Semesterarbeit

Evaluation von verschiedenen Ansätzen für Evolutionäre Algorithmen

Adrian Schmid

Zürich, 22.05.2014

Zusammenfassung

Diese Semesterarbeit beschäftigt sich mit Grundlagen der Automatentheorie, Grundlagen von evolutionären Algorithmen und schliesslich einem konkreten praktischen Anwendungsfall dieser theoretischen Gebiete. Es geht um das finden von endlichen Automaten zu gegebenen regulären Ausdrücken mithilfe von verschiedenen evolutionären Algorithmen. Das Verhalten der verschiedenen Algorithmen wird anhand verschiedener Problemstellungen analysiert und die Resultate werden miteinander verglichen um schlussendlich die Stärken und Schwächen der verschiedenen Herangehensweisen aufzuzeigen.

Inhaltsverzeichnis

1. Einleitung

1.1. Ausgangslage

Evolutionäre Algorithmen sind Optimierungsverfahren die sich, inspiriert durch in der Natur ablaufende Prozesse, den grundlegenden evolutionären Mechanismen von Mutation, Selektion und Rekombination bedienen. In dieser Arbeit werden verschiedene Selektionsstrategien an einem einfachen Beispiel miteinander verglichen.

1.2. Ziel der Arbeit

Ziel der Arbeit ist es, an einem einfachen Beispiel, verschiedene Ansätze zum Entwurf von relativen Fitnessfunktionen gegeneinander abzuwägen. Dazu werden evolutionäre Algorithmen mit unterschiedlichen Strategien implementiert. Die aus den unterschiedlichen Ansätzen resultierenden Lösungen werden dann hinsichtlich ihrer Qualität und der für ihre Suche benötigten Ressourcen (Rechenzeit) verglichen.

1.3. Aufgabenstellung

In dieser Arbeit sollen evolutionäre Algorithmen implementiert welchen, welche zu einem gegebenen Regulären Ausdruck einen entsprechenden endlichen Automaten finden. Für dieses konkrete Beispiel sollen evolutionäre Algorithmen mit verschiedenen Strategien implementiert werden, umso das Konvergenzverhalten und die Rechenzeit der verschiedenen Implementationen (und somit der Strategien) untersuchen und vergleichen zu können.

- Einarbeitung in das Thema
- Repräsentation und Mutation von endlichen Automaten definieren
- Evolutionärer Algorithmus mit fester, globaler Problemmenge implementieren
- Evolutionärer Algorithmus mit mutierender, globaler Problemmenge implementieren
- Evolutionärer Algorithmus, bei welchem jeder Lösungskandidat seine eigene, mutierende Problemmenge mitführt implementieren
- Gegenüberstellung der Resultate

1.4. Erwartete Resultate

- Überblick über Evolutionäre Algorithmen und Endliche Automaten im technischen Bericht
- Implementierung einer Repräsentation von endlichen Automaten, Implementierung von Mutationen zu endlichen Automaten und Dokumentation des Implementierten im technischen Bericht
- Implementierung eines evolutionären Algorithmus mit fester, globaler Problemmenge mit einer entsprechenden Dokumentation im technischen Bericht
- Implementierung eines evolutionären Algorithmus mit mutierender, globaler Problemmenge mit einer entsprechenden Dokumentation im technischen Bericht
- Implementierung eines evolutionären Algorithmus, bei welchem jeder Lösungskandidat seine eigene, mutierende Problemmenge mitführt mit einer entsprechenden Dokumentation im technischen Bericht
- Dokumentation der Gegenüberstellung der Resultate

2. Grundlagen

2.1. Endliche Automaten und reguläre Sprachen

Reguläre Sprachen und endliche Automaten gehören zur Automatentheorie, einem Teilbereich der theoretischen Informatik. Zum besseren Verständnis dieser Arbeit habe ich folgend einige gängige und wichtige Konzepte der Automatentheorie frei nach dem Standartwerk von Hopcroft, Motwani und Ullman zusammengefasst. [3]

Alphabet Ein Alphabet ist eine endliche, nicht leere Menge von Symbolen. Üblicherweise wird ein Alphabet durch das Symbol Σ dargestellt.

Zeichenreihen Eine Zeichenreihe (auch Wort oder String) ist eine endliche Folge von Symbolen eines bestimmten Alphabetes.

Sprachen Eine Menge von Zeichenreihen aus Σ^*, wobei Σ ein bestimmtes Alphabet darstellt, wird als Sprache bezeichnet. Das heisst, eine Sprache ist eine Menge von Zeichenreihen die mit den Zeichen aus einem Alphabet Σ gebildet werden können. Bei der Bildung von Wörtern müssen nicht alle Zeichen des gegebenen Alphabets verwendet werden.

Bei der Klasse der *regulären Sprachen* handelt es sich um die Menge aller Sprachen, welche durch einen endlichen Automaten beschrieben werden können. Alle Sprachen dieser Klasse können auch durch einen regulären Ausdruck dargestellt werden. Das heisst, dass es für jede reguläre Sprache sowohl einen regulären Ausdruck als auch einen endlichen Automaten gibt, der alle Wörter der Sprache akzeptiert.

Ein deterministischer endlicher Automat besteht gemäss Hopcroft, Motwani und Ullman [3] aus:

1. einer endlichen menge von Zuständen die meist durch Q bezeichnet wird.
2. einer endlichen Menge von Eingabesymbolen (auch Alphabet Σ).
3. einer Übergangsfunktion δ der ein Zustand und ein Eingabesymbol übergeben werden und die einen Zustand zurückgibt.
4. einem Startzustand welcher Teil der Menge Q sein muss.
5. einer Menge F akzeptierender Zustände. Die Menge F ist eine Teilmenge von Q.

Oft werden endliche Automaten als Quintupel in der Form $A = (Q, \Sigma, \delta, q0, F)$ dargestellt.

Zum besseren Verständnis folgend ein Beispiel eines Automaten welcher alle Binärzahlen die durch drei teilbar sind akzeptiert.

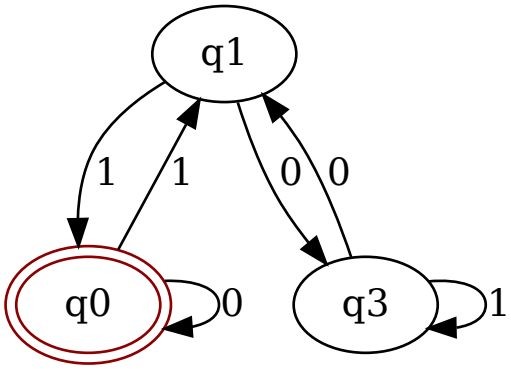

Abbildung 1: Beispiel: Endlicher Automat

Die Attribute des Quintupels für diesen Automaten sind:

1. $Q = \{q0, q1, q3\}$
2. $\Sigma = \{0, 1\}$
3. δ
4. $q0$
5. $F = \{q0\}$

Wobei man die Übergangsfunktion δ am einfachsten als Tabelle wie folgt darstellt:

	0	1
→q0*	q0	q1
q1	q3	q0
q3	q1	q3

Tabelle 1: Übergangstabelle

Konventionen Ich habe mich bei der Darstellung von Automaten im Rahmen dieser Arbeit auf folgende Konvention festgelegt:

- Zustände werden mit einem kleinen q gefolgt von einer Zahl bezeichnet (zB. $q1$)
- Zustände müssen nicht durchgängig nummeriert sein
- Zustände werden als Ellipsen dargestellt
- Zustandsübergänge sind beschriftete Pfeile
- der Startzustand erhält einen roten Rand
- alle akzeptierenden Zustände sind doppelt umrandet

2.2. Evolutionäre Algorithmen

«Unter evolutionären Algorithmen (EA) verstehen wir randomisierte Heuristiken, die Suchprobleme näherungsweise durch vereinfachende algorithmische Umsetzung von Prinzipien der natürlichen Evolution zu lösen versuchen. Somit geben evolutionäre Algorithmen in der Regel weder eine Garantie bezüglich der benötigten Rechenzeit noch der Güte der ausgegebenen Lösung. Ein Suchproblem besteht darin, zu einer Zielfunktion ein Element aus deren Definitionsbereich zu finden, dessen Funktionswert möglichst gut ist. Darunter verstehen wir im Folgenden, wenn nicht ausdrücklich anders vermerkt, einen möglichst grossen Funktionswert, weshalb der Zielfunktionswert eines Elements auch als seine Fitness bezeichnet wird.

Der Aufbau eines evolutionären Algorithmus lässt sich dann grob wie folgt beschreiben: in jedem Schritt verwaltet er eine Menge fester Grösse von Suchpunkten, die so genannte Population, wobei jeder einzelne Suchpunkt auch als Individuum bezeichnet wird. Aus den Punkten der Population neue Punkte zu erzeugen, ist Aufgabe von Mutation und Rekombination. Dabei steht hinter der Mutation die Idee, jeweils nur ein einzelnes Individuum zufällig zu verändern, ohne dass andere Individuen dabei berücksichtigt werden. Durch Rekombination wird hingegen aus mehreren, meist zwei Individuen zufällig ein neues gebildet, das von diesen möglichst gute Eigenschaften übernehmen soll. Durch Mutation und Rekombination werden also neue Individuen (Kinder genannt) aus bestehenden Individuen (Eltern genannt) erzeugt. Beide Operatoren hängen oftmals stark von Zufallsentscheidungen ab. Jedoch fliesst in der Regel weder in Mutation noch Rekombination der Zielfunktionswert der Individuen ein.

Die Zielfunktion beeinflusst nur die Selektion. Dieser Operator wählt Individuen der Population aus, sei es zur Auswahl der Eltern für eine Rekombination oder Mutation oder, um aus der Menge von Eltern und Kindern die nächste Population zu wählen, was den Übergang zur nächsten Generation darstellt. Dadurch, dass die Selektion Punkte

mit höherem Zielfunktionswert mit grösserer Wahrscheinlichkeit auswählt, soll erreicht werden, dass nach und nach immer bessere Punkte gefunden werden.»[1]

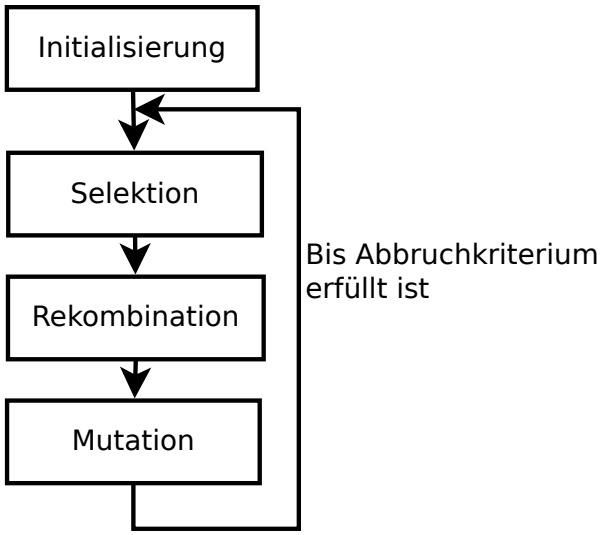

Abbildung 2: Evolutionärer Algorithmus [1]

2.2.1. Anwendung auf unser Problem

Unser Problem der Findung von regulären Automaten welche die selbe Sprache akzeptieren wie ein gegebener regulärer Ausdruck, lässt sich mit einigen Anpassungen in diesem Standardmodell eines evolutionären Algorithmus abbilden.

