technische universität
dortmund

Endbericht der Projektgruppe 578

HeISs - Hierarchische Inferenz evolvierender Spielstrategien

Sebastian Buschjäger, Markus Frohme,
Maren Geske, Simon Hacks, Paul Knulst,
Jos Kusiek, Jobby Malayil, Maik Manjura,
David Spautz

31. Oktober 2014

Betreuer:
Prof. Dr. Bernhard Steffen
Dr. Johannes Neubauer
Dipl.-Inf. Oliver Bauer

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Programmiersysteme (LS5)
`http://ls5-www.cs.uni-dortmund.de`

Inhaltsverzeichnis

1 Einleitung **1**

1.1 HeISs 1
1.2 Erstes Semester 3
1.3 Zweites Semester 3
1.4 Aufbau des Dokuments 5

2 Framework **7**

2.1 Aufbau 7
2.2 Kommunikation 8
2.2.1 Java Message Service 9
2.2.1.1 Modelle 9
2.2.1.2 Implementierung 10
2.3 Game Under Learning 10
2.3.1 Connect Four 13
2.3.1.1 Einfache KIs für Spielstrategien 13
2.3.1.2 Fortgeschrittene KIs für Spielstrategien 14
2.3.1.3 Determinisierung durch festen Random-Seed 19
2.3.2 Chain Reaction 20

3 Passives Lernen **23**

3.1 Theoretische Grundlagen 23
3.1.1 Modellbasierte Verfahren 25
3.1.2 Nicht-modellbasierte Verfahren 32
3.2 Realisierung 38
3.2.1 Implementierung 38
3.2.2 Anbindung an das Framework 40

4 Aktives Lernen **43**
4.1 Theoretische Grundlagen . . . 43
4.2 Realisierung . . . 46
4.2.1 Verwendetes Framework . . . 46
4.2.2 Erste Implemenierung und Allgemeines . . . 47
4.2.2.1 Allgemeiner Lernablauf . . . 48
4.2.2.2 Vor- und Nachteile . . . 49
4.2.3 Finale Implementierung . . . 50
4.2.3.1 Konfigurationsmöglichkeiten . . . 52
4.2.3.2 Adaptionen der LearnLib . . . 54
4.2.4 Anbindung an das Framework . . . 55
4.2.4.1 Konfigurationsdatei . . . 55
4.2.4.2 Speichern der Ergebnisse . . . 57

5 Inferenz **59**
5.1 Theorie . . . 59
5.2 Realisierung . . . 61
5.3 Beispiel . . . 62

6 Auswertung **67**
6.1 Modellqualität . . . 67
6.2 Inferenzqualität . . . 74

7 Fazit **77**
7.1 Zusammenarbeit im Team . . . 77
7.2 Evaluierte Techniken . . . 78
7.3 Umgesetzte Funktionen im Framework . . . 79
7.4 Ausblick . . . 79

Abbildungsverzeichnis **83**

Algorithmenverzeichnis **85**

Quellcodeverzeichnis **87**

Literaturverzeichnis **93**

1 | Einleitung

HeISs steht für „Hierarchische Inferenz evolvierender Spielstrategien“ - dem schrittweisen Ermitteln sich verändernder Spielstrategien. Ist die Strategie durch das Lernen in ein abstraktes Modell überführt, wie zum Beispiel in einem endlichen Automaten, kann dieses genutzt werden, um daraus Gewinnstrategien zu entwickeln. Dies ist bei deterministischen Spielen mit klassischem Automatenlernen durch das Erlernen des kompletten Zustandsraums möglich.

1.1 HeISs

Als Strategie bezeichnet man das *längerfristig ausgerichtete Anstreben eines Ziels unter Berücksichtigung der verfügbaren Mittel und Ressourcen* [Wik13]. Strategien beschreiben somit eine Planungstätigkeit, die nicht nicht nur in Brett- und Computerspielen eine wesentliche Rolle spielen, sondern auch in der Unternehmensführung und im alltäglichen Leben ihren Platz haben. Aus Sicht der Informatik lassen sich komplexe Systeme als Spiele mit Gegnern modellieren, für welche man eine Strategie finden möchte, mit der die genutzte Strategie des Gegners sich immer schlagen lässt. Hierbei ist es nicht nur wichtig die Strategie des Gegners zu kennen, sondern ebenfalls die (sich eventuell ändernden) Spielregeln des komplexen Systems zu erlernen und zu nutzen.
Zur Ermittlung gegnerischer Strategien bzw. der konkreten Spielregeln können in computerunterstützten Spielen das aktive und das passive Automatenlernen hilfreich sein. Das Anwendungsspektrum des Automatenlernens beschränkt sich nicht nur auf Spiele. Es hat seine Stärken vor allem im Bereich der testbasierten Qualitätssicherung evolvierender Systeme [WNS+13].

