') {
27     t.append("<sup>");
28     open.add("</sup>");
29     ins = true;
30     html = true;
31 } else if (c == '{') {
32     if (!ins) {
33         t.append(c);
34     } else {
35         ins = false;
36     }
37 } else if (c == '}') {
38     if (open.isEmpty()) {
39         t.append(c);
40     } else {
41         t.append(open.pop());
42     }
43 } else {
44     t.append(c);
45 }
46 }
47 while (open.size() > 1) {
48     t.append(open.pop());
49 }
50 if (ins) {
51     if (open.pop().equals("</sub>")) {
52         t.append('_');
53     } else {
54         t.append('^');
55     }
56 } else if (!open.isEmpty()) {
57     t.append(open.pop());
58 } else if (esc) {

```

```

59         t.append('\\\\');
60     }
61     if (html) {
62         t.append("</nobr></html>");
63         t.insert(0, "<html><nobr>");
64     }
65     return t.toString();
66 }

```

2.3 Tabellen

Tabellen in Java sind etwas gewöhnungsbedürftig, aber dafür recht mächtig. Es gibt drei Teile, über die man sich beim Bau einer Tabelle Gedanken machen muss.

Das erste ist das `TableModel`. Die wichtigen Methoden in dieser Klasse sind `getValueAt(int rowIndex, int colIndex)` und für editierbare Tabellen `isCellEditable(int rowIndex, int colIndex)` und `setValueAt(Object value, int rowIndex, int columnIndex)`. Es gibt vorgefertigte einfache `TableModels`, die aber schnell an die Grenzen stoßen. Alle `TableModels` (Objektinspektor, $Q \times Q$ -Tabelle und $Q \times S$ -Tabelle) sind überschrieben und geben in Abhängigkeit der Position direkt Daten des Graphen bzw. Automaten weiter bzw. schreiben diese Werte. Für Werte bei denen eine Referenz übergeben wurde und nur dieses Objekt bearbeitet und nicht überschrieben wurde (z.B. Listen) muss die `setValueAt`-Methode gar nichts unternehmen.

Der `Renderer` ist für die Darstellung verantwortlich. Der `DefaultRenderer` ist nur geeignet, wenn eine gesamte Spalte einen eindeutigen Typ hat. Dann kann er z.B. Boolean-Werte durch eine `CheckBox` darstellen. Objects werden einfach durch ihre Stringrepräsentation dargestellt. Alle Tabellen benutzen den `UniversalTableCellRenderer`, welcher die Methode `public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column)` überschreibt. In Abhängigkeit des Types von `value` (Test mit `instanceof`) wird eine geeignete Darstellung in einem `JLabel` erstellt und zurückgegeben.

Der `Editor` übernimmt die Bearbeitung der Werte. Zum einen muss die Methode `public Component getTableCellEditorComponent(JTable table, Object value, boolean isSelected, int rowIndex, int colIndex)` überschrieben werden, welche – wie beim `Renderer` – abhängig vom Typ von `value` eine geeignete Komponente zur Bearbeitung des Wertes zurückliefert. Als zweites muss die Methode `public Object getCellEditorValue()`, welche nach dem Beenden des Editierens aufgerufen wird und den neuen Wert an das `TableModel` zurück gibt, überschrieben werden. Dafür wird beim Wählen der Komponente diese gespeichert, damit dann daraus der Wert (mit dem richtigen Typ) zurückgegeben werden kann.

Ein Schwierigkeit bei der Implementierung war, dass das Beenden des Editierens einer Zelle nur durch explizite Aufforderung durch den Benutzer geschieht, also durch Auswahl einer anderen Zelle mit der Maus oder Bestätigen durch Enter oder Tab. Zum einen werden dadurch die neuen Werte erst spät übernommen, zum anderen kann es zu Problemen kommen, wenn sich während des Editierens der Inhalt der Zelle ändert, oder gar der Typ (z.B. wenn beim Objektinspektor ein anderer Elementtyp (Graph, Knoten, Kante) ausgewählt wird). Es soll eine Möglichkeit geben mit `table.putClientProperty("terminateEditOnFocusLost", Boolean.TRUE)`; das Editieren automatisch zu stoppen, aber das hat nicht funktioniert. Es funktioniert ebenfalls nicht, das Editieren zu Stoppen, wenn die Tabelle den Focus verliert, da die Tabelle den Focus auch nicht hat, wenn der Editor in der Tabelle ihn hat. Also muss jeder Editor einen FocusListener bekommen, in dem das Editieren bei FocusLost explizit gestoppt werden kann. Dabei ist zu beachten, dass z.B. JSpinner keinen Focus haben, sondern nur das TextFeld des Editors des Spinners, außerdem sollte dem TextFeld Focus gegeben werden, wenn direkt ein Button des Spinners benutzt wird. Weiterhin ist darauf zu achten, nicht bei jedem Focus-Verlust das Editieren zu beenden, denn wenn eine andere Zelle ausgewählt wird, wird das Editieren erst gestoppt und der nächste Editor erhält den Focus danach, d.h. es würde das Editieren sofort noch einmal gestoppt werden. Ähnliches gilt, wenn der Editor aus mehreren Komponenten besteht (z.B. Punkte mit zwei Spinners). Die Probleme wurden gelöst, indem alle Editorkomponenten, die einen Focus erhalten können, in einer Liste gespeichert werden und beim FocusLost getestet wird, ob die neue Komponente mit dem Focus nicht in dieser Liste ist.

Weiterhin ist zu beachten, dass beim Stoppen des Editierens nicht automatisch die neuen Werte beim JSpinner gesetzt werden, es sollte also vor dem Abfragen der Werte ein `((JSpinner.DefaultEditor) spinner.getEditor()).commitEdit()` aufgerufen werden. Bei Werten die über ein eigenes Fenster (Collection-Editor, Color) und bei Boolean-Werten sollte das Editieren automatisch sofort nach dem Zurückgeben der Komponente gestoppt werden, damit die Änderungen übernommen werden können. Das funktioniert, indem unmittelbar nach der Auswahl der Komponente, der `stopEditing`-Befehl in die Java-EventQueue eingetragen wird:

Listing 2.4: Java-Code

```

1 java.awt.EventQueue.invokeLater(new Runnable() {
2     @Override
3     public void run() {
4         stopCellEditing();
5     }
6 });

```

Eine ComboBox sollte über einen ActionListener nach der Auswahl eines Wertes das Editieren beenden.

Außerdem wird für alle textuellen Editoren ein Timer gestartet, der nach 3 Sekunden

ohne Eingabe das Editieren automatisch beendet (durch Key- und MouseListener wird dieser Timer bei beliebiger Interaktion neu gestartet).

2.4 Shapes

Für die graphische Repräsentation der Knoten und Kanten werden Klassen, die das Interface Shape implementieren, benutzt. Knoten können beliebige RectangularShape annehmen; Arc2D, Ellipse2D, Rectangle2D und RoundRectangle2D werden von Java mitgeliefert. Bis auf Arc2D können diese Formen auch als Knotenform im Objektinspektor ausgewählt werden. Es wäre ohne weiteres möglich, eine Klasse zu schreiben, die RectangularShape implementiert und diese dann als Knotenform zu benutzen. Die Shape-Klassen lassen sich dann sehr leicht auf den graphischen Kontext eines Panels zeichnen. Mit der contains-Methode lässt sich überprüfen, ob z.B. ein Mausklick innerhalb eines Shapes gemacht wurde.

Kanten werden als Path2D repräsentiert. Dieser setzt sich aus Abschnitten von geraden Linien, quadratischen oder kubischen Bezierkurven zusammensetzen. Der Pfad wird vom Startknoten zum Zielknoten über die Hilfspunkte berechnet. Berechnung von Start und Zielpunkt erfordert etwas Rechnerei, weil es nicht ausreicht, vom Zentrum oder Rand der BoundingBox die Linie zu zeichnen, dadurch würde ein Teil der Linie innerhalb des Knotens liegen (was zum einen bei Transparenz ein Problem ist, zum anderen den Pfeil verschwinden lässt) oder die Linie in der Luft schwebt. Zunächst wird die Richtung bestimmt als Vektor zwischen Start bzw. Ziel und wenn vorhanden nächster Hilfspunkt oder Ziel bzw. Start berechnet. An diesem Vektor wird dann so lange „entlang gelaufen“, bis man gerade innerhalb des Knotens bzw. gerade außerhalb des Knotens ist (Überprüfung mit contains()).