Initialisierung In der Initialisierungsphase wird eine Population von Automaten zufällig erzeugt. Das heisst eine eine zufällige Anzahl von Zuständen wird erzeugt, diese werden Zufällig miteinander verbunden und zum Schluss wird ein zufälliger Startzustand und eine Menge von akzeptierenden Zuständen festgelegt.

Selektion Es werden jeweils diejenigen Automaten selektiert welche eine höhere Fitness aufweisen.

Problemmenge Die Problemmenge ist eine Menge von Tupeln ($Wort$, $SollAusgabe$) welche verwendet wird um die Fitness von Automaten zu bestimmen. Im Rahmen dieser Arbeit werden verschiedene Ansätze zur Erzeugung und zum Verhalten solcher Problemmengen ausprobiert. Dabei wird das Konvergenzverhalten des evolutionären Algorithmus untersucht.

Fitness Die Fitness von Automaten wird durch ausprobieren verschiedener Problemen und dem darauf folgenden Vergleich der Ausgabe mit einer Soll-Ausgabe bestimmt. Die Soll Ausgabe wird mithilfe des gegebenen regulären Ausdrucks bestimmen. Der Wert der Fitnessfunktion entspricht der Anzahl richtig gelöster Probleme aus der entsprechenden Problemmenge.

Definitionen:
P = Problemmenge (Menge von Tupeln)
P_i = i-tes Element aus der Problemmenge (Tupel ($Wort$, $SollAusgabe$))
P_{i1} = 1. Element (Wort) des i-ten Elementes aus der Problemmenge
P_{i2} = 2. Element (Soll Ausgabe; 1 oder 0) des i-ten Elementes aus der Problemmenge
p = Anzahl Elemente in der Problemmenge P

$$Automat(Wort) = \begin{cases} 1 & \text{Wenn das Wort akzeptiert wird} \\ 0 & \text{Wenn nicht} \end{cases} \tag{1}$$

$$equal(x, y) = \begin{cases} 1 & \text{Wenn x und y gleich sind} \\ 0 & \text{Wenn nicht} \end{cases} \tag{2}$$

$$fitness(Automat, P) = \sum_{i=1}^{p} equal(Automat(P_{i1}), P_{i2}) \tag{3}$$

Daraus folgt:

$$\begin{aligned} \max fitness(Automat, P) &= p \\ \min fitness(Automat, P) &= 0 \end{aligned} \tag{4}$$

Rekombination Wenn man zwei endliche Automaten graphisch zusammenfügt, wird der resultierende Automat ein komplett anderes Verhalten aufweisen als die beiden Eltern. Deshalb verzichten wir auf die Rekombination im herkömmlichen Sinne. Wir klonen jeweils die selektierten Automaten und mutieren die so entstandenen Kinder.

Mutation Bei der Mutation soll jeweils eine zufällige Veränderung an einem Automaten durchgeführt werden. Wir haben uns auf folgende Mutationen festgelegt:

1. einen Zustand hinzufügen
2. einen Zustand entfernen
3. eine Verbindung zwischen zwei Zuständen ändern
4. einen nicht akzeptierenden Zustand zu einem akzeptierenden Zustand umwandeln
5. einen akzeptierenden Zustand zu einem nicht akzeptierenden Zustand umwandeln

3. Umsetzung

3.1. Automaten

Es wurde versucht Automaten sowohl möglichst nah an der theoretischen Vorgabe als auch möglichst effizient zu implementieren. Daraus hat folgende Grundimplementation resultiert.

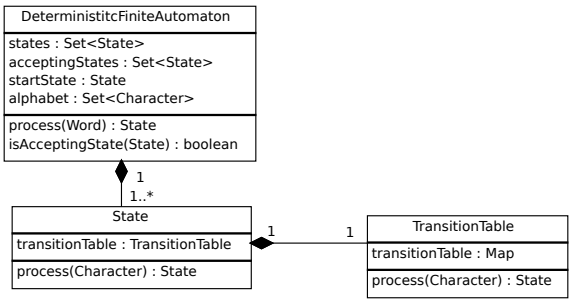

Abbildung 3: DFA Klassendiagramm vereinfacht

Das Quintupel (Q, Σ, δ, $q0$, F) ist darin wie folgt abgebildet:

Q	Die Menge der Zustände wurde als Set von States implementiert.
Σ	Das Alphabet ist ein Array von Zeichen (Character).
δ	Die Übergangsfunktion δ wird als Map (Zeichen -> Zustand) auf den einzelnen Zuständen abgebildet. Dies erlaubt uns eine einfache und effiziente Verarbeitung von Eingaben sowohl auf Zustands als auch auf Automaten Ebene.
$q0$	Eine Referenz zum Startzustand $q0$ ist in der DFA Klasse hinterlegt.
F	Die Menge der Akzeptierenden Zustände wird durch ein Set auf dem jeweiligen DFA Objekt representiert.

Tabelle 2: Implementation automaten

Ein solcher Automat kann nun mithilfe seiner process(Word) Funktion ein Wort (Eine Liste von Zeichen) verarbeiten indem er sich die Referenz des ersten Zustandes startState holt und dann Zeichen für Zeichen jeweils mit process(Character) die Referenz des nächsten Zustandes herausliest. Zurückgegeben wird schlussendlich derjenige Zustand der am

Ende der verarbeiteten Zeichenkette erreicht wurde. Mithilfe der isAcceptingState(State) Methode kann festgestellt werden ob das Wort akzeptiert wird oder nicht.

Zum besseren Verständnis folgend die process Methode der DFA Klasse als Pseudocode:

```
def process(word):
  state = startState
  for char in word:
    state = state.process(char)
  return state
```

Listing 1: Process Methode der DFA Klasse

Visualisierung Zur Visualisierung der Automaten wurde Graphviz [6] mit dem Graphviz Java API [7] verwendet. Konkret wurde ein Interface GraphvizRenderable entwickelt welches Klassen vorschreibt die generateDotString() Methode zu implementieren.

In der statischen Klasse GraphvizRenderer wurde eine Methode renderGraph(GraphvizRenderable , FileName) implementiert welche das API ansteuert und das gegebene GraphvizRenderable Objekt als SVG Vektorgraphik abspeichert.

Damit das generieren der Grafiken funktioniert, muss die generateDotString() Methode eine gültige Graphviz DOT Graphbeschreibung als String zurückgeben. Automaten sind eine Form von Digraphen und können in der DOT Sprache wie folgt beschrieben werden:

```
digraph G {
  q0 -> q1 [label="0"]
  q0 -> q2 [label="1"]
  q1 -> q2 [label="0"]
  q1 -> q0 [label="1"]
  q2 -> q0 [label="0"]
  q2 -> q1 [label="1"]
  q2 [peripheries=2]
  q0 [color=darkred]
}
```

Listing 2: Automat in DOT Sprache

- digraph G definiert den Graphentypen
- q0 -> q1 definiert den Übergang vom Knoten q0 auf den Knoten q1
- [label="0"] fügt die Beschriftung «0»dem Übergang hinzu
- [peripheries=2] umrandet den entsprechenden Knoten doppelt
- [color=darkred] färbt den entsprechenden Knoten dunkelrot ein

Eine Anleitung zur DOT Sprache ist in den Quellen verlinkt. [2]

Überprüfen von Automaten Um beim zufälligen mutieren und rekombinieren der Automaten prüfen zu können, ob ein Automat die selbe Sprache akzeptiert wie der gegebene reguläre Ausdruck, wurde das Java Package dk.brics.automaton von der dänischen Aarhus Universität verwendet.[5]

In diesem Package sind verschiedene Algorithmen rund um endliche Automaten implementiert. Unter anderem kann es einen Automaten aus einem regulären Ausdruck erzeugen und verschiedene Automaten darauf überprüfen ob sie die selbe Sprache akzeptieren. Die Umwandlung von einem Automaten unserer Implementation in einen dk.brics.automaton wurde im Package ch.zhaw.regularLanguages.dfa.transformation implementiert. Es enthält ein Interface Transformer<S, T> welches die Methode T transform(S input) vorschreibt und die konkrete Implementation TransformDFAToBricsAutomaton welches die Transformation von unserem DFA in einen dk.brics.automaton.Automaton beherrscht. Die Umwandlung läuft wie folgt ab:

1. Leerer «BRICS Automat»wird erzeugt
2. Für jeden unserer Zustände wird dem Automaten einen Zustand hinzugefügt
3. Die Übergangstabelle wird übertragen
4. Akzeptierende Zustände werden entsprechend markiert
5. Startzustand wird gesetzt

Vereinfachung Zur Erhöhung der Konsistenz von Automaten, wurde eine Methode zum entfernen aller nicht erreichbaren Zustände implementiert. Insbesondere verhindert dies eine «Aufsplittung»wie sie in Abbildung 4 zu sehen ist. In diesem Beispiel gibt es keine Verbindung vom Startzustand $q0$ zum einzig akzeptierenden Zustand $q4$.

