Semesterarbeit

Evaluation von verschiedenen Ansätzen für Evolutionäre Algorithmen

Adrian Schmid

Zürich, 01.01.2014

Dank

Ich danke Bla und Bli

Abstract

This is rather abstract

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 Menger aller Sprachen, welche durch einen endlichen Automaten beschrieben werden können. Alle Sprachen dieser Klasse haben auch die Eigenschaft, dass Sie durch einen regulären Ausdruck dargestellt werden können. 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.

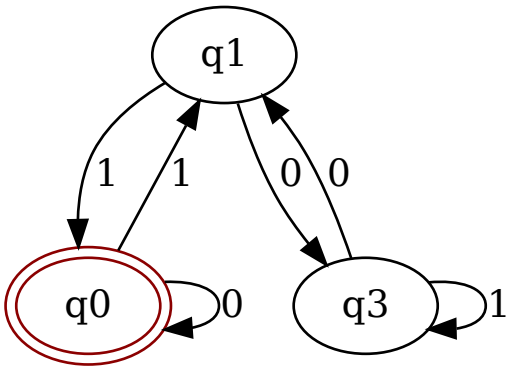

Abbildung 1: Beispiel: Endlicher Automat

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 bzgl. der benätigten 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 Zielfunktions wert 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]

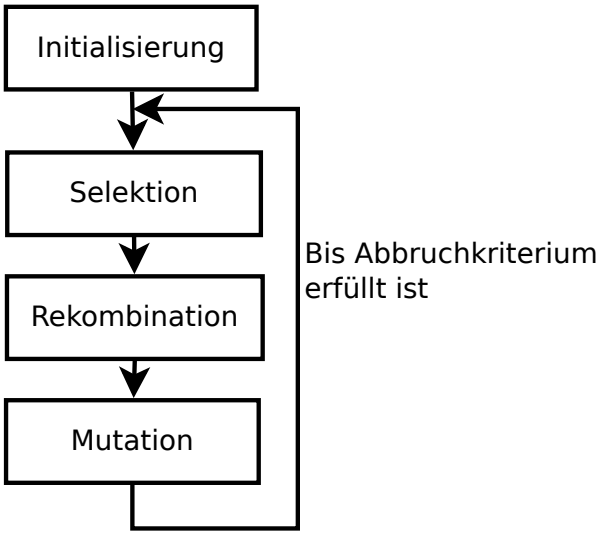

Abbildung 2: Evolutionärer Algorithmus

2.2.1 Anwendung auf unser Problem

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

Initialisierung In der Initialisierungsphase werden wir eine Population von Automaten zufällig erzeugen. Das heisst wir erzeugen eine Zufällige Anzahl von Zuständen, verbinden diese Zufällig und setzen einen zufälligen Start und zufällige Akzeptierende Zustände.

Selektion Es werden jeweils diejenigen Automaten selektiert welche eine höhere Fitness aufweisen. Es werden verschiedene Ansätze zur Selektion implementiert und es wird deren Konvergenzverhalten analysiert.

Fitness Die Fitness von Automaten bestimmen wir in dem wir ihn mit Wörtern füttern und die Ausgabe mit der soll Ausgabe vergleichen. Die Soll Ausgabe können wir anhand des gegebenen regulären Ausdrucks einfach bestimmen. Der Wert der Fitnessfunktion entspricht dann der Summe der richtig akzeptierten und richtig nicht akzeptierten Wörtern aus unserer *Problemmenge*.

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 unter anderem Verschiedene Ansätze zur Erzeugung und zum Verhalten solcher Problemmengen ausprobiert. Dabei wird das Konvergenzverhalten des evolutionären Algorithmus untersucht.

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. Herausgekommen ist dabei folgende Grundimplementation:

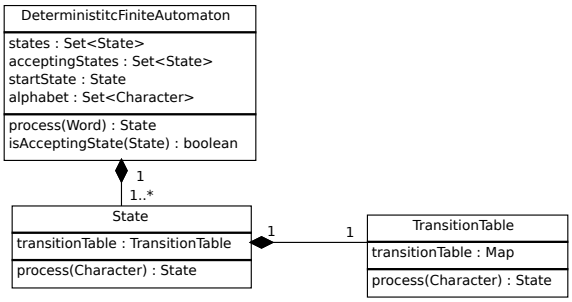

Abbildung 3: DFA Klassendiagramm vereinfacht

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 könnte man nun feststellen 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 [5] mit dem Graphviz Java API [6] 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 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.[4]

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 Um die konsistent der Automaten zu erhöhen, wurde eine Methode zum entfernen aller nicht erreichbarer 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 $q4$.

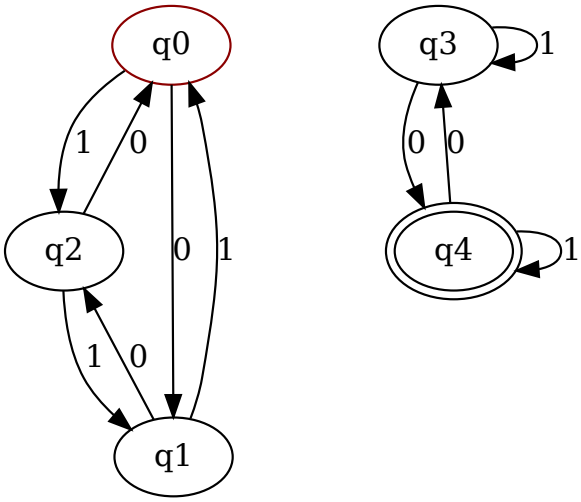

Abbildung 4: Aufgesplitteter Automat

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 nicht beibehalten. 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 ein erweiterter DFA welcher 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: 1 - ($2 \cdot AnzahlZeichen \cdot Komplexitaet$) 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 (1 - $\frac{AnzahlZustaende}{5}$) 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 erfoglreich 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:

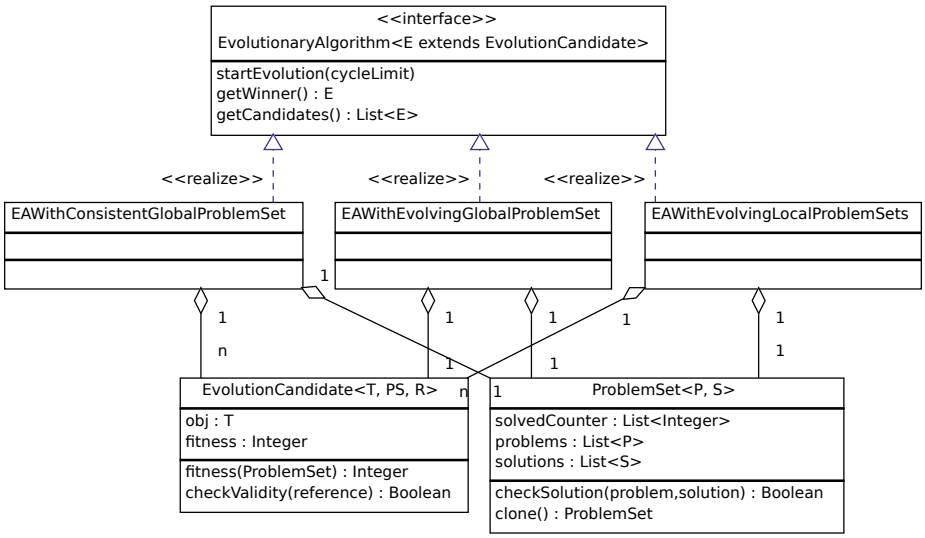

Abbildung 5: EA Grundstruktur vereinfacht

Die Klassen welche die Algorithmen beinhalten und verwenden eine Menge von EvolutionCandidate und ein ProblemSet. Der evolutionäre Algorithmus kann jeweils mit startEvolution gestartet werden und wird abgebrochen sobald eine gülige 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 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. Lösungskandidaten initialisieren (initCandidates(noCandidates))

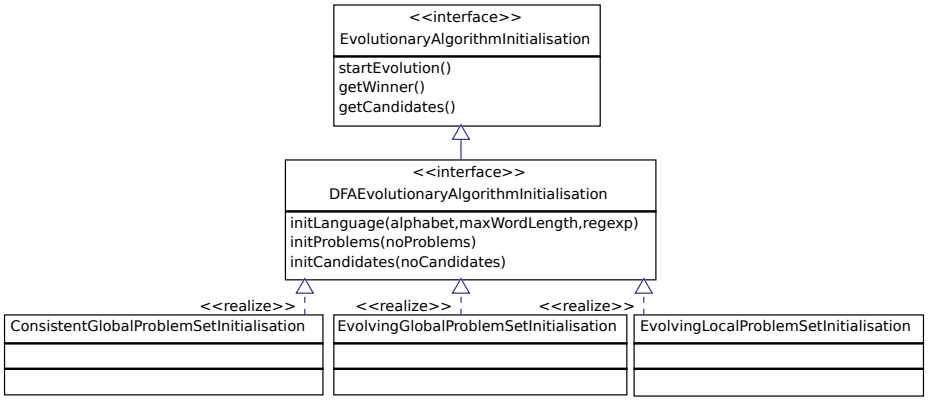

Abbildung 6: EA Initialisierung Klassendiagramm

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 unser 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 detailierte Beschreibung wie die Automaten zufällig zusammengestellt werden befindet sich im Kapitel 3.1.1.

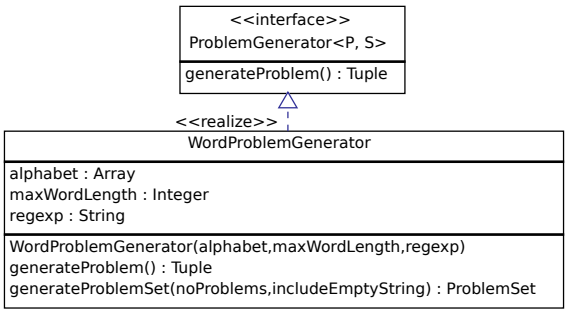

Abbildung 7: Problemgenerierung Klassendiagramm

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, Lösungskandidaten, Referenzautomat)
2. Berechnen der Fitness aller Lösungskandidaten
3. Sortieren der Lösungskandidaten nach Fitness
4. Hat ein Lösungskandidat 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 gebrauchter Zyklen wird zurückgegeben.
5. Von den besten 50% der Lösungskandidaten wird eine *Deep Copy* erstellt
6. Die kopien werden zufällig mutiert
7. Die besten 50% und deren mutierte kopien bilden die neue Menge der Lösungskandidaten
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 Lösungskandidaten auch die Problemmenge verändert wird.

1. Initialisierung (Sprache, Problemmenge, Lösungskandidaten, Referenzautomat)
2. Berechnen der Fitness aller Lösungskandidaten
3. Sortieren der Lösungskandidaten nach Fitness
4. Hat ein Lösungskandidat 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 gebrauchter Zyklen wird zurückgegeben.
5. Von den besten 50% der Lösungskandidaten wird eine *Deep Copy* erstellt
6. Die kopien werden zufällig mutiert
7. Die besten 50% und deren mutierte kopien bilden die neue Menge der Lösungskandidaten.
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.

3.3 Lokale mutierende Problemengen

Protokoll Kick-Off Meeting

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

Abbildungsverzeichnis

Tabellenverzeichnis

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

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

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