Im Rahmen dieser PG ist ein Verfahren, das passives und aktives Lernen einsetzt, entwickelt und implementiert worden, das es ermöglicht, gegnerische Strategien zu inferieren. Ausgehend von deterministischen Strategien sollten - soweit möglich -

Verfahren zum Umgang mit Nichtdeterminismus erfasst und untersucht werden. Folgende Punkte sollten im Zeitraum der PG umgesetzt werden [LCTD14b]:

1. Inferieren der Zugauswertungsregeln von Spielen mittels aktiven und passiven Automatenlernverfahren

2. Modellieren von (ausführbaren) Spielstrategien als Automaten

3. Adaption der zuvor verwendeten Automatenlernverfahren auf Inferenz dieser Spielstrategien

4. Analyse zur Eignung der entwickelten Verfahren bei evolvierenden und/oder nicht-deterministischen Spielstrategien

Die Projektgruppe HeISs wurde im Wintersemester 2013/2014 und im Sommersemester 2014 vom Lehrstuhl 5 der Fakultät Informatik der Technischen Universität Dortmund mit acht studentischen Mitgliedern durchgeführt. Unter der Leitung vom Prof. Dr. Bernhard Steffen wurde die Projektgruppe von zwei weiteren wissenschaftlichen Mitarbeitern - Johannes Neubauer und Oliver Bauer - betreut. Die Umsetzung des Projekts ist in mehrere Phasen unterteilt worden. Zunächst wurde die Seminarphase genutzt um sich in die Themen - das passive Automatenlernen und das aktive Automatenlernen - einzuarbeiten. In dieser Phase wurden auch die verschiedenen Ziele, Vorgehensweisen und die allgemeine Organisation diskutiert und definiert. In der nächsten Phase wurden die Lernalgorithmen zunächst näher untersucht und für ein einfaches Spiel implementiert. Diese Arbeit wurde zum Ende des ersten Semesters abgeschlossen. Die hierbei gewonnen Erfahrungen wurden schließlich dazu genutzt in der nächsten Phase im zweiten Semester ein Framework zu erstellen, welches das Lernen von weiteren strategiebasierten Brettspielen ermöglicht.

Die für die Umsetzung des Projekts benötigten Aufgaben wurden schrittweise und in kleinen Teilaufgaben konzipiert, definiert und umgesetzt. Die Gruppe traf sich jede Woche zu einem Abstimmungstermin. In der Sitzung wurden für die darauf folgende Woche Aufgaben definiert und an die Mitglieder verteilt. Gleichzeitig wurden die Ergebnisse aus der vorherigen Woche vorgestellt und die Schwierigkeiten bei der Umsetzung besprochen und Lösungsstrategien entwickelt. Kleine Teams wurden gebildet für eine bessere Organisationsstruktur.

1.2 Erstes Semester

Im ersten Semester hat sich ein Teil der Gruppe mit der Entwicklung des Anwendungsbeispiels - *Connect Four*- befasst. *Connect Four* wurde in seiner Kernfunktion [LCTD14a] neu implementiert, um dabei die Kernpunkte einer Spielesimulation inklusive ihrer künstlichen Intelligenz zu identifizieren. Des Weiteren wurde die Anbindung von *Connect Four* durch diese Eigenimplementierung an bestehende Lernalgorithmen durch Maßschneiderung der Schnittstellen deutlich vereinfacht. Zusätzlich konnte die Simulation durch den Einsatz eines optimierten Spielfeldes [Pep05] beschleunigt werden.
Zwei weitere Teams implementierten Lernalgorithmen zum aktiven und passivem Lernen, so dass sie auf *Connect Four* anwendbar ist. In beiden Teams wurde nach Möglichkeit auf bestehende Bibliotheken zurückgegriffen und diese in den sich entwickelnden Workflow eingebettet.
Eine vierte Gruppe beschäftigte sich mit weniger klassischen Ansätzen zum Automatenlernen, wie z.B. der Verwendung von logischen Formeln oder der Anwendbarkeit von Transformationen auf baumartigen Strukturen, die in unserem konkreten Beispiel dem Zustandsraum aller möglichen Züge entsprachen. Aus diesen Recherchen ergaben sich die spätere Anwendung von statistischen Lernenverfahren im Bereich des passiven Lernens, sowie einige Ideen zur eigentlichen Gegnerinferenz.