Ohne Abrunden von Ecken wird einfach eine Linie von Hilfspunkt zu Hilfspunkt gezeichnet. Mit Abrundung werden Bezierkurven von dem Mittelpunkt der Verbindung letzter Hilfspunkt (bzw. Start) zum nächsten Hilfspunkt (bzw. Ziel) mit dem aktuellen Hilfspunkt als Stützpunkt gezeichnet. Bei starker Abrundung wird eine quadratische Bezierkurve benutzt. Bei leichter Abrundung wird eine kubische Bezierkurve mit dem aktuellen Hilfspunkt für beide Stützpunkte benutzt, wodurch der Hilfspunkt stärker gewichtet wird. Die fehlenden Enden werden durch einfache Linien ergänzt.

Die Auswahl der Pfade erfolgt mit der intersect-Methode und einem um den gewünschten Punkt gelegtem Kreis. Dabei ist es ärgerlich, dass Java den Pfad automatisch schließt – grundsätzlich durch eine direkte Verbindung von Start und Ziel. Das ist unproblematisch ohne Hilfspunkte, aber mit Hilfspunkten entstehen dadurch unerwünschte Formen mit einem echten Flächeninhalt. Um dies zu umgehen, muss der Pfad noch einmal komplett rückwärts berechnet werden. Es werden also zwei Pfade für eine Kante gehalten, einmal der einfache zum Zeichnen und ein weiterer

Typ	Werte	Renderer	Editor	Stop Editing	Listener
String	Name, Kommentar	String	TextField	Focus, Timer	Focus, Key, Mouse
Character	Automaten-Symbole	toString	TextField mit DocumentFilter	Focus, Timer	Focus, Key, Mouse
Color	Farben	Label-Farbe, String, RGB-Wert	JColor-Chooser	instantan	-
Boolean	Initial, Final, Füllen, autom. Breite usw.	Haken/Kreuz	JLabel	instantan	-
Integer	Index, Breite, Höhe	toString	JSpinner	Focus, Timer	(Focus, Key, Mouse,) Change
Double	Winkel, Gewicht	toString	JSpinner	Focus, Timer	(Focus, Key, Mouse,) Change
Point	Positionen	int, int	2 JSpinner	Focus, Timer	je (Focus, Key, Mouse,) Change
Shape	Form	enum (String)	JComboBox	Focus, Auswahl	Focus, Action
PathMode	Ecken abrunden	enum (String)	JComboBox	Focus, Auswahl	Focus, Action
Vertex	Start, Ziel	getName()	JComboBox	Focus, Auswahl	Focus, Action
List	Hilfspunkte, QxS-Eintrag	zeilenweise	Collection-Editor	instantan	-
Set	Übergänge	zeilenweise	Collection-Editor	instantan	-
Map	Symbol-abkürzungen	zeilenweise Key => Val	Collection-Editor	instantan	-

Tabelle 2.2: Übersicht über Typen und dazugehörige Editoren

geschlossener zur Überprüfung eines Schnittes. Die Punkte des Pfeiles müssen extra berechnet werden unter Berücksichtigung der entsprechenden Winkel.

Die Beschriftungen sind in JLabels gespeichert. Es hat sich herausgestellt, dass man es vermeiden sollte, die Labels dem Container hinzuzufügen, da dadurch durchgehend repaints aufgerufen wurden. Stattdessen sollte man die Labels mit der paint-Methode in den graphischen Kontext des Panels zeichnen. Das bedeutet, dass man keine Mouse-Listener auf den Labels verwenden kann, sondern in dem MouseListener des Panels über contains überprüft werden muss, ob der Mausklick im Label war. Gleichzeitig wird es dadurch sehr einfach, das Label zu drehen, indem man vor dem Zeichnen den graphischen Kontext dreht. Diese Drehung muss auch bei der contains-Überprüfung beachtet werden.

2.5 Defaultwerte

Zunächst waren die Default-Werte als statische Klassenattribute direkt in der Kante bzw. dem Knoten gespeichert. Das hat den großen Nachteil, dass bei mehreren geöffneten Automaten bzw. Graphen die Defaultwerte global für alle gelten. Außerdem werden dadurch die Defaultwerte nicht mitgespeichert.

Die Alternative war, die Defaultwerte im Graphen (Automaten) zu speichern und in den Knoten-/Kantenobjekten das Elternelement zu speichern, damit auf diese Werte zugegriffen werden kann.

Ein weiterer Aspekt war, dass zunächst ein boolean-Attribut inherit benutzt wurde, welches angegeben hat, ob die Kante bzw. der Knoten die Standardeinstellungen erben soll. In diesem Fall wurde direkt der Standardwert im Getter zurückgegeben. Bei der Änderung eines solchen Wertes wurde auf inherit auf false gesetzt. Diese Art des inherit-Flags hatte den Vorteil, dass, wenn der Standard-Wert geändert wurde, dieser sofort automatisch von den Knoten bzw. Kanten übernommen wurde. Auf der anderen Seite entstanden dadurch unschöne Effekte, dass sich mit dem Ändern des inherit-Flags mehr Eigenschaften änderten, als eigentlich der Fall sein sollte. Deshalb wurde das Flag entfernt und stattdessen gibt es eine Methode, um die Werte zurück auf die Defaultwerte zu setzen. Nun werden Änderungen der Defaultwerte selbst nicht rückwirkend übernommen und gelten nur für neue Kanten und Knoten. Es wäre aber denkbar, dafür eine Option einzubauen, die dafür sorgt, dass der Defaultwert auch bei bereits vorhandenen Elementen gesetzt wird.

2.6 Graph/Fsm-Erweiterung

Da die Automaten eine graphische Darstellung unterstützen sollen, können Automaten als Graphen mit zusätzlichen Eigenschaften, die speziell für Automaten benötigt

werden, betrachtet werden. Es bietet sich daher an, zunächst eine Klasse Graph zu erstellen, die alle Informationen für eine graphische Darstellung mit Knoten und Kanten enthält und diese dann um die zusätzlichen Funktionen zu erweitern. Das bietet die Möglichkeit, im Programm auch Graphen ohne Automatenereigenschaften zu konstruieren. Lediglich zwei Eigenschaften von Graphen haben für Automaten keine Bedeutung, nämlich Kantengewichte, welche aber nicht stören, und die Möglichkeit ungerichtete Kanten zu benutzen. Das wird im Automaten durch ein Überschreiben der Setter-Methode für diesen Wert erreicht.

Beim Erweitern des Graphen müssten auch die Klassen Edge und Vertex erweitert werden, da die wesentlichen Eigenschaften der Automaten in diesen Klassen liegen. Kanten benötigen zusätzlich die Übergangsrelation und Knoten die Eigenschaften Start- bzw. Finalzustand. (Für die Potenzmengenkonstruktion haben die Knoten auch noch Felder und Methoden zur Berechnung der erreichbaren Zustände). Die Klasse Edge wurde auch tatsächlich mit EdgeFsm um die Übergangsrelation erweitert. Nachteil davon ist, dass auch z.B. Methoden zum Hinzufügen im Automaten durch die andere Klasse überschrieben werden müssen. Auch erfordert dies öfters Casts, wenn zunächst nur die allgemeine Kantenklasse vorliegt und die Automatenereigenschaften benötigt werden. Wegen dieser Unbequemlichkeiten wurde bei den Knoten zunächst darauf verzichtet, diese zu erweitern, sondern die benötigten Felder und Methoden liegen in der allgemeinen Knotenklasse Vertex, da diese Graphen nicht stören. Der Sauberkeit halber sollte eine Trennung in Vertex- und VertexFsm-Klasse nachgeholt werden.

2.7 Simulation

Zunächst war die Simulation so implementiert, dass in einer Liste die aktiven Zustände (und zur Darstellung aktive Kanten) gespeichert wurden und global der Resteingabe-String. Bei der Berechnung des nächsten Schrittes wurde von jedem aktiven Zustand die Erreichbarkeit mit der Resteingabe überprüft und die Resteingabe entsprechend angepasst. Die neuen aktiven Elemente waren dann die benutzen Kanten mit ihren Zielzuständen. Spontan erreichbare Zustände wurden nach jedem Berechnungsschritt hinzugefügt. Das funktioniert einfach und gut, solange die Blocklänge konstant ist und keine spezielle Berechnung nachvollzogen werden soll, denn dafür werden mehr Informationen gebraucht, bzw. unterschiedliche Berechnungen können eine unterschiedliche Resteingabe haben.