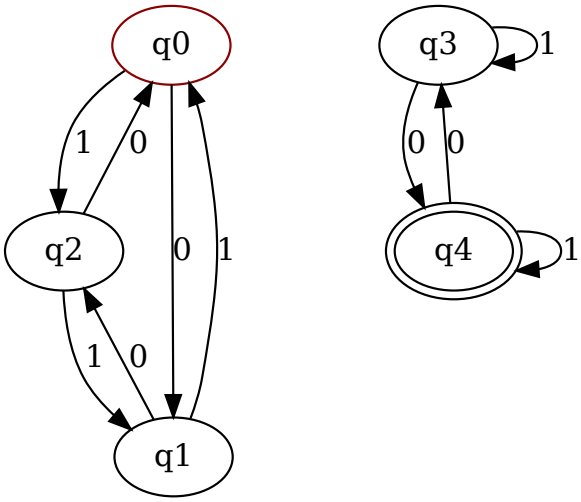

Abbildung 4: Aufgesplitteter Automat

In der Methode zum entfernen der nicht erreichbaren Zustände werden vom Startzustand aus alle möglichen Verbindungen getestet. Jene Zustände die so nicht erreicht werden können werden entfernt. Zum besseren Verständnis ist folgend die Methode als in Python-Pseudocode dokumentiert.

```
def removeUnreachableStates(dfa):
  processed = []
  current = dfa.getStartState()
  queue = [current]

  while queue.size > 0:
    nextList = []
    for char in dfa.getAlphabet():
      nextList.append(current.process(char))

    processed.append(current)
    queue.remove(current)

    for nextState in nextList:
      if !processed.contains(nextState):
        queue.append(nextState)

    current = queue[0]

  newStates = processed
  dfa.setStates(newStates)
```

Listing 3: Algorithmus zum entfernen von nicht erreichbaren Zuständen

3.1.1. RandomDeterministicFiniteAutomaton

Der RandomDeterministicFiniteAutomaton ist eine Erweiterung für unsere DFA Klasse, welche den normalen endlichen Automaten um einen zufälligen Konstruktor und eine mutate Methode zum zufälligen mutieren des Automaten erweitert.

Der Konstruktor erzeugt mithilfe eines gegebenen Alphabets und einem Faktor für die Komplexität des Problems - wie folgt beschrieben - zufällige Automaten.

1. Es wird eine Menge von Zuständen erzeugt (Anzahl Zustände: zwischen 1 und $(2 \cdot AnzahlZeichen \cdot Komplexitaet)$ wobei $AnzahlZeichen$ die Anzahl der Zeichen des Alphabets und die $Komplexitt$ ein Faktor ist welcher als Parameter an den Konstruktor übergeben wird).
2. Jedem Zustand werden für jedes Zeichen des Alphabets zufällig Verbindungen auf andere Zuständen zugeordnet.
3. Der erste Zustand (q0) wird zum Startzustand.
4. Um sicherzustellen dass jeder Automat erreichbare akzeptierende Zustände hat, werden alle nicht erreichbaren Zustände entfernt.
5. Eine zufällige Menge von Zuständen (zwischen 1 und $\frac{AnzahlZustaende}{5}$) wird zu akzeptierenden Zuständen.

Die mutate Methode greift auf ein Mutationsregister zurück in welchem die Methoden zum Verändern von Automaten registriert sind. Dabei wählt es zufällig eine Methode aus und führt diese durch. Wenn die Mutation erfolgreich war sind wir fertig. Wenn nicht wird erneut eine zufällige Methode ausgesucht. Dies wird solange wiederholt bis der Automat erfolgreich verändert wurde.

Aktion	**Beschreibung**	**Bedingungen**
Zustand hinzufügen	Es wird ein Zustand zum Automaten hinzugefügt. Für jedes Zeichen im Alphabet wird ein Übergang auf einen zufälligen Zustand des Automaten gelegt. Danach wird berechnet wieviele Verbindungen in diesem Automaten durchschnittlich zu einem Zustand führen (avg). Mit diesem Resultat werden zwischen $\frac{avg}{2}$ und $avg \cdot 2$ zufällig ausgewählte Verbindungen von zufällig ausgewählten Zuständen zum neuen gelegt.	-
Zustand entfernen	Es wird zufällig ein Zustand ausgewählt und entfernt. Alle Übergänge die auf diesen Zustand zeigen werden zufällig auf andere Zustände umgeleitet.	Nicht der letzte Zustand; Nicht der letzte akz. Zustand
Akz. Zustand hinzufügen	Es wird ein zufälliger nicht akzeptierender Zustand ausgewählt. Dieser wird zum akzeptierenden Zustand gemacht.	Nicht akzeptierende Zustände vorhanden
Akz. Zustand entfernen	Es wird ein zufälliger akzeptierender Zustand ausgewählt. Dieser wird zu einem regulären nicht-akzeptierenden Zustand umgewandelt.	Nicht der letzte akz. Zustand
Übergang ändern	Es wird ein zufälliger Zustand und ein zufälliges Zeichen ausgewählt. Der abgehende Übergang für das gewählte Zeichen wird zufällig auf einen anderen Zustand gelegt.	-

Tabelle 3: Mutationen

3.2. Evolutionäre Algorithmen

Bei der Implementierung der Evolutionären Ansätze wurde insbesondere Wert darauf gelegt, dass sowohl die theoretischen Grundlagen abgebildet sind als auch dass die verschiedenen Ansätze mit der grundsätzlich selben Struktur implementiert werden können um Redundanz im Code so gut als möglich zu vermeiden. Mit diesen Zielen vor Augen wurde folgende Grundstruktur für die Implementierung von evolutionären Algorithmen entworfen:

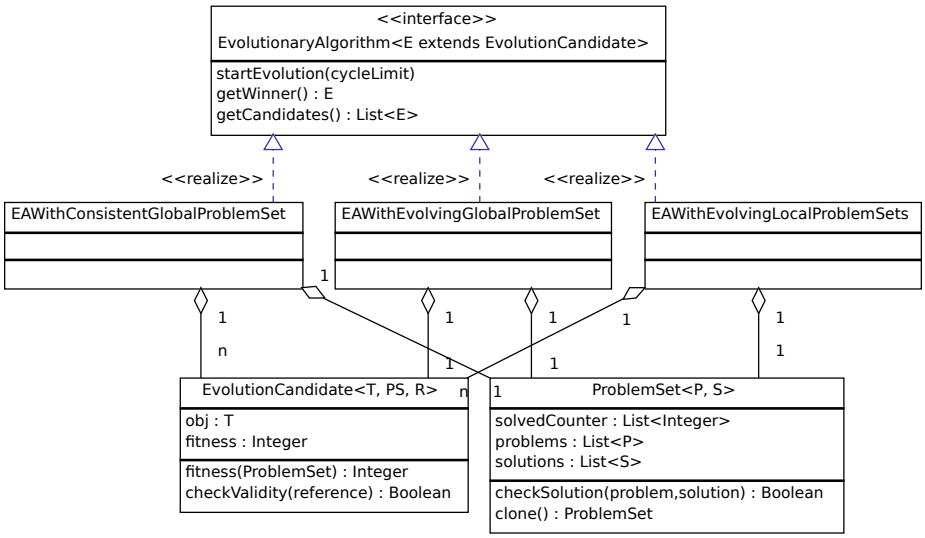

Abbildung 5: EA Grundstruktur vereinfacht

Die Klassen welche die Algorithmen beinhalten verwenden eine Menge von EvolutionCandidate (Population) und ein ProblemSet. Der evolutionäre Algorithmus kann jeweils mit startEvolution gestartet werden und wird abgebrochen sobald eine gültige Lösung gefunden wurde.

Das ProblemSet besteht Grundsätzlich aus zwei Listen. In der Ersten ist die «Problemstellung»abgelegt, in der zweiten die entsprechenden Musterlösung. Mithilfe der Methode checkSolution(problem, solution) kann geprüft werden ob eine erarbeite Lösung (Parameter solution) der Musterlösung entspricht. In unserem Fall wird so geprüft, ob unser Automat die Zugehörigkeit eines Wortes zu der gegebenen Sprache richtig erkennt oder nicht.

Der EvolutionCandidate entspricht einem Individuum aus unserer Population und besteht aus dem «obj», welches in unserem Fall den Automaten an sich beinhaltet, der fitness als Eigenschaft (wird zur Sortierung verwendet) und einer fitness Funktion welche ein ProblemSet entgegen nimmt und mithilfe des gegebenen «obj»versucht alle Probleme aus dem ProblemSet zu lösen. Die Fitness als Wert entspricht der Anzahl korrekt gelöster Probleme.

3.2.1. Initialisierung

Um die Klassen der evolutionären Algorithmen schlank zu halten, wurde die Initialisierung ausgelagert. In der Abbildung 6 sieht man, dass die Initialisierung eines evolutionären Algorithmus zur Findung eines endlichen Automaten zu einem gegebenen regulären Ausdruck in drei Schritten abläuft:

1. Sprache initialisieren (initLanguage(alphabet, maxWordLength, regexp))

2. Probleme initialisieren (initProblems(noProblems))
3. Population initialisieren (initCandidates(noCandidates))

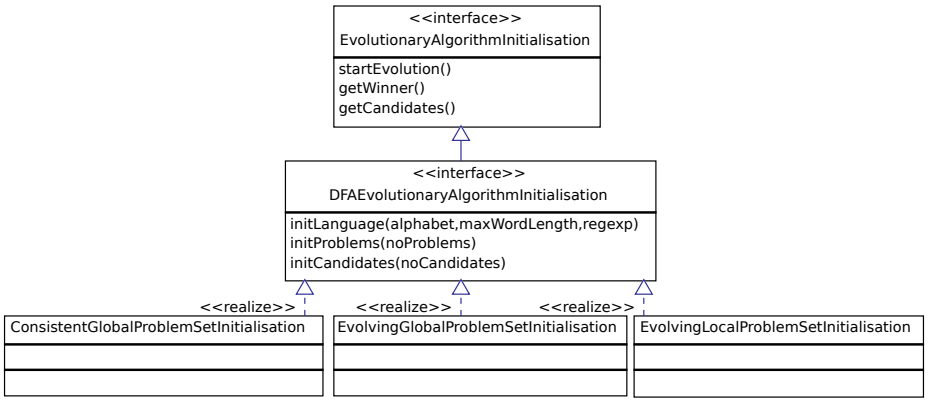

Abbildung 6: EA Initialisierung Klassendiagramm

Initialisierung der Sprache Beim initialisieren der Sprache werden das Alphabet und der reguläre Ausdruck für die spätere Verwendung abgelegt, es wird eine Instanz des WordProblemGenerators angelegt und ein Referenzautomat (dk.brics.automaton) wird erzeugt, welcher später zur Überprüfung von Resultaten verwendet wird.