1.3 Zweites Semester

Durch die breit gefächerte Arbeit der Gruppe im ersten Semester war zu Beginn des zweiten Semester die Möglichkeit gegeben ein allgemeineres Framework zum Lernen von Spielstrategien zu implementieren. Hier konnten nicht nur die Erfahrungen mit *Connect Four* genutzt werden, sondern vor allem auch die Erfahrungen mit den verschiedenen Lernalgorithmen. Als weiteres Testfeld wurde kurz vor dem Anfang des zweiten Semesters eine Schnittstelle für *Chain Reaction* implementiert um in einem komplexeren Spiel, verglichen mit *Connect Four*, die Arbeitsergebnisse aus dem ersten Semester zu testen. Dennoch entschied sich die Gruppe anstatt die Evaluierung des *Chain Reaction* zu vertiefen, sich auf die Entwicklung des Frameworks zu konzentrieren.
Das Framework wird in Abbildung 1.1 dargestellt. Zentrale Eingabe ist ein Spiel, in dem verschiedene Künstlichen Intelligenzen (KI) eingestellt werden können. Dieses Spiel muss entweder selber oder über einen Wrapper das Interface der Game

Under Learning (GUL) implementieren. Über dieses Interface können das passive und das aktive Lernen die jeweilige Strategie der eingestellten KI in einen Automaten extrahieren. Diese Automaten kann die Inferenz im Anschluss daran dazu nutzen zu erkennen, welche KI in einem Spiel eingestellt wurde. Sollte die Inferenz dabei ein mögliches Beispiel finden, das möglicherweise noch nicht in einem aktiven Automaten berücksichtigt wurde, teilt sie dies dem aktiven Lernen mit. Die Kommunikation zwischen den einzelnen Komponenten wird dabei über einen JMS-Server abgewickelt, was jedoch nicht in der Abbildung dargestellt wird, da dies fachlich keine Relevanz hat.

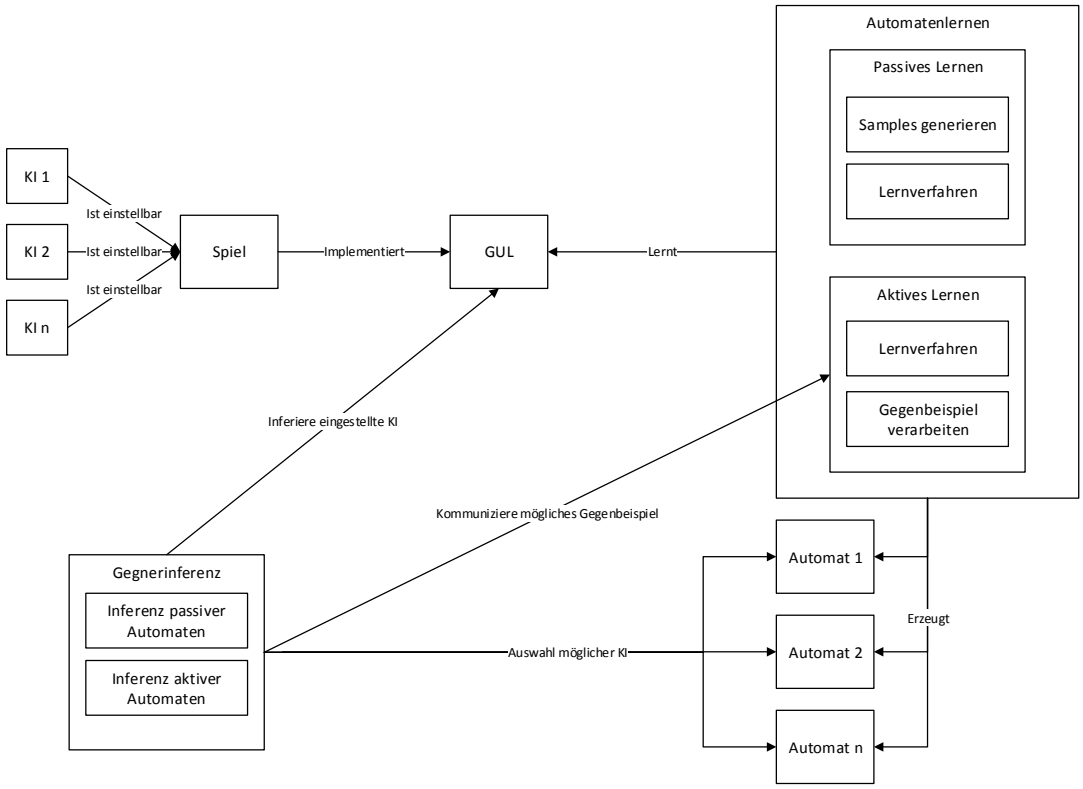


Abbildung 1.1: Zusammenspiel der einzelnen Komponenten.

Hauptziele bei der Entwicklung des generischen Frameworks waren folgende Funktionen anzubieten:

1. Im ersten Schritt soll ein Spiel mittels, der im ersten Semester implementierten, aktiven, passiven und statistischen Lernalgorithmen, gelernt werden.