Jetzt hält die Simulation eine Liste von Berechnungen, was einer Liste einer Liste von Konfigurationen entspricht, wobei eine Konfiguration ein aktiver Zustand mit der dazugehörigen Resteingabe ist. Eine Konfiguration wird durch die Klasse Configuration repräsentiert. Zusätzlich kann zur Visualisierung die aktive Kante gespeichert werden. Die Klasse Configuration bietet eine Methode, um eine Liste von Folgekonfiguration zu berechnen. Spontane Übergänge erfolgen nun wie andere Übergänge auch

in einem eigenen Simulationsschritt. Die Liste der aktiven Elemente wird weiterhin für die Visualisierung genutzt und muss entsprechend nach einem Berechnungsschritt aus der Konfiguration berechnet werden.

Bei der Berechnung der Folgekonfigurationen wird jeder von dem Zustand der Konfiguration aus mögliche Übergang mit dem Beginn der Resteingabe gleicher Länge verglichen. Dabei gibt es einige Besonderheiten. Any-Übergänge funktionieren immer. Spontane Übergänge auch, verbrauchen aber keinen Buchstaben und müssen für Else-Übergänge gezählt werden. Else-Übergänge müssen zunächst gemerkt werden, diese werden nur den Übergängen hinzugefügt, wenn nur spontane gemacht wurden (deshalb müssen sie gezählt werden). Tritt keiner dieser Sonderfälle auf, wird der Block (auch Länge 1) zeichenweise verglichen. Dabei müssen die Abkürzungssymbole umgesetzt werden.

Der Start der Simulation erfolgt mit der Methode `startSimulation(String input)`. Beim Start der Simulation werden Konfigurationen mit der vollen Eingaben für jeden Startzustand in die Liste eingetragen. Der nächste Simulationsschritt wird mit der Methode `nextStep()` berechnet. Dabei wird für jede letzte Konfiguration der Berechnung in der Liste der Berechnungen die Folgekonfigurationen berechnet. Die alte Berechnung wird aus der Liste entfernt und für jede Folgekonfiguration wird die alte Berechnung konkateniert mit der entsprechenden Folgekonfiguration wieder hinzugefügt. Dabei muss explizit der Fall ausgeschlossen werden, dass ein spontaner Kreis geschlossen wird, was nicht zur Erkennung der Sprache beiträgt, aber unendlich lange Berechnungen produziert. Dafür wird für jede Konfiguration mit gleichem Input überprüft, ob bereits eine Konfiguration mit dem gleichen Zustand existiert.

2.8 Algorithmen

2.8.1 Automaten optimieren

Zu dieser Kategorie zählen Entfernen von unproduktiven Zuständen, Entfernen von unerreichbaren Zuständen, Entfernen unnötiger und zusammenfassen mehrfacher Kanten und das Hinzufügen eines Hotel-California-Zustanden. All dies sind wenig komplexe Algorithmen.

Für das Auffinden von unerreichbaren Zuständen wird von den Startzuständen aus eine Breitensuche durchgeführt, für das Auffinden von unproduktiven Zuständen wird dies im dualen Automaten gemacht. Dabei werden die erreichbaren Zustände durch die Startzustände initialisiert. Zu diesen Zuständen wird sich gemerkt, ob sie schon verarbeitet wurden. Für alle erreichbaren, aber nicht verarbeiteten Zustände, werden die Nachbarn zu den erreichbaren Zuständen hinzugefügt, anschließend wird dieser Zustand als verarbeitet markiert. Wenn keine Änderungen mehr auftreten, also alle

erreichbaren Zustände verarbeitet wurden, sind alle erreichbaren Zustände gefunden und die übrigen können entfernt werden.

Für die Optimierung der Kanten wird durch alle Kanten iteriert. Ist die Übergangsmenge leer, wird die Kante entfernt. Andernfalls wird überprüft, ob bereits eine Kante mit den selben Start- und Zielzuständen gefunden wurde. Ist dies der Fall, werden alle Übergänge den Übergängen der anderen Kante hinzugefügt und diese Kante entfernt.

Das Hinzufügen eines HCZ kann intelligent oder nicht intelligent geschehen. Bei der nicht intelligenten Form wird einfach ein Zustand, welcher weder Start- noch Endzustand ist, ergänzt und von jedem Zustand aus wird ein Else-Übergang zu diesen Zustand hinzugefügt. Bei der intelligenten Version muss zunächst das Alphabet berechnet werden. Dann werden nur dann Übergänge mit den verbleibenden Zeichen hinzugefügt, wenn der Zustand lokal nicht vollständig ist. Wie bei der globalen Vollständigkeit, ist die Überprüfung auf lokale Vollständigkeit bei variabler Blocklänge sehr schwierig. Außerdem könnte überprüft werden, ob bereits ein HCZ existiert und dieser benutzt werden. Die intelligente Variante ist noch nicht implementiert.

2.8.2 Dualer Automat

Die Berechnung des dualen Automaten ist sehr einfach. Zunächst wird durch alle Zustände iteriert. Ist ein Zustand Startzustand, wird er Endzustand und umgekehrt. Anschließend wird durch alle Kanten iteriert und Start- mit Zielknoten vertauscht.

2.8.3 Epsilon-Elimination

Die Elimination spontaner Übergänge kann durch Absorption von links, rechts oder beidseitig erfolgen. Bei der Absorption von links werden erst spontane Übergänge gemacht und dann der normale Übergang angehängt, von rechts erst normaler Übergang und dann spontane Übergänge und beidseitig werden vorher und nacher spontane Übergänge gemacht.

Es wird also von jedem Zustand n aus bei nicht rechts die Epsilon-Erreichbarkeit berechnet, sonst ist nur n erreichbar. Nun wird vom Zustand n für jede Kante e eine Kante mit dem Übergang von n zum Zielknoten von e hinzugefügt. Bei nicht links außerdem zu den vom Zielknoten von e durch spontane Übergänge erreichbaren Zuständen. Es spielt keine Rolle, ob dabei spontane Kanten kopiert wurden. Die neuen Kanten müssen gespeichert werden, da die neuen Kanten im vorigen Schritt nicht benutzt werden dürfen. Erst am Ende werden die spontanen Übergänge selbst entfernt.

2.8.4 Potenzmengenkonstruktion

Durch die Sonderfunktionen bei der Berechnung ist die Potenzmengenkonstruktion sehr aufwändig.

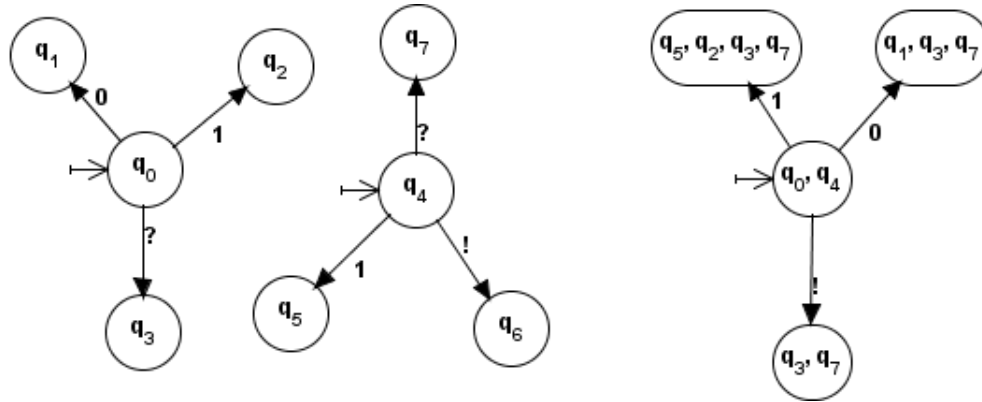
Zunächst muss von jedem Knoten die Erreichbarkeit (ohne spontane Übergänge) berechnet werden. Diese wird in einer Map gespeichert, die von Übergangslabeln auf eine Menge von Zuständen abbildet. Dabei wird jeder mögliche Übergang betrachtet, wobei Any und Else-Übergänge nicht gesondert betrachtet werden und spontane Übergänge übersprungen werden. Der durch den Übergang erreichbare Zustand sei n . Existiert dieser Übergang in der Map, wird n der dazugehörigen Zustandsmenge hinzugefügt. Existiert er nicht, muss überprüft werden, ob es eine Überschneidung von diesem Übergang mit einem Übergang in der Map gibt, was durch die abkürzenden Symbole vorkommen kann. Gibt es eine Überschneidung, müssen dieser Eintrag entfernt und bis zu drei neue Einträge hinzugefügt werden: ein Eintrag, mit dem Übergangslabel, das übrig bleibt, wenn man die Überschneidung von dem ursprünglichen Übergang abzieht, und der ursprünglichen Zustandsmenge, ein Eintrag mit der Überschneidung und der ursprünglichen Zustandsmenge erweitert um n und ein Eintrag mit dem Übergangslabel, das übrig bleibt, wenn man die Überschneidung vom aktuellen Übergang abzieht, und n . Gibt es keine Überschneidung wird ein neuer Eintrag mit dem aktuellen Übergang und n angelegt.