Generieren von Problemen Probleme werden von einem ProblemGenerator erzeugt. Die für dieses Problem erstellte Implementation WordProblemGenerator generiert mithilfe eines gegebenen Alphabets Tupel vom Typ Tuple<CharArray, Boolean> wobei im CharArray eine zufällige Zeichenfolge der Zeichen aus dem Alphabet mit einer zufälligen länge zwischen 0 und maxWordLength Zeichen ist. Der Boolean zeigt an, ob die generierte Zeichenfolge vom ebenfalls gegebenen regexp akzeptiert wird oder nicht.

Zu dem generieren von einzelnen «Problem»- «Lösung»Tupeln kann der Problemgenerator auch ganze ProblemSets generieren. Als Parameter hierfür benötigt der WordProblemGenerator die gewünschte Länge des Sets und ein Flag welches angibt ob der leere String immer im Set vorhanden sein soll oder nicht.

Generieren von Lösungskandidaten Das Generieren von zufälligen Automaten mit einem gegebenen Alphabet erledigt der Konstruktor der RandomDeterministicFiniteAutomaton Klasse. Eine detaillierte Beschreibung wie die Automaten zufällig zusammengestellt werden befindet sich im Kapitel 3.1.1.

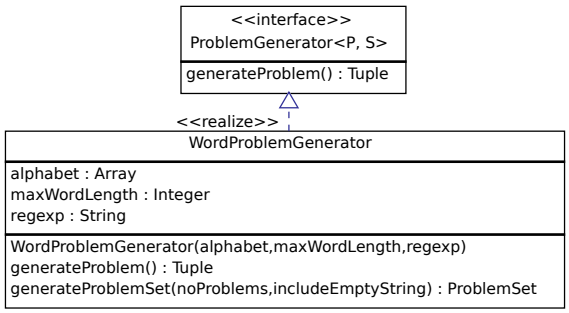

Abbildung 7: Problemgenerierung Klassendiagramm

3.2.2. Feste globale Problemmenge

Als «Grundversion»und Grundlage für die anderen Implementationen wurde zuerst ein evolutionärer Algorithmus mit einer festen, globalen Problemmenge implementiert.

Der Algorithmus an sich funktioniert hierbei grob wie folgt:

1. Initialisierung (Sprache, Problemmenge, Population, Referenzautomat)
2. Berechnen der Fitness aller Individuen
3. Sortieren der Individuen nach Fitness
4. Hat ein Individuum alle Probleme korrekt gelöst? Falls ja, wird er mit dem Referenzautomaten verglichen. Wenn er einer korrekten Lösung entspricht wird der Algorithmus angehalten und die Anzahl benötigter Zyklen wird zurückgegeben.
5. Von den besten 50% der Individuen wird eine Deep Copy erstellt
6. Die Kopien werden zufällig mutiert
7. Die besten 50% und deren mutierte Kopien bilden die neue Population
8. Die Variable zum zählen der Zyklen wird um 1 erhöht.
9. Wenn das Zykluslimit noch nicht überschritten ist, weiter mit 2. ansonsten war der Algorithmus nicht erfolgreich und bricht ab.

3.2.3. Mutierende globale Problemmenge

Die Implementation des Algorithmus mit der mutierenden globalen Problemmenge ist nahezu identisch mit der zuvor beschriebenen.

Sie unterscheidet sich lediglich darin, dass nach dem mutieren der Individuen auch die Problemmenge verändert wird.

1. Initialisierung (Sprache, Problemmenge, Population, Referenzautomat)
2. Berechnen der Fitness aller Individuen
3. Sortieren der Individuen nach Fitness
4. Hat ein Individuum alle Probleme korrekt gelöst? Falls ja, wird er mit dem Referenzautomaten verglichen. Wenn er einer korrekten Lösung entspricht wird der Algorithmus angehalten und die Anzahl benötigter Zyklen wird zurückgegeben.
5. Von den besten 50% der Individuen wird eine Deep Copy erstellt
6. Die Kopien werden zufällig mutiert
7. Die besten 50% und deren mutierte Kopien bilden die neue Population
8. **Die 50% einfachsten Probleme aus der Problemmenge werden gelöscht und durch zufällige, neue ersetzt.**
9. Die Variable zum zählen der Zyklen wird um 1 erhöht.
10. Wenn das Zykluslimit noch nicht überschritten ist, weiter mit 2. ansonsten war der Algorithmus nicht erfolgreich und bricht ab.

3.2.4. Lokale mutierende Problemengen

Der evolutionäre Algorithmus mit den lokalen mutierenden Problemmengen basiert auf einem anderen Prinzip. Hier treten jeweils zwei Kandidaten gegeneinander an und versuchen die Mitgeführten Probleme des Gegenspielers zu lösen. Derjenige der besser abschneidet gewinnt und kommt eine Runde weiter.

1. Initialisierung (Sprache, Population mit einer Problemmenge pro Individuum, Referenzautomat)
2. Durchiterieren der Population in Zweierschritten (i+=2). Selektion des von iten und i+1ten Individuums
3. Berechnung der Fitness des iten Individuums unter Verwendung des Problem Sets des i+1ten Individuums

4. Berechnung der Fitness des i+1-ten Individuums unter Verwendung des Problem Sets des i-ten Individuums
5. Hat ein Individuum alle Probleme korrekt gelöst? Falls ja, wird er mit dem Referenzautomaten verglichen. Wenn er einer korrekten Lösung entspricht wird der Algorithmus angehalten und die Anzahl benötigter Zyklen wird zurückgegeben
6. Ist die Fitness beider Individuen gleich? Falls ja, wird zufällig eines selektiert. Falls nein, wird das Individuum mit der höheren Fitness selektiert
7. Vom selektierten Individuum wird eine Deep Copy erstellt und zufällig mutiert
8. Die 50% einfachsten Probleme der Problemmengen vom selektierten Individuum und der Kopie werden gelöscht und durch zufällige, neue ersetzt
9. Das selektierte Individuum und die mutierte Kopie werden der Population der nächsten Runde hinzugefügt
10. Die Population der nächsten Runde wird durchmischt (Die Reihenfolge der Individuen wird randomisiert)
11. Wenn das Zykluslimit noch nicht überschritten ist, weiter mit 2. ansonsten war der Algorithmus nicht erfolgreich und bricht ab

3.3. Messen und Auswerten

Bei ersten Versuchsreihen hat sich gezeigt, dass die evolutionären Algorithmen nicht in jedem Falle in einer annehmbaren Zeit konvergieren. Abhängig von der Grösse der Population und der Grösse der Problemmenge und des betrachteten Problems wurden Unterschiede im Konvergenzverhalten festgestellt.

Zum Vergleichen der verschiedenen Algorithmen wurden verschiedene Werte für die Grössen der Problemmenge und der Population gewählt. Die Algorithmen werden mit diesen, wechselnden, Eingabeparametern laufen gelassen und es wird gemessen wie oft die Algorithmen in annehmbarer Zeit eine Lösung finden. Dieser Wert wird dann auf eine Zahl zwischen 0 und 100 herunter skaliert um vergleichbare Werte zu erhalten. Zum Schluss werden diese Werte visualisiert.

Dieser Ablauf wurde in zwei Schritte unterteilt. In einem ersten Schritt laufen Java-Konsolenanwendungen und versuchen gegebene Probleme mit verschiedenen Eingabeparametern zu lösen. Sie loggen diese Ergebnisse in Logfiles mit folgendem Format:

```
1 Mon Dec 30 06:05:10 CET 2013: Problem Count: 150
2 Mon Dec 30 06:05:10 CET 2013: CandidatesCount: 250
3 Mon Dec 30 06:05:10 CET 2013: Max Cycles: 500
4 Mon Dec 30 06:05:20 CET 2013: 0: finished (9245ms, 500cycles)
5 Mon Dec 30 06:05:20 CET 2013: 0: No solution found.
```

```
...
Mon Dec 30 06:14:53 CET 2013: 76: finished (812ms, 43cycles)
Mon Dec 30 06:14:53 CET 2013: 76: Solution found.
...
Mon Dec 30 06:17:34 CET 2013: 99: finished (12438ms, 500cycles)
Mon Dec 30 06:17:34 CET 2013: 99: No solution found.
Mon Dec 30 06:17:34 CET 2013: Solution Found: 37
Mon Dec 30 06:17:34 CET 2013: Avg cycles: 71
Mon Dec 30 06:17:34 CET 2013: Max cycles: 261
Mon Dec 30 06:17:34 CET 2013: Min cycles: 14
Mon Dec 30 06:17:34 CET 2013: No solution found: 63
Mon Dec 30 06:17:34 CET 2013: logging finished. (runtime: 744178ms)
```

Listing 4: Log Format

In diesem Beispiel wurde ein Algorithmus mit einer 250 Problemen, 150 Individuen und einem Zykluslimit von 500 laufen gelassen. In 37 von 100 Fällen wurde eine Lösung gefunden. Bei den erfolgreichen Versuchen wurde durchschnittlich nach 71 Zyklen die Lösung entdeckt. Im schnellsten Fall war die Lösung bereits nach 14 Zyklen gefunden, im langsamsten Fall dauerte die Lösungsfindung 261 Zyklen.

Die Java-Konsolenanwendungen sind im Package ch.zhaw.regularLanguages.evolution.runners implementiert.

Im zweiten Schritt werden diese Logfiles von einem Python Script ausgelesen, Logfiles mit dem selben Präfix und der selben Problem- und Populationsgrössen werden zusammengefasst. Am Schluss werden die Ergebnisse mithilfe der Python Matplotlib [4] geplottet. Das Python Script ist unter doc/ statistics /log_analysis.py hinterlegt.

Um Vergleichswerte zu haben, wurde ein Zufallsalgorithmus implementiert, welcher versucht die Probleme durch einfaches ausprobieren zufälliger Automaten zu lösen. Die Erfolgsquote dieses Algorithmus wurde in die selbe Form gebracht wie diejenige der anderen.

4. Ergebnisse & Analyse

Um überhaupt etwas analysieren zu können mussten zuerst passende Probleme gefunden werden. Am Kickoff Meeting wurden folgende, zu untersuchende Probleme definiert:

- $\Sigma = \{'a', 'b'\}$, RE:[ab]∗abab; Die Sprache aller Wörter die aus a und b bestehen und auf *abab* enden. Folgend kurz «Das abab Problem»genannt
- $\Sigma = \{'0', '1'\}$, RE:(1(01∗0)∗1|0)∗; Die Sprache aller durch drei teilbaren Binärzahlen. Folgend kurz «Das durch drei teilbar Problem»
- $\Sigma = \{'0', '1'\}$, RE:(0|101|11(01)∗(1|00)1|(100|11(01)∗(1|00)0)(1|0(01)∗(1|00)0)∗0(01)∗(1|00)1)∗; Die Sprache aller durch fünf teilbaren Binärzahlen. Folgend kurz «Das durch fünf teilbar Problem »

Um sich einen ersten Überblick zu verschaffen wurden bei jeder Sprache als erstes folgende Eingabeparameter ausprobiert:

- Problemmengengrösse: 50, 100, 150, 200, 250
- Populationsgrösse: 50, 100, 150, 200, 250

Nach diesem ersten Screening wurde klar, dass nicht alle Probleme im gleichen Bereich interessant sind. So konvergiert das durch drei teilbar Problem zum Beispiel bereits bei Problemmengengrössen von 50 fast immer. Entsprechend wurden für weitere Analysen problemspezifische Eingabeparameter verwendet.