2. Darauf folgend soll im nächsten Schritt der Gegner identifiziert werden. Wenn dies nicht möglich ist, soll der erste Schritt wiederholt werden, um das genutzte Modell zu verfeinern.

3. Um das generische Framework, zukünftig mit der Möglichkeit im letzten Schritt eine Gegenstrategie anzubieten, auszustatten, soll eine entsprechende Schnittstelle angeboten werden.

Als Minimalziel sollten *Connect Four* und *Chain Reaction* auf dem Framework gelernt werden können. In Absprache mit den Betreuern wurde dieses Ziel leicht vereinfacht. Als Verfeinerung wurde noch ein Messageserver (JMS) zur Funktionalität hinzugefügt. Dies ermöglicht ein besseres Koordinieren der einzelnen Schritte, weil das Framework durch einfache Nachrichten gesteuert werden kann. Somit ist prinzipiell verteiltes und paralleles Arbeiten durch den Messageserver möglich.
Die PG 578 war als eine PG konzipiert, die nicht auf Arbeiten vorhergehender Gruppen basierte und aufgrund der Problemstellung den Mitgliedern der PG viel Freiraum ließ, weshalb der konkrete Fokus selbst gewählt werden konnte. Stattdessen sollten in einem breiteren Rahmen generell das Thema des Lernens untersucht werden. Die Hauptvorgaben und die Themen der Untersuchung waren vorgegeben, die näheren Teilziele sollten in der PG von den Mitgliedern unter Aufsicht der Betreuern erarbeitet werden. Hierbei bildete nach hinreichender Diskussion die Implementierung des Frameworks einen zentralen Punkt, um die Grundlage für spätere Projektgruppen zu legen.

1.4 Aufbau des Dokuments

Das Dokument gliedert sich in sieben Kapitel, beginnend mit dieser Einleitung. Im zweiten Kapitel wird zunächst das Endprodukt dieser PG in seiner Funktionalität beschrieben. In Kapitel drei und vier werden die wichtigen theoretischen Grundlagen des passiven und aktiven Automatenlernens jeweils vorgestellt. In diesen Kapiteln wird auch jeweils die Realisierung der Algorithmen und die Anbindung an das Framework im Einzelnen behandelt. Im fünften Kapitel wird die Grundlagen zur Inferenz und dessen Realisierung vorgestellt. Im sechsten Kapitel werden die Modelle und Arbeiten der beiden Semester evaluiert und die Qualität der Ergebnisse ausgewertet. Im letzten Kapitel erfolgt eine Zusammenfassung der Erfolge sowie ein Ausblick auf dem eine mögliche nachfolgende PG aufbauen könnte.

2 Framework

Im Rahmen der Projektgruppe ist ein Framework zur Gegnererkennung entwickelt worden. Hierzu wird ein aktuell laufendes Spiel analysiert und auf Informationen aus vorherigen Spielen zurückgegriffen. Dieses Kapitel beschäftigt sich mit diesem Framework und beschreibt Aufbau und Verhalten. Darüber hinaus enthält es Informationen über die verwendeten Komponenten.

2.1 Aufbau

Das gesamte Framework besteht aus drei Hauptkomponenten die über ein Nachrichtenystem miteinander kommunizieren. Hierbei agiert jedes System eigenständig und kann seperat ausgeführt werden. Auch eine verteilte Ausührung auf unterschiedlichen Rechnersystemen zur Erhöhung der Performanz ist möglich.

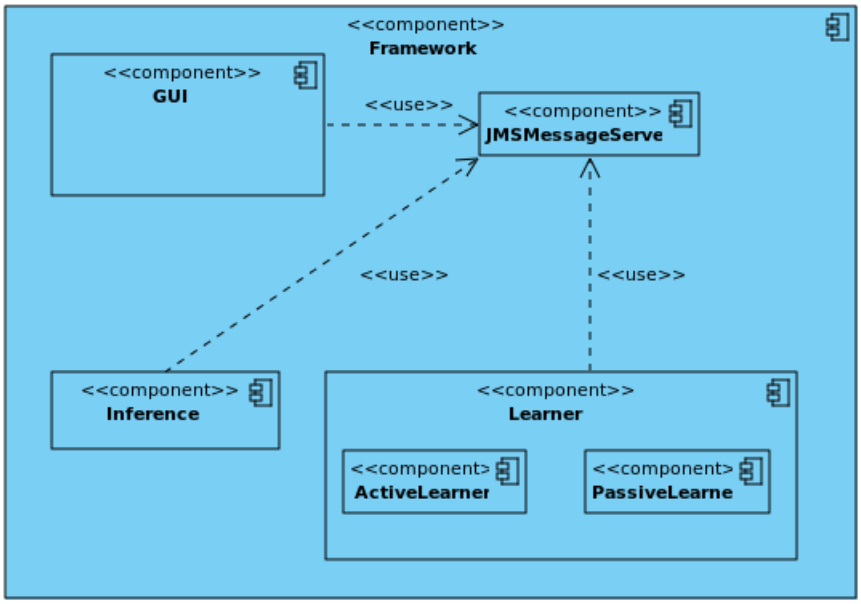


Abbildung 2.1: Darstellung der einzelnen Komponenten in dem Framework.