Um die Überschneidung zu berechnen, werden beide Übergänge zeichenweise verglichen. Stimmen die Zeichen überein (auch bei Symbolabkürzungen), ist dies ein Teil der (potentiellen) Überschneidung. Ist dies nicht der Fall, müssen beide Zeichen auf Symbolabkürzungen überprüft und ggf. miteinander verglichen werden. Finden sich dabei keine gleichen Zeichen, gibt es keine Überschneidung. Beim Vergleich ist es möglich, dass mehrere Zeichen gleich sind und so mehrere Überschneidungen entstehen.

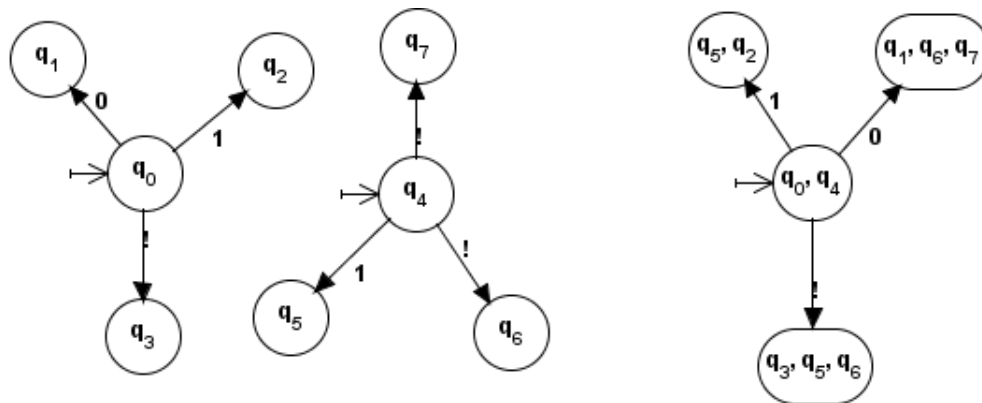
Um die Reste zu berechnen, wird für jedes Symbol des ersten Strings die entsprechende Menge von Zeichen angelegt, wobei die Zeichen des zweiten Strings jeweils abgezogen werden. Jedes in den Mengen verbliebende Zeichen wird dann einmal an entsprechender Stelle im ersten Originalstring eingesetzt und der Menge der Reste hinzugefügt.

Bei der eigentlichen Potenzmengenkonstruktion wird mit dem Metazustand, der aus der Menge der Startzustände besteht gestartet. Für diesen und jeden weiteren entstandenen Metazustand wird folgendes getan. Die Übergänge aller Zustände der Zustandsmenge des Metazustandes werden zusammengefasst und zwar auf die gleiche Art wie bei der Berechnung der Erreichbarkeit von einem Knoten aus. Bei dieser Gelegenheit werden gleich leere Else-Übergänge entfernt, also wenn es zu dem Else-Übergang noch ein Any-Übergang beim gleichen Knoten gibt. Anschließend müssen Any- und Else-Übergänge gesondert behandelt werden. Die Zustände, die über das Any-Symbol erreichbar sind, müssen jedem Übergang hinzugefügt werden. Die Zustände,

die über das Else-Symbol erreichbar sind, müssen jedem Übergang der anderen Knoten hinzugefügt werden. Any-Symbole werden anschließend zu Else-Symbolen. Nun werden entsprechend der berechneten Übergänge Kanten und - sofern sie noch nicht existieren - neue Metazustände angelegt.



- (a) Any: Der zusammengesetzte Automat links wird zum Potenzmengenautomat rechts. Der Else-Übergang zu q_6 entfällt. Die Ziele der Any-Übergänge müssen sowohl den Zielen von 1 also auch den Zielen von 0 hinzugefügt werden. Damit kann der Any-Übergang dann zum Else-Übergang werden, um Determinismus zu gewährleisten.



- (b) Else: Der zusammengesetzte Automat links wird zum Potenzmengenautomat rechts. Die Else-Übergänge werden jeweils nur den Übergangszielen der Übergangsetiketten des anderen Teilautomaten hinzugefügt.

Abbildung 2.3: Beispiele Potenzmengenkonstruktion Any und Else-Übergänge

2.8.5 Automateneigenschaften

Vollständigkeit: Um die Vollständigkeit zu überprüfen, muss zunächst das Alphabet berechnet werden. Für variable Blockgrößen ist schwer feststellbar, ob der Automat vollständig ist. Bei konstanter Blockgröße größer als 1, ist zu entscheiden, ob ein Automat als vollständig gelten kann, da in dem Fall die Eingabe ein Vielfaches der Blockgröße sein muss, damit der letzte Schritt funktionieren kann. Grundidee ist für

2 Implementierung

Automaten ohne Blöcke zu kontrollieren, ob für jedes Zeichen des Alphabetes ein Übergang existiert.

Hinweis: Die Funktion ist für Blöcke nicht vollständig implementiert.

Determinismus: Beachte die hier benutzte Definition von Determinismus (1.2). Bei der Überprüfung, ob der Automat deterministisch ist, muss überprüft werden, ob von einem Zustand bzw. der Epsilon-Erreichbarkeit ein Übergang Präfix eines anderen ist (also evt. auch gleich ist).

Hinweis: Die Funktion ist für Blöcke nicht vollständig implementiert.

Alphabet: Die Berechnung des Alphabetes erfolgt, indem jeder Übergang betrachtet wird und eine Liste der benutzten Zeichen erstellt wird. Blöcke werden dabei in ihre einzelnen Zeichen zerlegt. Any-, Else- und spontane Symbole werden dabei nicht berücksichtigt. Symbolabkürzungen werden umgesetzt.

Spontanität: Die Überprüfung, ob der Automat spontane Symbole benutzt, erfolgt durch ein Durchsuchen sämtlicher Übergänge nach dem spontanen Symbol.

Initial- bzw Finalzustande: Für die Rückgabe der Initial- bzw Finalzustande werden alle Zustände durchlaufen und ein Zustand der Liste hinzugefügt, wenn er entsprechend Initial- oder Finalzustand ist.

3 Abstandsautomaten

Dieses Kapitel beschreibt die theoretischen Ergebnisse in Bezug auf die Abstandsautomaten. Das Muster sei dabei der „fehlerfreie“ Text, mit dem eine Eingabe verglichen werden soll.

3.1 Spezielle Abstandsautomaten

Der **Abstand** oder die **Distanz** zwischen zwei Wörtern ist definiert als die minimale Anzahl an Operationen, die benötigt werden, um ein Wort in das andere zu überführen. Welche Operationen dabei erlaubt sind, hängt von dem Abstand ab. Abstände sind Metriken auf dem Raum von Symbolsequenzen. Eine Metrik ist eine Funktion, die jedem Paar von Elementen des Raums einen nicht negativen reellen Wert zuordnet. Metriken sind definit, symmetrisch und erfüllen die Dreiecksungleichung.

3.1.1 Hamming-Abstand

Beim Hamming-Abstand ist nur Substitution von Buchstaben erlaubt. Das bedeutet insbesondere, dass Muster und Eingabe die gleiche Länge haben müssen.

Die unterste Zeile ist die Zeile ohne Fehler, jeweils eine Zeile höher bedeutet ein Fehler mehr. Die Anzahl der Zeilen entspricht also dem Abstand + 1. Mit jedem Buchstaben der Eingabe wandert man eine Spalte nach rechts. Ein waagerechter Else-Übergang bedeutet ein zum Muster passender Buchstabe, ein diagonal ein nicht passender.

3.1.2 Levenshtein-Abstand

Der Levenshtein-Abstand erlaubt zusätzlich zur Substitution von Buchstaben auch das Einfügen und Löschen von Buchstaben.

Der Grundaufbau ist derselbe wie bei dem Hamming-Automaten. Es kommen lediglich zwei Übergangstypen hinzu. Ein senkrechter Else-Übergänge bedeutet ein Einfügen eines Buchstabens, ein diagonal spontaner Übergänge das Löschen eines Buchstabens.