Die Untersuchung der verschiedenen Algorithmen mit den verschiedenen Problemstellungen hat, unabhängig vom jeweiligen Problem, durchgängig vergleichbare Resultate hervorgebracht. Bei der folgenden Analyse der Algorithmen werden jeweils exemplarische Untersuchungsresultate gezeigt, welche aufgrund ihres aussagekräftigen Charakters ausgewählt wurden. Die Logfiles und die entsprechenden Auswertungen aller herbeigezogenen Untersuchungen finden Sie im Dateianhang im Ordner doc/ statistics .

4.1. Konstante globale Problemmenge

In einem ersten Schritt wurde versucht alle Probleme mithilfe einer konstanten, globalen Problemmenge zu lösen. Dabei fiel auf, dass dieser Algorithmus, unabhängig vom Problem, mit sowohl der Grösse der Problemmenge als auch der Populationsgrösse skaliert. Ein gutes Beispiel hierfür ist Abbildung 8. Auf der X-Achse sind sowohl die Grösse

der Problemmenge (z.B. $P : 50$) als auch die Populationsgrösse (z.B. $C : 100$) abgebildet. Auf der Y-Achse sieht man die Performance des Algorithmus bei den jeweiligen Eingabeparametern.

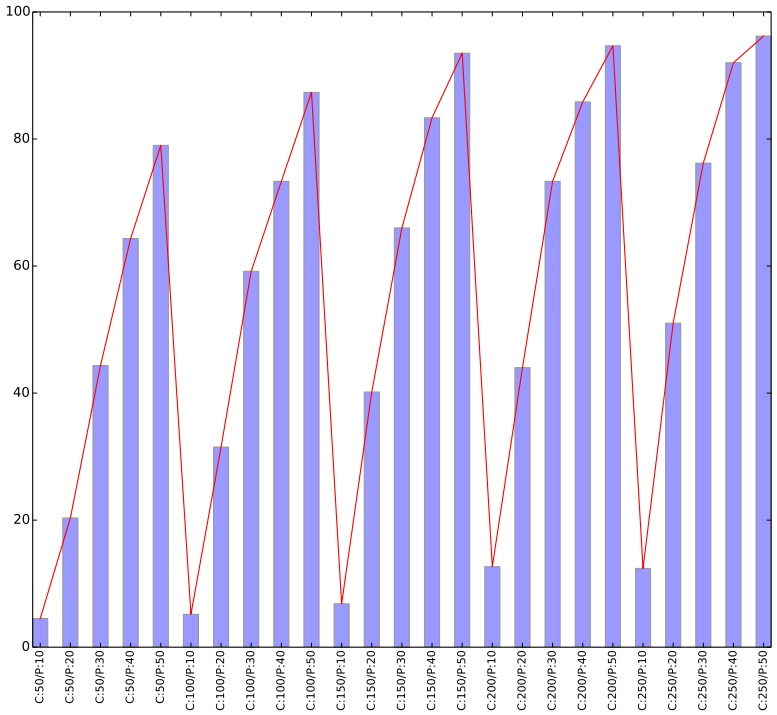

Abbildung 8: Durch fünf teilbar Problem - KG Algorithmus

Die Eingabeparameter für die Abbildung 8 waren:

- Problemmengengrösse: 10, 20, 30, 40, 50
- Populationsgrösse: 50, 100, 150, 250

In dieser Abbildung zeichnet sich ein «Haifischzahn förmiges»Muster ab, was darauf schliessen lässt, dass dieser Algorithmus mit zu kleinen Problemmengen sehr schlecht konvergiert. Dies kann auch nur teilweise durch die Verwendung von grösseren Populationen wettgemacht werden. Um dies weiter zu untersuchen wurden bei dieser Aufgabenstellung («durch fünf Teilbar Problem») Tests bei konstanter Populationsgrösse durchgeführt.

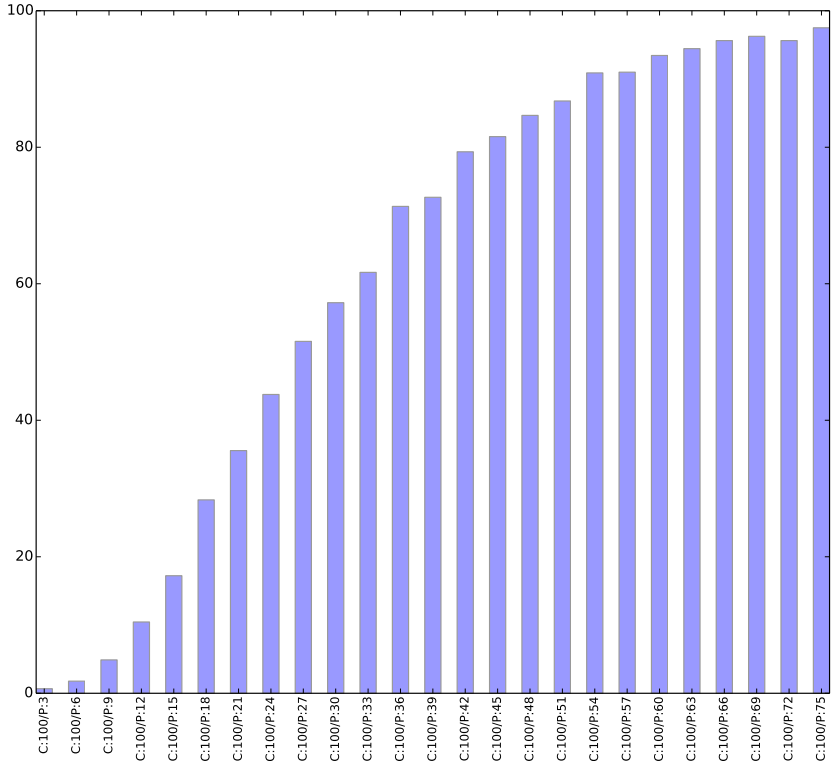

Abbildung 9: GK Algorithmus - Skalierung mit Problemmengengrösse

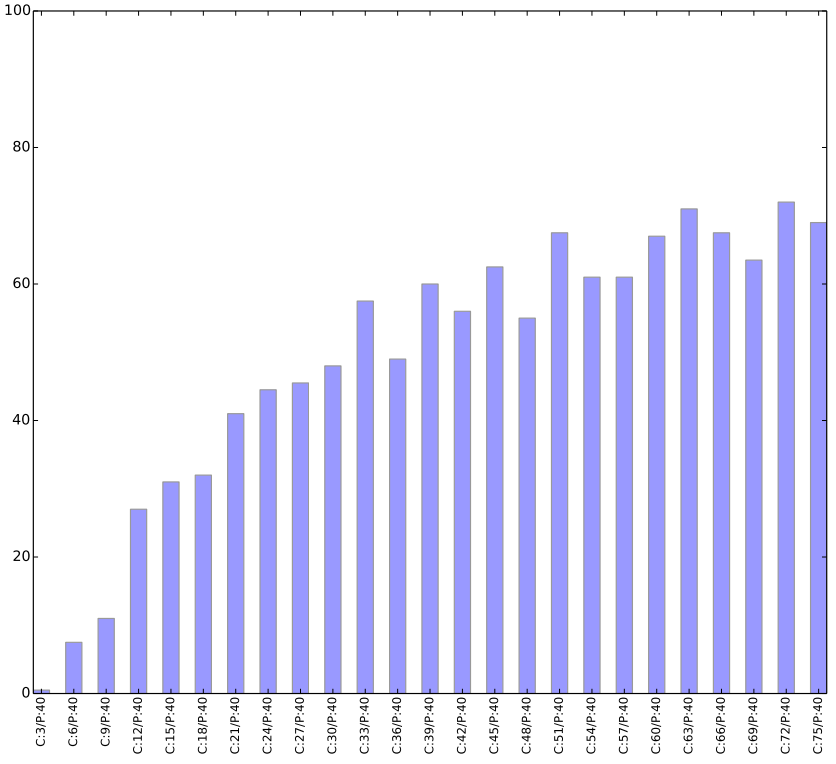

Abbildung 10: GK Algorithmus - Skalierung mit Populationsgrösse

Die Eingabeparameter für die Abbildung 9 waren:

- Problemmengengrösse: 3, 6, 9, 12, ..., 72, 75
- Populationsgrösse: 100

In Abbildung 9 sieht man wie eine zu kleine Problemmenge den Erfolg des Algorithmus einschränken kann. Jedoch flacht die Kurve ab circa 40 Problemen langsam ab und ab circa 65 Problemen scheint eine Vergrösserung der Problemmenge zu keiner signifikanten Verbesserung des Ergebnisses mehr zu führen.

Um zu messen wie der Algorithmus mit der Populationsgrösse skaliert, wurde eine Testreihe mit fixer Anzahl Probleme und variabler Populationsgrösse implementiert.

Die Eingabeparameter für die Abbildung 10 waren:

- Problemmengengrösse: 40
- Populationsgrösse: 3, 6, 9, 12, ..., 72, 75

Hier wurde bewusst eine zu kleine Problemmenge gewählt um nicht an die 100% Marke zu kommen. Man sieht hier wiederum, dass der Algorithmus mit sehr kleinen Mengen (3, 6, 9 Individuen) praktisch gar nicht konvergiert. Die Kurve steigt danach steil an und flacht ab einer Populationsgrösse von circa 35 ab, steigt jedoch weiter.

4.2. Mutierende globale Problemmenge

Die untersuchten evolutionären Algorithmen mit einer globalen mutierenden Problemmenge haben im Gegensatz zum Ansatz mit den Festen globalen Problemmengen ein abweichendes Konvergenzverhalten im Bezug auf die grösse der Problemmenge an den Tag gelegt.