Bei den Komponenten handelt es sich um die Lernkomponente, die Inferenz und eine GUI zur Steuerung des Lernvorgangs. Das Verteilungsdiagramm in Abbildung 2.1 zeigt die Verbindungen zwischen den einzelnen Komponenten. Es wird deutlich, dass die gesamte Kommunikation der einzelnen Komponenten über einen zentralen Nachrichtenserver stattfindet. Durch diese Architektur sind die einzelnen Komponenten strikt voneinander getrennt, sodass diese unabhängig voneinander funktionieren können. Dies führt zu einer losen Komponentenkopplung in welcher die Komponenten zwar Wissen über den Nachrichtenserver und die verschiedenen Nachrichtentypen haben, allerdings kein Bezug zu einer anderen Komponente besteht. Dadurch ist es möglich jede Komponente durch eine andere, die dieselbe Aufgabe verfolgt, zu ersetzen.
Des Weiteren führt die Trennung dazu, dass einzelne und eventuell rechenintensive Programmteile auf verschiedene Computersysteme ausgelagert werden können um die Performanz zu erhöhen.
Darüber hinaus ist durch den Nachrichtenserver eine asynchrone Bearbeitung der Aufgaben möglich, da sich die Komponente zu beliebigen Zeiten am Nachrichtenserver ab- und anmelden können. Dies erlaubt eine Durchführung der Lernaufträge in einer Art batch-processing, sodass im Anschluss die Inferenz auf diese Ergebnisse zugreifen kann. So ist es auch mglich Nutzereingaben mit Hilfe der GUI beim Empfang von Nachrichten abzufragen und entsprechend zu reagieren. Der Nutzer kann dadurch z.B. einen weiteren Lernvorgang starten sobald ein vorheriger Durchgang abgeschlossen wurde. Parallel dazu versuchen die Inferenzkomponenten, die Gegner mit den gelernten Modellen zu erkennen. Um die Ergebnisse zu erhalten ist es nicht notwendig, dass die GUI durchgehend ausgeführt wird. Sie kann geschlossen werden und zu einem späteren Zeitpunkt geöffnet werden um die Ergebnisse anzuschauen.

2.2 Kommunikation

Einen wichtigen Punkt des Frameworks bildet die Kommunikation zwischen den einzelnen Komponenten. Damit jede Komponente eigenständig funktionieren kann, darf diese keine Verbindung zu einer anderen Komponente besitzen. Um dies zu garantieren wurde ein Nachrichtenserver implementiert, welcher den Kern der Kommunikation bildet. Dieser sorgt für die Vermittlung und Lagerung von Nachrichten. Dabei wurde sichergestellt, dass jederzeit Nachrichten für eine bestimmte Komponente versendet werden können selbst wenn diese zur Zeit nicht erreichbar ist. Der Nachrichtenserver übernimmt die Speicherung und asynchrone Vermittlung der

Nachricht.
Das ermöglicht die Auslagerung rechenintensiver Komponenten auf verschiedene Computersysteme, um die Gesamtperformance zu erhöhen.
Der Nachrichtenserver wurde mit Hilfe des `Java Message Service` implementiert, welches im Folgenden genauer erklärt wird.

2.2.1 Java Message Service

Bei dem `Java Message Service (JMS)` handelt es sich um eine Programmierschnittstelle die ein Teil der Java Platform Enterprise Edition (JEE) ist [MH99]. JMS wird genutzt um Nachrichten zwischen einzelnen Komponenten verteilter Anwendungen zu Senden und Empfangen. Dabei findet die Kommunikation zwischen den Komponenten beim Senden und Empfangen asynchron statt. Diese Art der Kommunikation ermöglicht eine zeitliche Entkopplung zwischen Sender und Empfänger. So werden Komponenten weiter voreinander entkoppelt, welches den Austausch einzelner Komponenten und somit die Wartbarkeit des Gesamtsystems erhöht und erleichtert.

2.2.1.1 Modelle

Innerhalb von JMS gibt es zwei Modelle, welche zum Versenden von Nachrichten genutzt werden können: „Point-to-point“ und „publish and subscribe“.