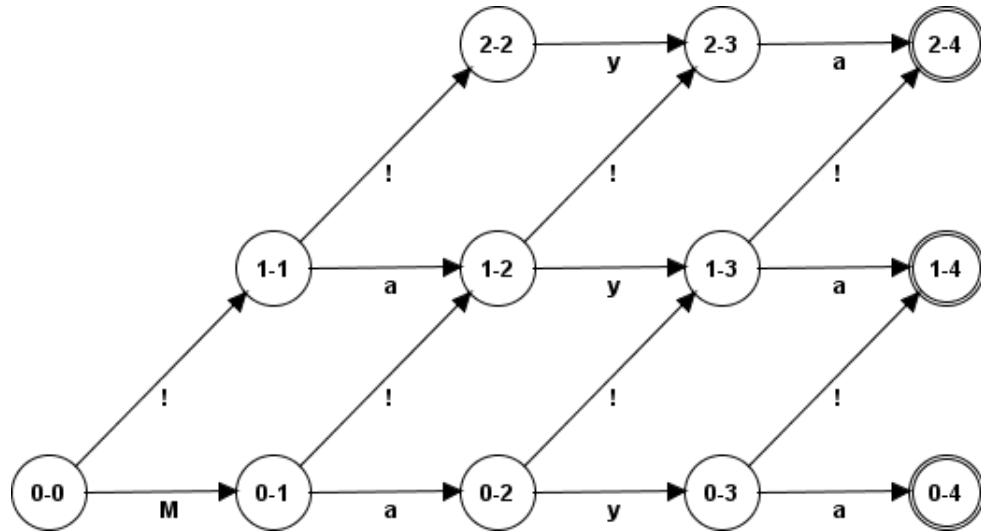


Abbildung 3.1: Ein Hamming-Automat mit dem Abstand 2 für das Muster Maya

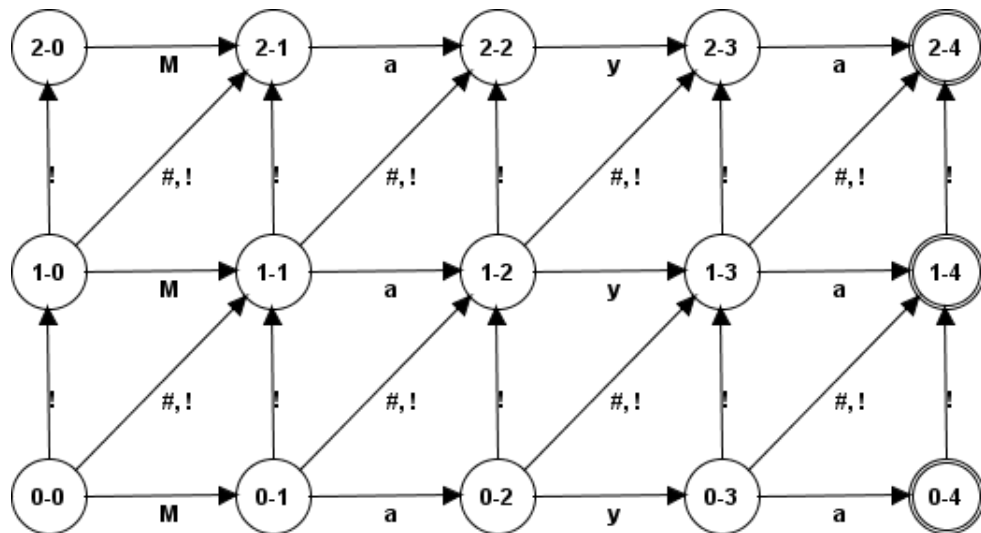


Abbildung 3.2: Ein Levenshtein-Automat mit dem Abstand 2 für das Muster Maya

3.1.3 Damerau-Levenshtein

Der Damerau-Levenshtein-Abstand erlaubt neben der Substitution auch das Einfügen und Löschen von Buchstaben Transposition benachbarter Buchstaben. In der Praxis ist dies relevant, da Vertauschen benachbarter Buchstaben ein typischer Tippfehler ist. Sowohl Substitution als auch Transposition könnte durch zwei Operationen (Einfügen und Löschen) erreicht werden. Jedoch führt dies zu einem anderen Abstand.

3.2 Motivation universeller Automaten

Anwendungsgebiet von Abstandsautomaten ist z.B. die Rechtschreibkorrektur, wobei ein Wort, das nicht im Wörterbuch gefunden wird mit einem bestimmten Abstand auf jedes Wort im Wörterbuch getestet wird. Die akzeptierten Wörter bilden dann die Menge der Korrekturvorschläge. Andere Anwendungen wären z.B. alternative Suchvorschläge („Meinten Sie ...“) oder bei fehlerbehafteter Datenübertragung.

Der Test des Abstandes muss oft sehr schnell gehen, da sehr viele Tests durchgeführt werden sollen. Während der Automat für Hamming deterministisch ist, gilt dies nicht für Levenshtein- und Damerau-Levenshtein-Automaten (Problem ist die Entscheidung welche Operation benutzt werden soll). Simulation von Nichtdeterminismus bedeutet einen deutlich größeren Aufwand. Auch das Umwandeln der NEAs in äquivalente DEAs durch Potenzmengenkonstruktion ist relativ aufwändig und die resultierenden Automaten wären sehr groß (Abhängig von der Länge des Musters).

Dadurch ist die Idee motiviert, universelle Abstandsautomaten einzuführen. Dabei muss nur ein einziges mal (für einen festen Abstand) der Automat konstruiert werden (und ggf. deterministisch gemacht werden). Der Preis dafür ist eine notwendige Kodierung der Eingabe vor der Simulation.

Die Idee eines universellen Levenshteinautomaten wurde in [1] vorgestellt. Ziel dieser Arbeit ist es, die dort vorgestellte Form des Automaten zu untersuchen und Funktionsweise und Konstruktion zu erläutern. Außerdem sollen universelle Abstands-Automaten auch für den (einfacheren) Hamming-Abstand und den (komplizierteren) Damerau-Levenshtein-Abstand untersucht werden.

3.3 Ergebnisse

3.3.1 Hamming-Abstand

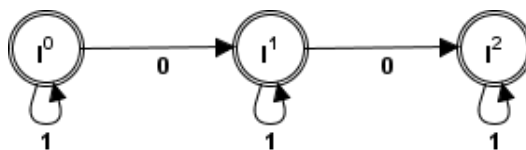


Abbildung 3.3: Ein universeller Hamming-Automat mit dem Abstand 2

Ein universeller Abstandsautomat für den Hamming-Abstand ist durchaus möglich, aber überflüssig. Zum einen ist der spezielle Hamming-Automat bereits deterministisch. Außerdem ist der Aufwand zu Kodierung der Eingabe bereits mindestens so groß wie die Berechnung des Abstandes selbst. Da beim Hamming-Abstand nur Substitution erlaubt ist, reicht es zur Berechnung des Abstandes, jeden i -ten Buchstaben

auf Gleichheit zu testen und bei jeder Ungleichheit den Abstand um 1 zu erhöhen. Dieser Vergleich muss beim Kodieren der Eingabe ebenfalls geschehen und entsprechend als 0 im Falle der Ungleichheit und 1 im Falle der Gleichheit an der i -ten Stelle in der codierten Eingabe auftauchen. Der Fall der ungleichen Längen muss gesondert behandelt werden, da in diesem Fall der Abstand nicht definiert ist. Die kodierte Eingabe könnte in diesem Fall aus einem Dummy-Symbol bestehen oder eines oder mehrere Dummy-Symbole beinhalten, die keinen Übergang im Automaten haben.