Die Eingabeparameter für die Abbildung 11 waren:

- Problemmengengrösse: 50, 100, 150, 200, 250
- Populationsgrösse: 50, 100, 150, 200, 250

In Abbildung 11 ist ersichtlich wie sich ein «teppenförmiges»Muster abzeichnet. Daraus erschliesst sich die These, dass der Algorithmus unabhängig von der Problemmengengrösse eigentlich immer gleich oft konvergiert. Um diese These zu überprüfen wurde auch mit diesem Algorithmus ein Test bei konstanter Populationsgrösse durchgeführt. Als Problem der Wahl wurde hier aus Gründen der Vergleichbarkeit wieder das durch fünf Teilbar Problem gewählt.

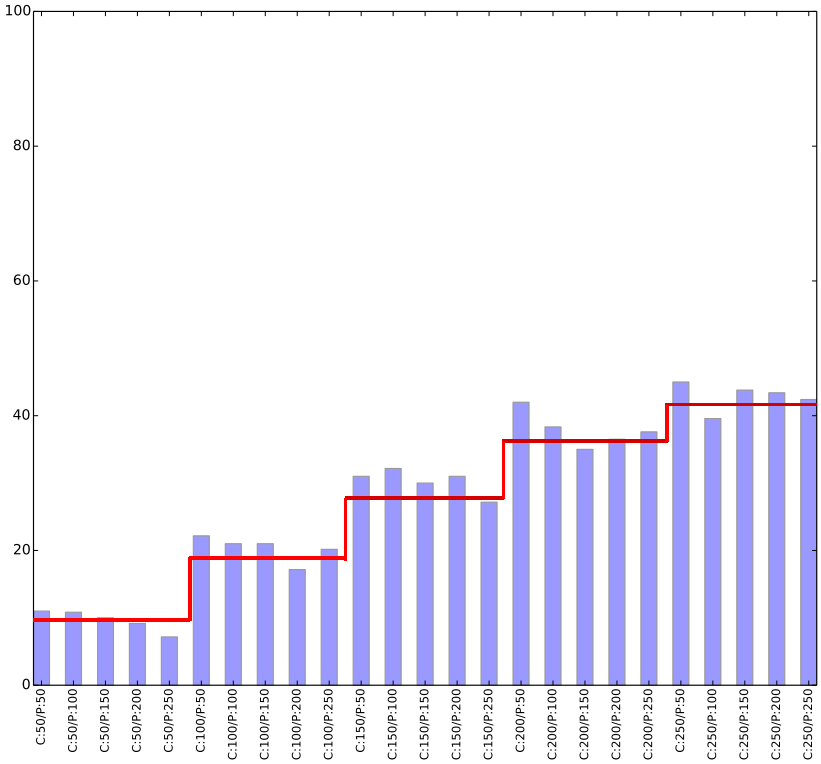

Abbildung 11: ABAB Problem - Globale Mutierende Problemmenge

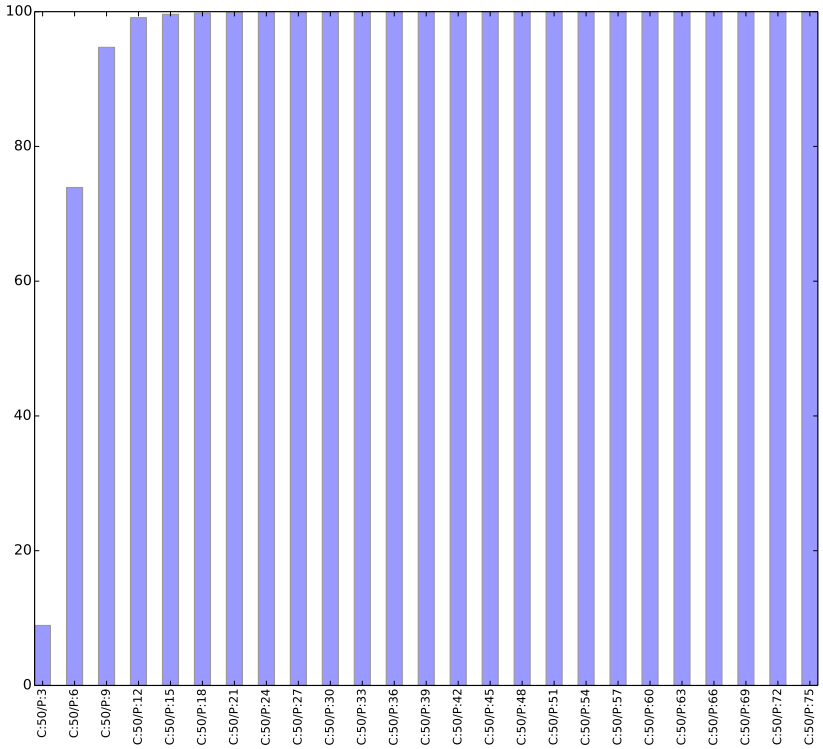

Abbildung 12: GM Algorithmus - Skalierung mit Problemmengengrösse

Die Eingabeparameter für die Abbildung 12 waren:

- Problemmengengrösse: 3, 6, 9, 12, ..., 72, 75
- Populationsgrösse: 50

Bei diesem Versuch (Abbildung 12) sieht man, dass auch dieser Algorithmus mit sehr kleinen Problemmengen (3 oder 6 Probleme) mühe hat. Jedoch konvergiert er bereits ab circa 10 Problemen in jedem Fall. Das heisst, sobald eine kritische Problemmengengrösse erreicht wurde, hat ein weiteres erhöhen dieser keinen Einfluss mehr auf das Ergebnis.

Das bedeutet, dass diese Art von evolutionären Algorithmen primär mit der Populationsgrösse skalieren. Welches die Optimale Problemmengengrösse für ein konkretes Problem ist, muss individuell geprüft werden. Für unser Problem der durch fünf teilbaren Binärzahlen konvergiert der Algorithmus zum Beispiel bereits ab 20 Problemen sehr zuverlässig.

Um zu prüfen wie das Konvergenzverhalten in Abhängigkeit der Populationsgrösse ist, wurde auch hierzu entsprechende Daten gesammelt.

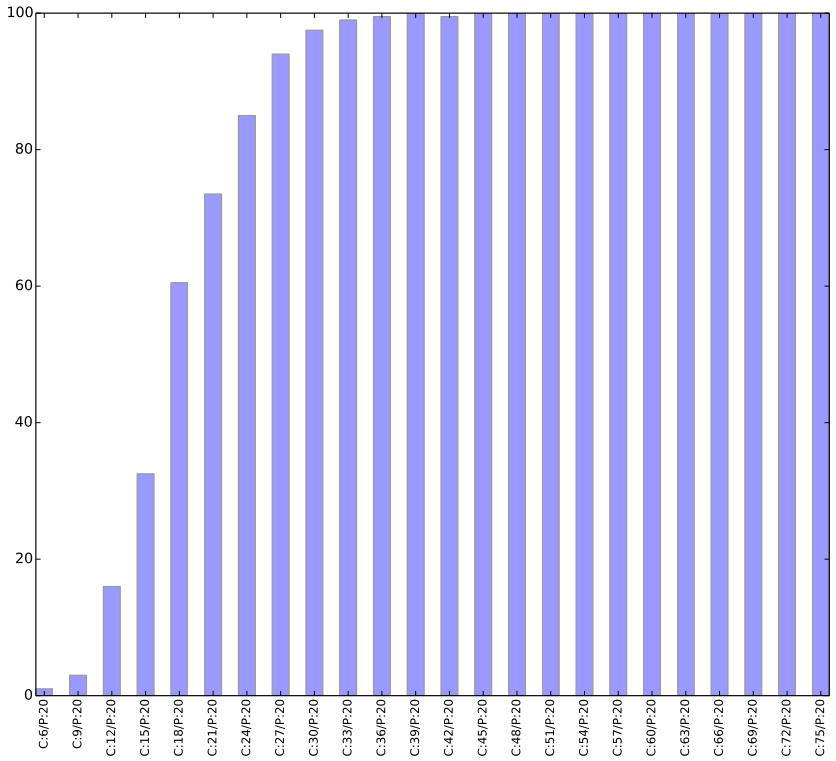

Abbildung 13: GM Algorithmus - Skalierung mit Populationsgrösse

Die Eingabeparameter für die Abbildung 13 waren:

- Problemmengengrösse: 20
- Populationsgrösse: 3, 6, 9, 12, ..., 72, 75

In dieser Abbildung sieht man erneut, dass eine zu kleine Population ein Problem ist. Der steile Anstieg der Konvergenzrate und das Abflachen (in diesem Fall sogar erreichen der 100% Marke) ist vergleichbar mit dem Verhalten des Algorithmus mit konstanter Problemmenge.

4.3. Mutierende lokale Problemmengen

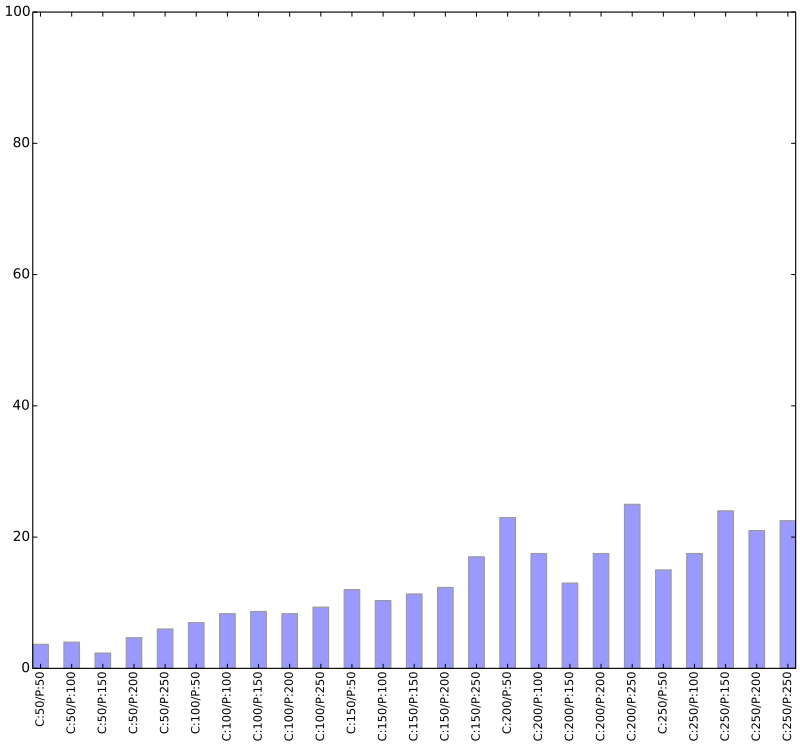

Abbildung 14: ABAB Problem - Lokale mutierende Problemmenge

Die Untersuchungen zum Ansatz mit den lokalen Problemmengen, waren ernüchternd. Das Konvergenzverhalten bezüglich Problemmengen- und Populationsgrösse war von der Form her identisch mit demjenigen des Algorithmus mit einer globalen mutierenden Problemmenge. Die Ergebnisse unterschieden sich jedoch dadurch, dass diejenigen mit den

lokalen Problemmengen bei gleichen Eingabeparametern schlechter und unregelmässiger waren. Dazu kommt, dass der Rechenaufwand durch den Aufbau des Algorithmus massiv grösser ist als bei den beiden zuvor untersuchten.