Point-to-point: Bei dem „point-to-point“ Modell handelt es sich um ein Warteschlangen-Modell, welches die Nachrichten an eine Queue weitergibt an welche ein oder mehrere Empfänger gekoppelt sind. Hierbei ist jede Nachricht einem bestimmten Empfänger zugeordnet und kann nur von diesem abgeholt werden. Das bedeutet, dass der Sender den Empfänger kennen muss, um eine Nachricht zu schicken. Falls beim Versenden der Nachricht der Empfänger nicht an der Queue lauscht wird die Nachricht gespeichert. Die Speicherung hält solange an bis der richtige Empfänger sich anmeldet und die Nachricht abholt oder die maximale Speicherzeit erreicht ist.

Publish and subscribe: Bei dem „publish and subscripe“ Modell handelt es sich um einen Ansatz, welcher Nachrichten an Komponenten senden, die einem bestimmtem Themengebiet - einem sog. Topic zugeordnet sind.An diesem Topic können sich Abonnenten registrieren, welche dann Nachrichten empfangen, die an das Topic gesendet wurden. Hierbei haben Empfänger und Sender keine Beziehung zueinander

und kennen sich nicht. Wenn es keine Abonnenten gibt, bleibt die Nachricht bestehen, bis sie abgerufen wird, verfällt durch Erreichen des Verfallsdatum oder durch eine andere ersetzt wird.

2.2.1.2 Implementierung

Innerhalb des Frameworks kommt das „publish and subscribe“ Modell zum Einsatz. Dieses Verfahren wurde gewählt, da es einige Vorteile gegenüber dem „point-to-point“ besitzt. Innerhalb des Framework wurde das „publish and subscribe“ Verfahren verwendet, da die Architektur des Frameworks so aussieht, dass eine Komponente Nachrichten von verschiedenen Empfängern erhält. Der Vorteil gegenüber dem „point-to-point“ Verfahren liegt darin, dass ein Topic pro Komponente ausreicht, um Nachrichten zu versenden. Beim „point-to-point“ Verfahren wäre es nötig gewesen, für jede Verbindung zwischen zwei Komponenten eine Queue zu erstellen, so dass jede Komponente mindestens zwei offene Queue haben müsste.
Beim „publish and subscribe“ Verfahren hingegen wird jeder Komponente ein Topic zugeordnet, an welches Nachrichten gesendet werden können, die dann im Anschluss von der Komponente bearbeitet werden. So reicht ein Topic pro Komponente aus. Hier besitzt der Nachrichtenserver drei verschiedene Topics: „Learner“, „Inferenz“ und „GUI“. Diese drei Topics entsprechen einer Liste von Aufgaben, die in den verschiedenen Komponenten abgearbeitet werden.
Des weiteren ist es möglich beim Starten einer Komponente den Nachrichtenserver zu kontaktieren und mögliche Nachrichten abzurufen, die daraufhin abgearbeitet werden.
Die Nachrichten in dem Framework sind so aufgebaut, dass sie den Sender, eine eindeutige Message-ID und ein Datenfeld enthalten. Durch eine Kombination von Sender und Message-ID ist es dann möglich die richtige Aufgabe auszuführen. Das Datenfeld enthält hierbei optionale Daten, die Anwendungsparameter enthalten können.

2.3 Game Under Learning

Der zentrale Bestandteil der Learner-Komponente (vgl. Abbildung 2.1) bildet das `Game Under Learning (GUL)`. Das GUL stellt die Schnittstelle zu dem Spiel bereit, welches gelernt werden soll. Hierzu erweitert es das SUL-Interface aus der *LearnLib*[1] (siehe Abschnitt 4.2) um spielespezifische Funktionen.

[1] `http://www.learnlib.de`

Ziel bei der Definition des GULs war es, eine maximal große Anzahl an möglichen Spielen abzudecken um so das Framework möglichst wenig einzuschränken. Dies hat zur Folge, dass der Nutzer lediglich das GUL implementieren muss um einen Lern- und Inferenzvorgang zu modellieren. Hierzu müssen folgende Funktionen implementiert werden:

Listing 2.1: GUL-Interface

```
public interface GUL<I, O> extends SUL<I, O> {
    public Collection<I> getInputAlphabet();

    public Collection<O> getOutputAlphabet();

    public Collection<O> getAcceptingOutputs();

    public Collection<O> getNonAcceptingOutputs();

    public Collection<O> getNeutralOutputs();

    public void setAI(String ai);

    public String getAI();

    public Collection<String> getAvailableAIs();

    public void setParams(Map<String, String> params);
}
```

- **getInputAlphabet:** Das Spiel wird als Black-Box mit definierten Eingaben betrachtet. Eine Eingabe kann z.B. ein gesamter Zug oder ein Teil eines Zuges sein. Das Eingabealphabet bezeichnet die Menge aller möglichen Eingaben die an das Spiel getätigt werden können.