3.3.2 Levenshtein-Abstand

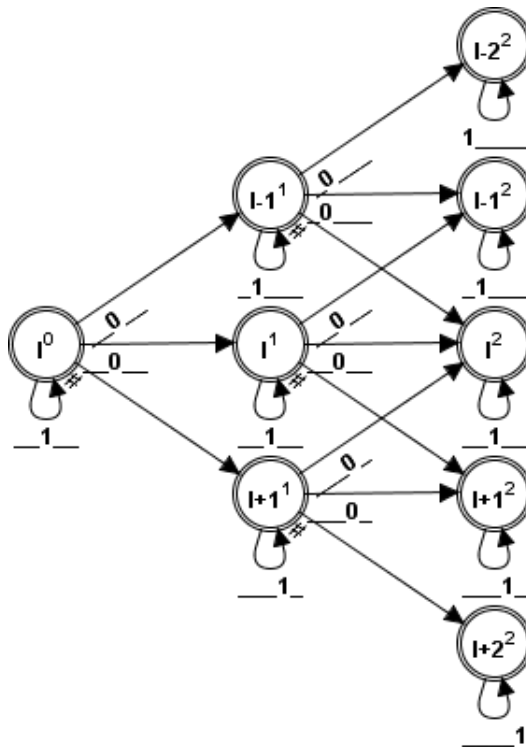


Abbildung 3.4: Ein universeller Levenshtein-Automat mit dem Abstand 2

Die Idee beim universellen Automaten für den Levenshtein-Abstand ist, dass in jedem Schritt die gleiche Entscheidung getroffen werden muss, nämlich ob ein passender, ein löschender, ein einfügender oder ein substituierender Schritt gemacht werden soll. Die einzigen Vorinformationen, die benötigt werden, sind, wieviele Fehler bereits gemacht wurden und mit dem wievielten Buchstaben des Musters verglichen werden muss. Das ist notwendig, denn die Zahl der Berechnungsschritte lässt sich nicht eins zu eins auf die Buchstaben des Musters abbilden. Wurde ein Buchstabe gelöscht, muss ein Buchstabe weiter vorne im Muster abgeglichen werden. Wurde ein Buchstabe eingefügt, muss ein Buchstabe weiter hinten abgeglichen werden. Substitution und passende Übergänge ändern dies nicht. Die Anzahl der Fehler lassen sich leicht im Zustand kodieren. Allerdings muss in jedem Zustand der Vergleich zu jedem möglichen

Musterbuchstaben (beschränkt durch den Abstand) zur Verfügung stehen. Welcher Vergleich betrachtet wird, wird wieder im Zustand kodiert.

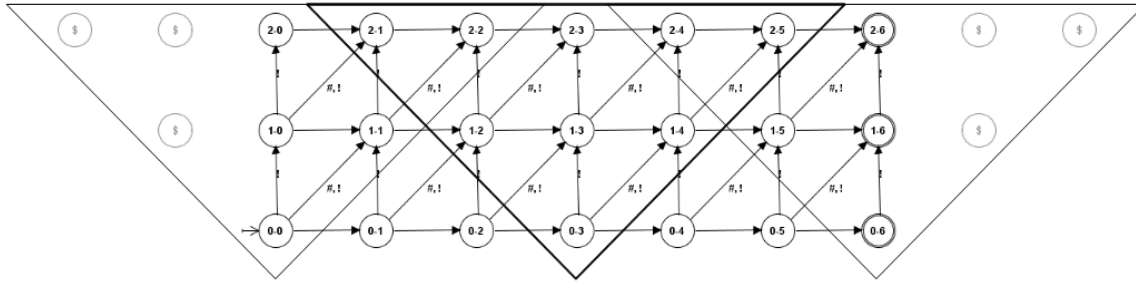


Abbildung 3.5: Symbolische Dreiecke - erreichbare Zustände nach n verarbeiteten Buchstaben

Betrachtet man die erreichbaren Zustände nach n verarbeiteten Buchstaben der Eingabe im speziellen Automaten liefert das ein gleichschenkliges Dreieck mit der Spitze unten (vgl. Bild). Dies sind die Zustände, die im universellen Automaten vorkommen. Damit für den Beginn und das Ende, wenn die Dreiecke nicht mehr vollständig sind, keine gesonderte Behandlung erfolgen muss, werden einfach virtuelle Zustände an den fehlenden Stellen angenommen. Dies findet sich auch in der Kodierung wieder.

Die Kodierung muss also für jeden Berechnungsschritt einen Block der Größe $2 * k + 1$ liefern, wobei k der Abstand ist. Dieser Block wird in Mihov/Schulz charakteristischer Vektor genannt. Jedes Zeichen des charakteristischen Vektors ist entweder eine 0 oder eine 1, je nachdem ob der Buchstabe der Eingabe mit dem entsprechenden Buchstaben des Teils des Musters übereinstimmt. Da bei Beginn der Simulation kein Fehler gemacht wurde, ist zunächst der mittlere Buchstabe relevant, deshalb muss der erste Teil der Musters in der Mitte beginnen und die ersten k Symbole sind 0 (werden z.B. mit einem Buchstaben verglichen, der im Alphabet nicht vorkommt, vergleiche virtuelle Zustände im oben beschriebenen symbolischen Dreieck). In jedem Schritt wird der Vergleichsblock um 1 nach links geschiftet und mit dem nächsten Buchstaben aus dem Muster aufgefüllt. Ist das Muster zu Ende, werden wie am Anfang Dummy-Buchstaben eingefügt. Es müssen so viele charakteristische Vektoren erzeugt werden, bis die Eingabe zu Ende ist und das letzte Zeichen des Musters an erster Stelle im Vergleichsblock ist. All diese charakteristischen Blöcke werden dann konkateniert und bilden die tatsächliche Eingabe zur Simulation.

Die signifikante Stelle eines Zustandes sei Stelle Mitte + Löschvorgänge - Einfügevorgänge. Unabhängig von der Fehleranzahl gibt es in jedem Zustand eine Schleife mit einer 1 an der signifikanten Stelle, die restlichen Stellen spielen keine Rolle (dürfen also 0 oder 1 sein). Diese Schleife repräsentiert der Fall, dass kein Fehler an dieser Stelle gemacht werden muss. Für den Fehlerfall gibt es drei Übergänge zu den erreichbaren Zuständen der nächsten Fehlerstufe. Für einen Einfügevorgang und einen Substitutionsvorgang hat der Übergang an der signifikanten Stelle eine 0, wobei die anderen

Eingabesymbol	Vergleichsblock	kodierter Block/charakteristischer Vektor
M	\$\$May	00100
a	\$Maya	00101
j	Maya\$	00000
a	aya\$\$	10100
h	ya\$\$\$	00000
\$	a\$\$\$\$	01111

Tabelle 3.2: Beispiel: Eingabe Majah bei dem Muster Maya für den Abstand 2.

Zeichen wieder keine Rolle spielen. Es ist ebenso möglich, an der Stelle auch eine 1 zu akzeptieren, in diesem Fall wird der Automat allerdings nicht minimal (und der Potenzmengenautomat wird auch größer). Der Löschvorgang hat die Besonderheit, dass der soeben verarbeitete Buchstabe anschließend noch verglichen werden muss. Dafür gibt es zwei Möglichkeiten. Zum einen kann der Übergang mit einer 0 an der signifikanten Stelle und einer 1 an der signifikanten Stelle + 1 erfolgen. Dies deckt jedoch nicht den Fall ab, dass mehrere Löschvorgänge hintereinander gemacht werden können. Dafür werden zusätzliche Kanten in die höheren Fehlerstufen benötigt mit 0 an der signifikanten Stelle des Startzustandes und einer 1 an der signifikanten Stelle des Zielzustandes. Die Alternative ist, diesen Übergang spontan zu machen, was aber bedeutet diesen Übergang auch bei einem fehlerlosen Übergang zu ermöglichen, wodurch – wie oben – ein größerer Automat entsteht.

Der Startzustand ist der einzige Zustand in der fehlerfreien Stufe. Alle Zustände sind Endzustände. Die Erkennung des Endes der Eingabe wird durch die Kodierung des Wortes abgedeckt. Dieser NEA kann dann durch Potenzmengenkonstruktion in einen DEA umgewandelt werden.

3.3.3 Damerau-Levenshtein

Ausgegangen wird von dem universellen Levenshtein-Automaten aus dem letzten Abschnitt. Die Kodierung der Eingabe ist dieselbe.

Für jeden Zustand in allen Fehlerstufen außer der letzten, wird ein zusätzlicher Zwischenzustand benötigt. Dieser ist dadurch motiviert, dass eine Transposition zwei Vergleiche benötigt, um erkannt zu werden. Der Tausch eines Zeichens von hinten nach vorne und umgekehrt ist äquivalent, es muss also nur ein Fall behandelt werden. Von dem Ursprungszustand zu diesem Zwischenzustand geht also ein Übergang mit einer 0 an der signifikanten Stelle des Zustandes und einer 1 an der signifikanten Stelle + 1. Von diesem Zwischenzustand geht ein Übergang mit einer 1 an der signifikanten Stelle - 1 zu dem Zustand mit derselben signifikanten Stelle in der nächsten Fehlerstufe.