Die Eingabeparameter für die Abbildung 14 waren:

- Problemmengengrösse: 50, 100, 150, 200, 250
- Populationsgrösse: 50, 100, 150, 200, 250

Man sieht auch in dieser Grafik erneut eine «Treppenform»auch wenn diese weniger ausgeprägt ist als in Abbildung 11. Weiter fiel auf, dass dieser Algorithmus - wenn er konvergierte - dazu länger brauchte als die Ansätze mit globalen Problemmengen. Länger heisst in diesem Falle, sowohl mehr Zyklen als auch mehr Rechenzeit.

4.4. Zufälliges generieren von Automaten

Um Vergleichswerte zu haben, wurde ein Zufallsalgorithmus implementiert, welcher so oft neue Automaten generiert bis einer zufällig der geforderten Lösung entspricht. Der Aufbau des Algorithmus und die Messmethode ist dabei gleich wie bei den zuvor gezeigten. Das heisst, es gibt eine Population welche nach jedem Zyklus neu generiert wird. Dies wird so oft wiederholt, bis ein Zykluslimit erreicht wurde. Gemessen wird wie oft (in Prozent) der Algorithmus eine Lösung findet. In Abbildung 15 ist ersichtlich, dass dieser Algorithmus bedeutend weniger oft konvergiert als alle von uns untersuchten.

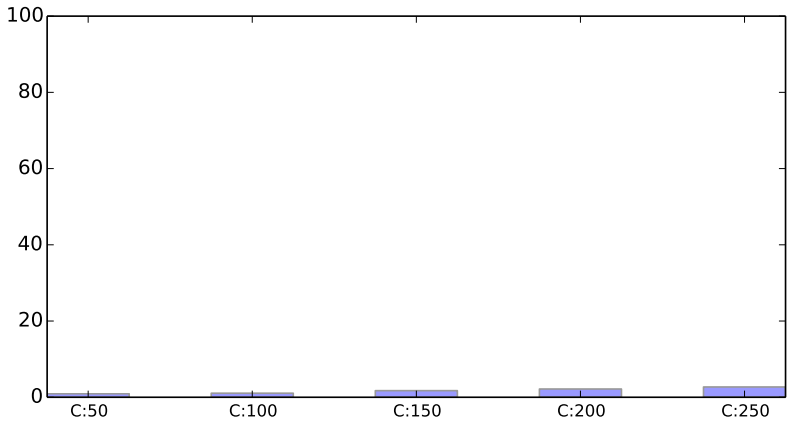

Abbildung 15: Random Algorithmus ABAB Problem

4.5. Schlussfolgerungen

Die Untersuchungen haben aufgezeigt, dass die Klasse der evolutionären Algorithmen zum finden von endlichen Automaten zu gegebenen regulären Ausdrücken folgende Eigenschaften aufweist:

- Verfahren mit einer konstanten Problemmenge skalieren stark mit der Grösse ebendieser
- Alle Verfahren skalieren mit der Populationsgrösse
- Globale Selektionsverfahren konvergieren schneller und zuverlässiger als lokale

4.5.1. Skalierung mit der Problemmengengrösse

Die Skalierung mit der Problemmengengrösse erschloss sich mir nach einem Brainstorming zum Thema Mengenlehre.

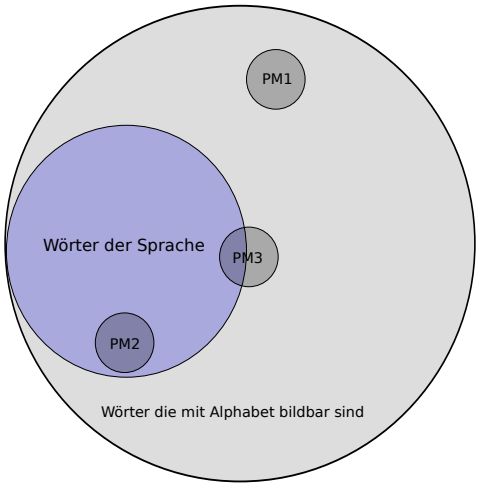

Abbildung 16: Problemmengen

In Abbildung 16 sind die (unendliche) Menge aller bildbaren Wörter mit dem gegebenen Alphabet, die (bei unseren Beispielen ebenfalls unendliche) Menge aller Wörter der entsprechenden Sprache (bei uns repräsentiert durch reguläre Ausdrücke und endliche Automaten) und mögliche Problemmengen (PM1 bis PM3) abgebildet.

- PM1: Diese Problemmenge beinhaltet kein einziges Wort der Sprache. Damit lässt sich nicht testen ob ein Automat Wörter der Sprache erkennt.
- PM2: Diese Problemmenge beinhaltet nur Wörter der Sprache. Damit lässt sich nicht testen ob ein Automat Wörter die nicht in der Sprache sind nicht akzeptiert.

- PM3: Diese Problemmenge beinhaltet sowohl Wörter der Sprache als auch solche die nicht zur Sprache gehören. Eine solche Problemmenge wäre die Voraussetzung wenn ein Algorithmus konvergieren soll.

Die Wahrscheinlichkeit eine Problemmenge mit den Eigenschaften von PM3 zu erhalten hängt sowohl von der Menge der generierten Probleme, als auch vom Verhältnis der Menge aller bildbaren Wörter zur Menge aller Wörter der Sprache ab. Ersteres erklärt warum mutierenden Problemmengen mit kleineren Problemmengen auskommen. Sie generieren nach jedem Zyklus $\frac{p}{2}$ (wenn p die Grösse der Problemmenge ist) neue Probleme und generieren so in ihrer Laufzeit viel mehr Probleme als ihre konstanten Gegenstücke. Letzteres erklärt warum die Algorithmen mit den unterschiedlichen Problemstellungen unterschiedlich oft konvergieren. Bei den beiden Binärzahlen-Teilbarkeits-Problemen ist das Verhältnis aller Binärwörter zum Verhältnis der Wörter der Sprache 1:3 bzw. 1:5. Beim ABAB Problem ist dieses Verhältnis bedeutend kleiner (1:16).

4.5.2. Skalierung mit der Anzahl Lösungskandidaten

Die Skalierung mit der Populationsgrösse lässt sich dadurch erklären, dass der Algorithmus nur konvergieren kann wenn ein genügend grosses Gefälle in der Fitnessverteilung vorhanden ist.

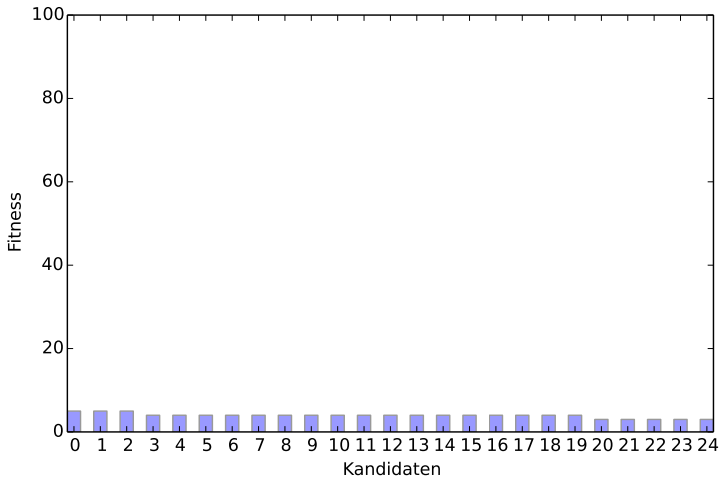

Abbildung 17: Zu geringes Gefälle in der Fitnessverteilung

In der Abbildung 17 ist ersichtlich, dass in diesem Beispiel alle Individuen schlecht konditioniert sind und dass praktisch kein Gefälle vorhanden ist. Dies kann verschiedene Ursachen haben:

- Es wurden zufällig sehr ähnliche oder ähnlich schlecht konditionierte Automaten generiert
- Die verwendete Problemmenge beinhaltet nur oder gar keine Wörter der Sprache
- Die verwendete Problemmenge ist zu klein

Die Wahrscheinlichkeit, dass Ersteres eintritt wird durch das verwenden von grösseren Populationen vermindert. Die anderen beiden Faktoren müssen über die Problemmenge abgefangen werden. Das Verwenden von grösseren Populationen führt auch dazu, dass die Wahrscheinlichkeit bereits bei der Initialisierung einen oder mehrere Automaten zu generieren die sehr nahe an der Lösung sind steigt. Die Kehrseite des Ganzen ist, dass der Rechenaufwand linear mit der Populationsgrösse zunimmt.

4.5.3. Lokale vs. globale Selektionsverfahren

Warum lokale Selektionsverfahren weniger schnell konvergieren als globale zeigte sich nach dem plotten von Fitnesswerten und dem entsprechenden Evolutionsverhalten.

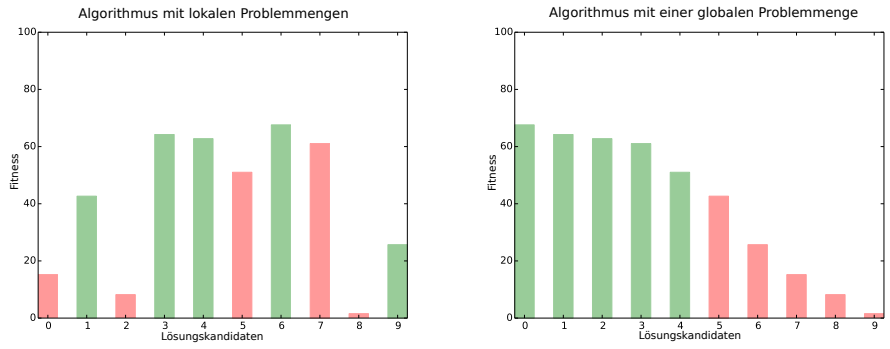

Abbildung 18: Evolutionsverhalten lokale Problemenge vs. globale Problemmenge

In der Abbildung 18 sind links und rechts die selben Individuen abgebildet. Grün eingefärbt sind diejenigen, welche die nächste Runde erreichen, die roten sterben aus. In der linken Grafik sterben die Individuen 5 und 7 aus obwohl sie besser konditioniert sind als die Individuen 1 und 9. Dies ist so, weil für dieses Verfahren ein «1 versus 1»Selektionsverfahren implementiert wurde (Siehe Kapitel 3.2.4). Das heisst, dass hier jeweils die Individuen 0 und 1, 2 und 3, 4 und 5 etc. gegeneinander antreten und das jeweils bessere weiter kommt. Wie gut die Automaten im Vergleich zum grossen Ganzen sind ist irrelevant.