- **getOutputAlphabet:** Analog zu den definierten Eingaben die man an das System stellen kann, antwortet das System mit definierten Ausgaben. Im Allgemeinen ist das Ausgabealphabet eines Blackbox-Systems nicht bekannt, da nur diejenigen Ausgaben bekannt sind, die für bestimmte Eingaben beobachtet werden konnten. Dennoch ist es für die Inferenz von Spielen erforderlich, dass das gesamte Ausgabealphabet bekannt ist. Aus algorithmischer Sicht ergibt sich diese Notwendigkeit aus der Gegnerinferenz (vgl. Abschnitt 5). Rein intuitiv bedeutet dies, dass die Lernalgorithmen zumindest die Regeln des Spiels insofern kennen, als dass sie wissen welche Zustände das Spiel annehmen kann.

- **getAcceptingOutputs:** Eine Teilmenge des Ausgabealphabets bezeichnet die Gewinnzustände des Spiels. Dies kann z.B: Ein einzelner dezidizierter GE-

WONNEN-Zustand sein oder eine Menge verschiedener Zustände wie z.B. {SPIELER_1_GEWONNEN,SPIELER_2_GEWONNEN}.

- **getNonAcceptingOutputs:** Analog zur Teilmenge der Gewinnzustände kann ein Spieler auch verlieren. Auch dies kann widerum ein einzelner VERLOREN-Zustand sein oder eine Menge verschiedener Zustände wie z.B. {SPIELER_1_VERLOREN,SPIELER_2_VERLOREN}.
- **getNeutralOutputs:** Viele Spiele können auch in einem Unentschiedenen enden. Diese Zustände können hier definiert werden.
- **getAvailableAIs:** Liefert eine Menge von möglichen Gegner-KIs zurück die das Spiel zur Verfügung stellt.
- **setAI(String ai):** Eine Hilfsfunktion zum Setzen der aktuellen Gegner-KI im Spiel. Das Framework ruft diese Funktion immer mit einer KI auf, die durch `getAvailableAIs` zurückgegeben wird.
- **getAI:** Liefert die aktuell gesetzte KI zurück.
- **setParams:** Bietet die Möglichkeit Parameter, wie z.B. Spielfeldgröße für das Spiel zu definieren und zu setzen. Das Framework liest die Parameter und ihre Werte lediglich ein führt aber kein besonderes Parsing durch. Dies ist dem Spiel selber überlassen.
- **pre/post/step(I in):** Diese Funktionen werden von dem SULInterface der *LearnLib* vorgegeben und müssen entsprechend der Konventionen der *LearnLib* implementiert werden.

Zentrale Aspekte des GULs sind die Modellierung des Spiels in Ein- und Ausgabesymbole, sowie die passende Anbindung des Spiels in die `step(...)`-Funktion. Die Modellierung des Spiels in Ein- und Ausgabesymbole lässt einigen Spielraum zum experimentieren. So können die Eingabesymbole des GULs die Züge des Frameworks sein, die Ausgabesymbole jedoch den aktuelle Spielzustand (nach dem Zug der Gegner-KI) darstellen. Auf diese Art und Weise würde das Framework vor allem die Spielzustände lernen. Alternativ können die Ausgabesymbole die tatsächlichen Gegnerzüge modellieren, sodass das Framework speziell die Züge des Gegners lernt. Des Weiteren ist es dem Nutzer frei überlassen, inwiefern das Ausgabealphabet in Gewinn-,Neutral- und Verlierzustände modelliert wird. Es wird jedoch angenommen, dass diese drei Mengen paarweise disjunkt voneinander sind. Eine übliche Annahme

wäre es z.B: einen expliziten GEWONNEN-Zustand zu modellieren und alle übrigen Zustände als VERLOREN zu werten - in diesem Fall gäbe es kein Unentschieden bzw. ein Unentschieden wird als VERLOREN gewertet. Auch hierbei stellt das GUL bzw. das Framework nur minimale Anfoderungen und lässt dem Nutzer viele Freiheiten offen.
Um die Algorithmen im Framework zu testen wurden zwei Spiele betrachtet: *Connect Four* und *Chain Reaction*. Die folgenden zwei Kapitel erklären kurz die Regeln dieser Spiele und beschreiben ihre Anbindung an das Framework.