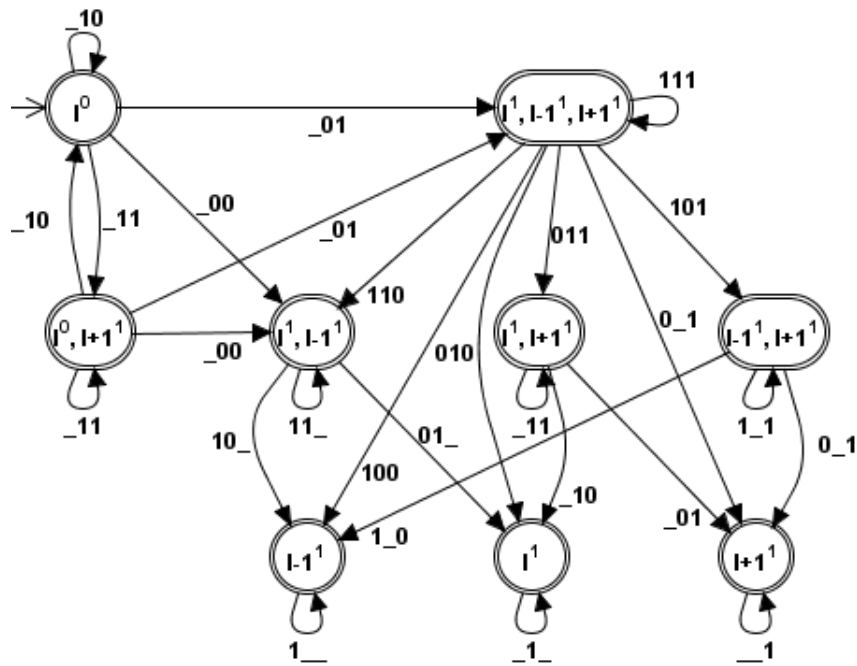


Abbildung 3.6: Die deterministische Version des Levenshtein-Automaten für den Abstand 1

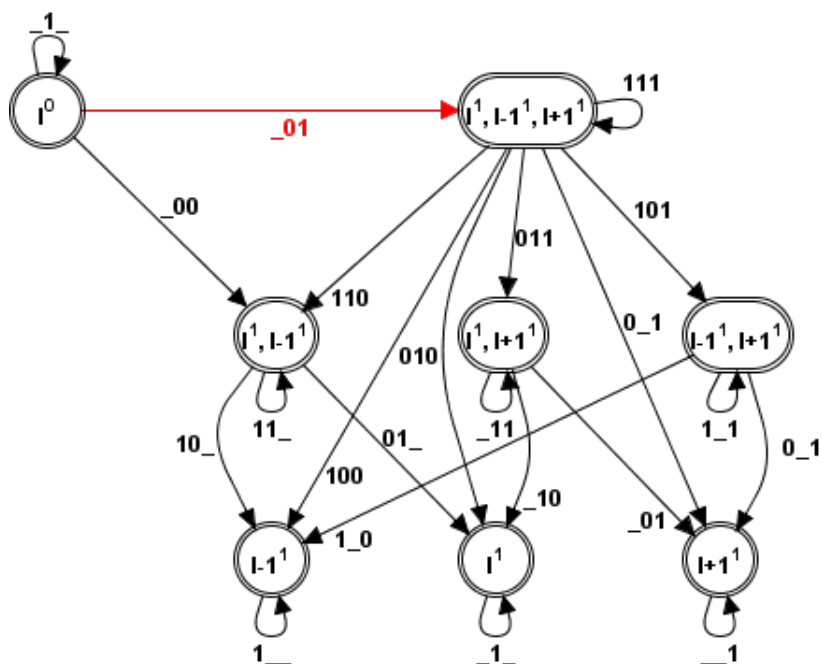


Abbildung 3.7: Die deterministische Version des Levenshtein-Automaten für den Abstand 1 mit geforderter 0 bzw. die minimierte Version.

Diese Übergänge sind in der Lage, einfache, unabhängige Tauschoperationen zu erkennen. Mehrfach verknüpfte Tauschoperationen oder Tauschoperationen in Verbindung mit Einfüge- oder Löschoptionen werden dadurch nicht erkannt. Eine Tauschoperation mit einem Substitutionsvorgang verknüpft, ergibt keinen Sinn. Außerdem müssen Tauschoperationen, bei denen ein bereits getauschter mit einem nicht getauschten Buchstaben vertauscht werden (also das „Durchreichen“ eines Buchstabens), nicht gesondert betrachtet werden. Dieser Effekt kann ab zwei Tauschoperationen mindestens mit genauso wenig Operationen erreicht werden, indem das Symbol an der „falschen“ Stelle gelöscht und an der „richtigen“ Stelle eingefügt wird (konstant zwei Operationen).

Betrachtet man Tauschvorgänge von jeweils bereits unabhängig getauschten Symbolen (relevant ab einem Abstand von 3), lässt sich feststellen, dass der gleiche Effekt durch eine Tauschoperation verknüpft mit einem Löscho- oder Einfügevorgang erreicht werden kann. Folgende Beispiele machen dies deutlich:

$abcd \rightarrow bacd \rightarrow badc \rightarrow bdac$

3 Tauschoperationen

$abcd \rightarrow bacd \rightarrow bdacd \rightarrow bdac$

3 Operationen: 1 Tausch mit 1 Einfügevorgang, 1 Löschovorgang

$abcde \rightarrow bacde \rightarrow badce \rightarrow bdace \rightarrow bdaec$

4 Tauschoperationen

$abcde \rightarrow bacde \rightarrow bdacde \rightarrow bdace \rightarrow bdaec$

4 Operationen: 1 Tausch mit 1 Einfügevorgang, 1 Tausch mit 1 Löschovorgang

Die benötigten Übergänge für die Einfüge- und Löschovorgänge werden analog zu denen im Levenshtein-Automaten eingefügt. Diese Übergänge sind notwendig; ob sie hinreichend sind, ist noch nicht abschließend untersucht.

3.4 Vergleich zu Mihov/Schulz

Die grundsätzliche Funktionsweise des hier vorgestellten Levenshteinautomaten ist die gleiche wie bei Mihov/Schulz. Es gibt jedoch ein paar Unterschiede. Mihov/Schulz benutzt eine getrennte Nicht-Endzustands- und Endzustandsmenge, sie erweitern also am Ende die symbolischen Dreiecke nicht um virtuelle Zustände. Entsprechend werden in der Kodierung, wenn das Muster zu Ende ist, keine Dummy-Symbole eingefügt, sondern der charakteristische Vektor wird kürzer. Die Anzahl der charakteristischen Vektoren ist hier genau die Länge der Eingabe. Dieses Vorgehen macht es allerdings nötig, den charakteristischen Vektor um eine Stelle zu erweitern. Diese Stelle wird benutzt, um zu überprüfen, ob das Ende der Eingabe erreicht ist, denn wenn der charakteristische Vektor nicht mehr vollständig ist, wird in die Endzustandsmenge gewechselt.