5. Projektfazit

Trotz anfänglicher Bedenken aufgrund der Tatsache, dass sich die Funktionsweise eines endliche Automaten selbst bei kleinsten Veränderungen komplett ändern kann, ist es mithilfe von evolutionären Algorithmen gelungen solche Automaten gezielt und mit endlichem Rechenaufwand zu generieren. Auch konnten Unterschiede im Konvergenzverhalten der verschiedenen Algorithmen, sowohl bei unterschiedlichen Problemstellungen, als auch bei den verschiedenen Ansätzen zum Umgang mit Problemmengen festgestellt, analysiert und ergründet werden. Es wurde festgestellt, dass der Rechenaufwand beim finden von endlichen Automaten zu gegebenen regulären Ausdrücken mithilfe eines evolutionären Algorithmus mit einer einer globalen, mutierenden Problemmenge geringer ist als bei den beiden anderen untersuchten Ansätzen. Ob und falls ja, wieweit dieses Ergebnis generalisiert werden kann war nicht Ziel dieser Arbeit und müsste in weiteren Studien untersucht werden. Ich bedanke mich herzlich bei Dandolo Flumini. Er hat mich mit guten Ideen beliefert und so zum Erfolg dieses Projektes beigetragen.

A. Automaten zu den gegebenen Problemen

A.1. Das abab Problem

RegExp: [ab]∗abab

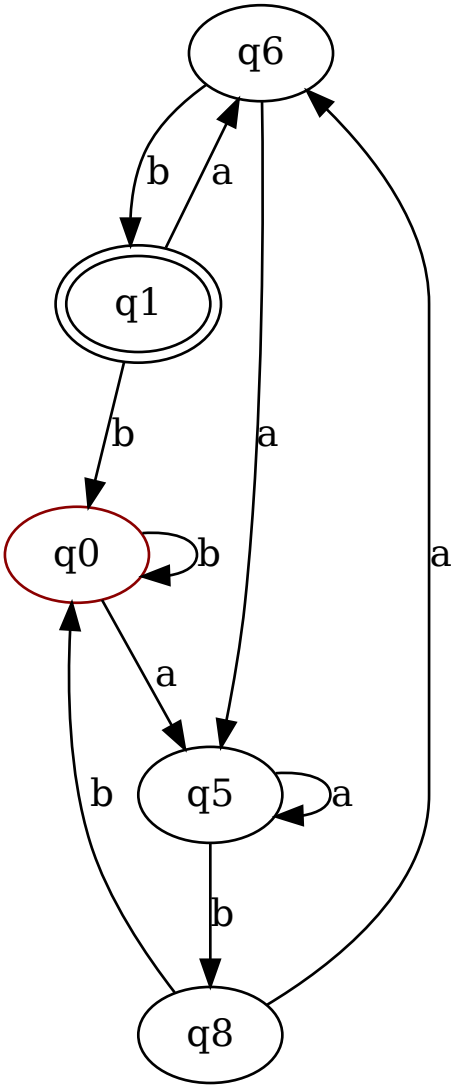

Abbildung 19: «abab»Automat

A.2. Das durch drei teilbar Problem

RegExp: $(1(01*0)*1|0)*$

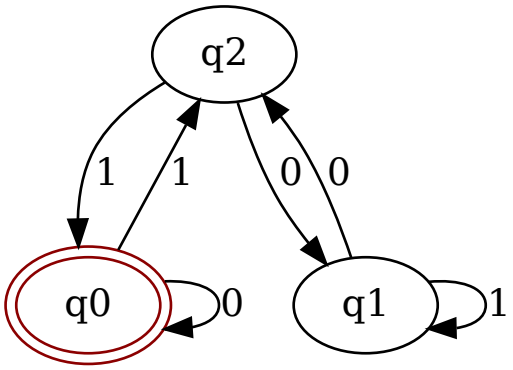

Abbildung 20: «Durch drei teilbar»Automat

A.3. Das durch fünf teilbar Problem

RegExp: $(0|101|11(01)*(1|00)1|(100|11(01)*(1|00)0)(1|0(01)*(1|00)0)*0(01)*(1|00)1)*$

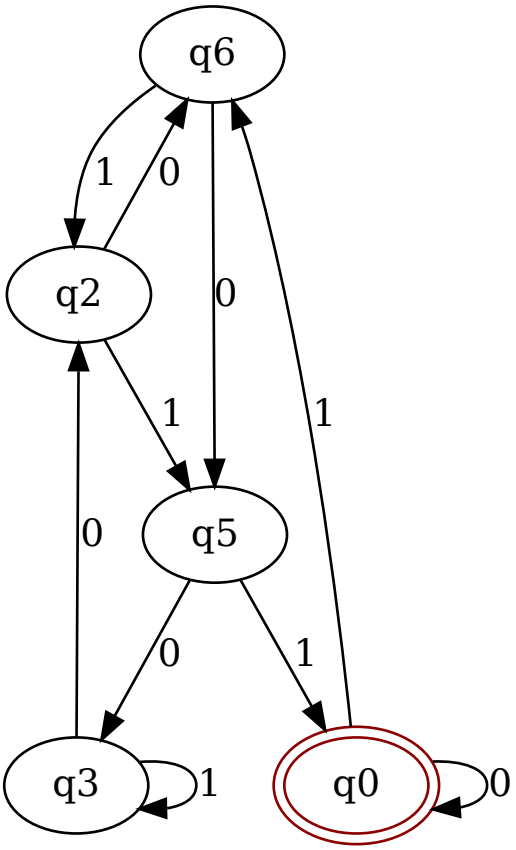

Abbildung 21: «Durch fünf teilbar»Automat

B. Protokoll Kick-Off Meeting vom 27.12.2014

Steht die Auftraggeberin bzw. der Auftraggeber hinter dieser Semesterarbeit?
Ja

Sind die fachliche Kompetenz und die Verfügbarkeit der Betreuungsperson sichergestellt
Ja

Sind die Urheberrechte und Publikationsrechte geklärt?
Ja

Bekommt die Studentin oder der Student die notwendige logistische und beratende Unterstützung durch die Auftraggeberin bzw. den Auftraggeber?
Es exitistert kein externer Auftraggeber

Entsprechen Thema und Aufgabenstellungen den Anforderungen an eine Semesterarbeit?
Ja

Ist die Arbeit thematisch klar abgegrenzt und terminlich entkoppelt von den Prozessen (des Unternehmens) der Auftraggeberin bzw. des Auftraggebers?
Ja

Ist eine Grobplanung vorhanden? Sind die nächsten Schritte klar formuliert (von der Studentin oder dem Studenten)?
Nächste Schritte:

- Implementation von endlichen Automaten in Java
- Implementation von Automaten-Mutationen in Java
- Implementation grafischen Darstellung von endlichen Automaten
- Implementation der evolutionären Algorithmen
- Testen der Konvergenzverhaltne der verschiedenen implementationen

Dazu parallel: Dokumentieren des erarbeiteten Wissens / erstellen des technischen Berichts.

Ist die Arbeit technisch und terminlich von der Studentin oder dem Studenten umsetzbar?
Ja

C. Protokoll Design Review Meeting vom 14.05.2014

Ist der Auftrag für die Semesterarbeit von der Studentin oder dem Studenten korrekt erfasst?
Ja

Sind die Ausgangslage und das Umfeld ausreichend analysiert, berücksichtigt und bearbeitet?
Ja

Wurde eine systematische Recherche durchgeführt?
Ja

Wurde ein klares Konzept erarbeitet und klar dargestellt?
Ja

Wurden alternative Lösungen betrachtet?
n/a

Entsprechen das Arbeits- und Lösungskonzept den Anfoderungen an eine Semesterarbeit?
Ja

Ist die Lösung grundsätzlich für die Auftraggeberin bzw. den Auftraggeber akzeptabel?
Ja

Ist das Konzept bzw. die Lösung technisch und terminlich im Rahmen der verbleibenden Zeit umsetzbar?
Ja

Sind die nächsten Arbeitsschritte klar formuliert?
Nächste Schritte:

- Implementierung einer zufälligen Suche zum Vergleich
- Durch 5 Teilbar Regexp implementieren
- Analyse & Interpretation dokumentieren
- Fazit schreiben
- Quelle [1] lesen

Glossar

Deep Copy Eine Kopie eines Objektes bei welchem auch alle enthaltenen Objekte kopiert wurden. 18–20

GK Algorithmus Ein evolutionärer Algorithmus mit einer globalen, konstanten Problemmenge. 24, 39

GM Algorithmus Ein evolutionärer Algorithmus mit einer globalen, mutierenden Problemmenge. 26, 27, 39

Individuum ist Teil der Population. Entspricht in unserem Falle einem endlichen Automaten oder einem endlichen Automaten mit Problemmenge. 6

Population ist die Menge aller Individuen in einem evolutionären Algorithmus. 6

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literatur

[1] Stefan Droste. *Zu Analyse und Entwurf evolutionarer Algorithmen.* Universität Dortmund, Fachbereich Informatik, 2000.

[2] Emden Gansner, Eleftherios Koutsofios, and Stephen North. *Drawing graphs with dot.* 2010.

[3] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* Pearson Education, 2002.

[4] John Hunter. Matplotlib. `http://matplotlib.org/`.

[5] Anders Møller. dk.brics.automaton. `http://www.brics.dk/automaton/`, 2011.

[6] AT&T Labs Research and Contributors. Graphviz - graph visualization software. `http://www.graphviz.org/`, before 2000.

[7] Laszlo Szathmary. Graphviz java api. `http://www.loria.fr/~szathmar/off/projects/java/GraphVizAPI/index.php`, 2003.