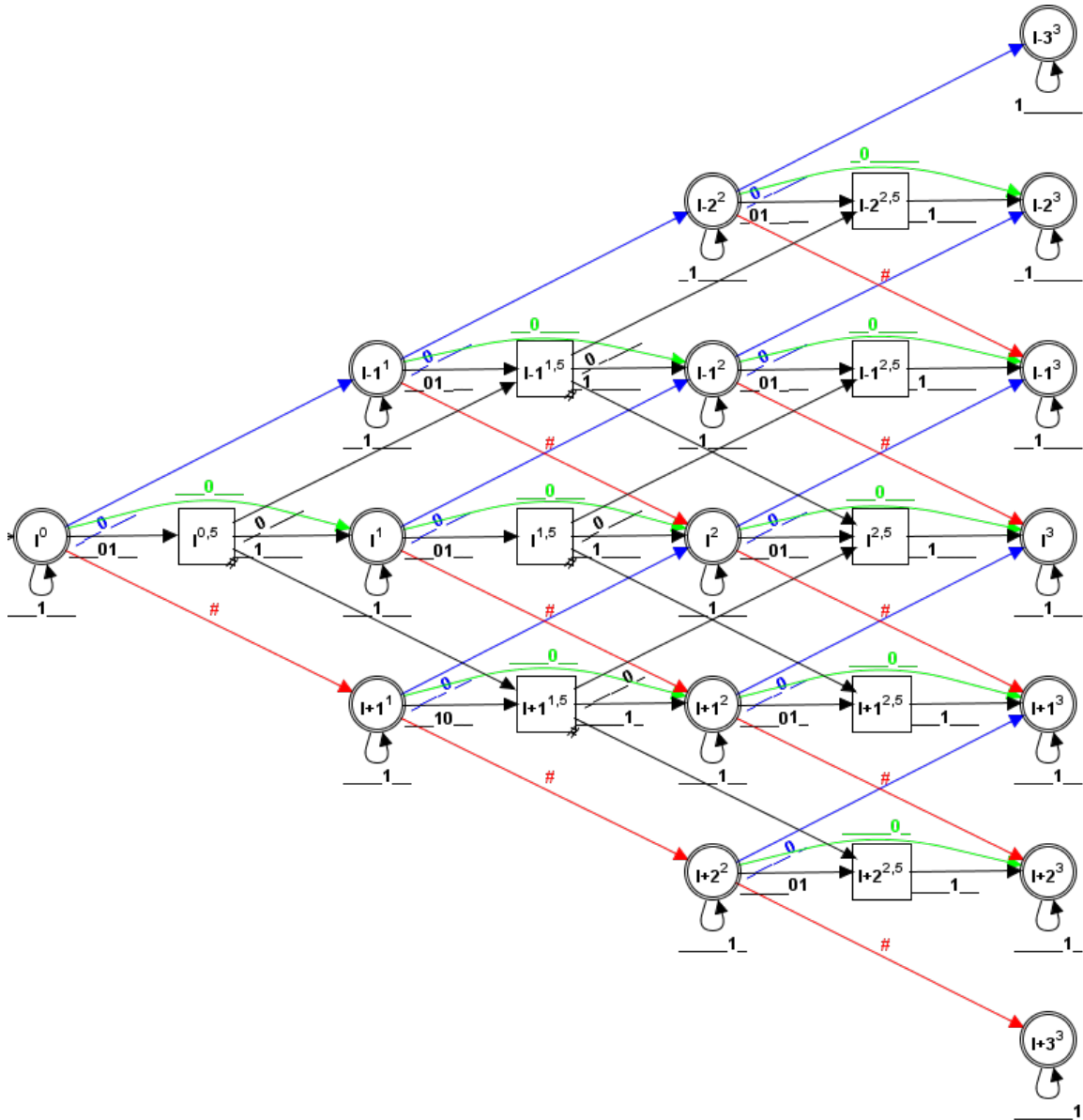


Abbildung 3.8: Der Versuch eines universellen Damerau-Levenshtein-Automaten für den Abstand 3. Die rechteckigen Zustände sind die zusätzlichen Zustände zum Levenshtein-Automaten. Die farbigen Zustände sind die Zustände aus dem Levenshtein-Automaten (rot Löschen, blau Einfügen, grün Substitution). Die schwarzen Übergänge sind die zusätzlich benötigten Übergänge. Der waagerechte Übergang durch den zusätzlichen Zustand ist ein normaler Tausch, der steigende ein zusätzliches Einfügen, der fallende ein zusätzliches Löschen.

Das bedeutet, dass der Mihov/Schulz-Automat eine größere Zustandsmenge hat (mehr als die Hälfte kommt nochmal dazu) und eine andere Eingabewortlänge. Auf der einen Seite wird pro Vektor ein Zeichen mehr benötigt, auf der anderen Seite werden die Vektoren am Ende kürzer. Es kann außerdem sein, dass bei der hier vorgestellten Methode mehr charakteristische Vektoren berechnet werden müssen. Allerdings ist eine konstante Blockgröße vorteilhaft bei der Simulation. Große Abweichungen der Wortlänge (also größer als der Abstand) kann bereits bei der Kodierung abgefangen werden, da diese Wörter nicht akzeptiert werden können. Das gleicht den großen Unterschied der Eingabegrößen bei kurzer Eingabe zu einem langen Muster aus.

Außerdem wurde bei Mihov/Schulz direkt ein DEA konstruiert ohne den Zwischenschritt über den NEA. Dabei betrachten sie direkt eine optimierte Potenzmenge der Zustände in dem symbolischen Dreieck, was genau den Zuständen entspricht, die die Potenzmengenkonstruktion bei den Übergängen mit geforderter 0 berechnet. Die Potenzmenge wird optimiert, indem geschaut wird, ob von mehreren Zuständen mit gleicher Eingabe ein Endzustand erreicht werden kann. Diese müssen dann nicht gesondert betrachtet werden. Die Zustände, die so zusammengefasst werden können, sind genau die, die in einem symbolischen Dreieck liegen; diese Zustände können dann durch den Zustand in der Spitze repräsentiert werden.

Die Konstruktion der Übergangsfunktion wird nicht explizit beschrieben. Es wird lediglich erklärt, dass die Übergänge so gebaut werden müssen, dass die aktiven Zustände für die Berechnungen übereinstimmen müssen. Das bedeutet, dass bei den Übergängen innerhalb einer Fehlerstufe an den signifikanten Stellen des Zielzustandes eine 1 stehen muss, und an den Stellen, die zwar signifikante Stellen des Start- aber nicht des Zielzustandes sind, eine 0 stehen muss. Für Übergänge von einer Fehlerstufe auf die nächste, gibt es zwei Übergänge, nämlich mit 01 an der signifikanten Stelle des Startzustandes zu dem Zustand mit den signifikanten Stellen -1, +0 und +1 relativ zur alten signifikanten Stelle und einen mit 00 an der signifikanten Stelle zum Zustand ohne die signifikante Stelle +1.

4 Future Work

Das Projekt bietet viele Erweiterungsmöglichkeiten. Einige seien hier vorgestellt.

Einige Features sind wegen Zeitmangel nicht vollständig implementiert oder nicht ausreichend getestet. Auch von kleineren Features für den Bedienkomfort, z.B. Defaultwerte rückwirkend zu setzen, Übergangslabel direkt als Blöcke eingeben, komfortables Drehen von Schleifen, sind nicht alle implementiert.

Zunächst wäre es sehr nützlich, (mindestens) einen funktionierenden Algorithmus zum Graphzeichnen zu haben. Gerade die bei der Potenzmengenkonstruktion entstehenden Automaten haben meist ein sehr schlechtes Layout.

Als Erweiterung für den Bedienkomfort wären Aspekte wie Zoom, um z.B. schnell einen Überblick über große Graphen zu bekommen, Mehrfachauswahl, um ganze Teile des Graphen zu verändern oder verschieben zu können und eine Rückgängig-Option nützlich. Interessant wäre eine Möglichkeit, Graphen nach TeX zu exportieren, wahlweise als TikZ-Bild oder als pstricks-Bild. Eine sehr große Erweiterung wäre z.B. die Erweiterung von Graphen auf eine dritte Dimension. Damit ließen sich komplexe Graphen teilweise deutlich besser anordnen. Zusätzliche graphische Elemente dürften sich zwar weitgehend durch Knoten simulieren lassen, aber eine bessere Unterstützung wäre durchaus vorstellbar. Insbesondere im Zusammenhang mit der Mehrfachauswahl wäre eine Methode zum Gruppieren von Elementen gut.

So wie die Automaten lediglich eine Erweiterung der Graphen sind, könnte man die Automaten recht leicht erweitern um einen Stack, um auch Kellerautomaten zu betrachten und simulieren. Unterstützung für Turingmaschinen wäre vermutlich etwas aufwändiger, aber diese Arbeit wäre eine geeignete Grundlage.

Weitere Algorithmen, die zu implementieren sind, wären der Kleene-Algorithmus zur Identifikation der Sprache durch reguläre Ausdrücke und die Minimierung von DEAs und evt. auch die aufwändigere Minimierung von NEAs. Aufwändig, aber interessant, wäre eine Simulation aller Algorithmen, was den Wert als didaktisches Begleitwerkzeug zur Theoretischen Informatik erheblich steigern würde.

Weiterhin könnte man das Programm als Grundlage für graphentheoretische Probleme benutzen. Eine Vielzahl von Algorithmen ließen sich implementieren, wie z.B. Breiten- und Tiefensuche, Dijkstra, Kruskal, Bellmann-Moore-Ford usw. Auch eine Untersuchung des Graphen auf besondere Eigenschaften wie Baumstruktur, Zusammenhang oder Planarität wären denkbar.

Literatur

- [1] STOYAN MIHOV, Klaus U. Schulz: Fast Approximate Search in Large Dictionaries. In: *Computational Linguistics* 30(4) (2004), S. 451 – 477.
– <http://www.cis.uni-muenchen.de/people/Schulz/Pub/fastapproxsearch.pdf> x, 37, 45