2.3.1 Connect Four

Connect Four, auch bekannt als `Vier gewinnt` ist ein einfaches Spiel für zwei Spieler. Ein übliches *Connect Four* Spielfeld besteht aus sieben Spalten und sechs Zeilen, d.h. 42 Feldern. Jeder Spieler wirft nacheinander Spielsteine seiner Farbe in einer der Spalten, wobei der Spielstein den untersten freien Platz dieser Spalte besetzt. Das Spiel endet, sobald einer der beiden Spieler eine horizontale, vertikale oder diagonale Reihe mit vier Steinen erreicht. Wenn dies keinem Spieler gelingt bevor alle Spielfelder belegt sind, so endet das Spiel unentschieden.
Connect Four eignet sich gut als einfaches Beispiel für Spielstrategien. Die Regeln des Spiels sind einfach genug, sodass jeder (menschliche) Spieler dieser auf Anhieb verstehen kann, jedoch widerum so komplex, dass man eine optimale Lösung nur schwer berechnen kann [All88]. Es bietet einige Möglichkeiten der Variation - die Anzahl der Spalten und die Anzahl der Zeilen ist beliebig erweiterbar um das Spiel noch komplexer zu machen. Zusätzlich können bei Bedarf weitere Dimensionen hinzugefügt werden [WIK14].

2.3.1.1 Einfache KIs für Spielstrategien

Durch die Einfachheit des Spiels konnten verschiedene KIs implementiert werden um damit die verschiedenen Aspekte der Inferenz zu untersuchen. Hierzu werden folgende Spielstrategien, die zunächst für *Connect Four* verwendet wurden, näher erläutert:

- Clocky: Diese KI platziert ihre Spielsteine immer eine Spalte weiter nach rechts und fährt dann im Uhrzeigersinn fort.
- Downy: Hier besteht die Strategie darin, zunächst immer in die Spalten zu werfen, in denen die Anzahl der bereits platzierten Steine minimal ist. Existieren mehrere solcher Spalten, wird so weit wie möglich nach links gesetzt.

- Lefty: Die Vorgehensweise dieser KI besteht aus dem letzten Ansatz von Downy. Es werden so lange Spielsteine weit möglichst nach links gesetzt, bis die entsprechende Spalte voll ist und die nächste befüllt werden kann.

Aufgrund ihrer Simplizität verhalten sich alle drei oben genannten Intelligenzen deterministisch, das bedeutet, dass ihre Vorgehensweisen eindeutig festgelegt sind. Durch Heranziehung dieser deterministischen Strategien werden beim Automaten-Lernen die Untersuchung und Nachvollziehbarkeit der erzielten Ergebnisse erleichtert.

2.3.1.2 Fortgeschrittene KIs für Spielstrategien

Als Gegenstück zu den drei oben genannten Vorgehensweisen wird auch eine KI benötigt, die intelligenter agiert und dadurch keine bloßen Muster abarbeitet. Dies erhöht den Anspruch beim Automaten-Lernen, erzielte Ergebnisse können anhand von unterschiedlich eingestuften KIs besser miteiander verglichen werden. Als komplexe Lernstrategie ist hier das sogenannte AlphaBeta-Pruning ausgewählt worden, welches im Folgenden erläutert wird.
Das AlphaBeta-Pruning ist eine Weiterentwicklung des MiniMax-Verfahrens. Beim MiniMax-Verfahren wird der Zug ausgesucht, der am meisten verspricht, dass die künstliche Intelligenz am Ende des Spiels siegreich sein wird. Dazu werden alle Züge und deren darauf folgenden Züge berechnet und dann der Zug ausgewählt, bei dem es am wahrscheinlichsten ist, dass man gewinnt. Bei jedem Zug versuchen beide Spieler einen maximalen Wert zu erreichen, also zu gewinnen. Das „Mini“ rührt daher, dass eine Bewertung immer aus der Sicht desselben Spielers statt findet und eine Maximierung des Gegners eine Minimierung für einen selber darstellt. [Lug02, S. 168ff]
Allerdings kann dieses Verfahren nur bei sehr trivialen Spielen, wie Tic-Tac-Toe, in akzeptabler Zeit bis zum Ende durchgerechnet werden. Bei komplexeren Spielen sollte ab einer bestimmten Tiefe die Berechnung abgebrochen werden, da zum Beispiel bei Tic-Tac-Toe circa 40.000 Züge berechnet werden müssen [Mat14]. Wird die Berechnung vorzeitig abgebrochen, lässt sich jedoch nicht immer bestimmen, ob ein Zug zum Sieg führt. Daher müssen Heuristiken genutzt werden, die abschätzen, ob das Spielfeld eine positive oder negative Entwicklung für den Spieler bedeutet.
Allerdings kann man mit dem MiniMax-Verfahren nicht besonders tief den Spielbaum untersuchen. Aus diesem Grund wurde das AlphaBeta-Pruning entwickelt. Das AlphaBeta-Pruning liefert dasselbe Ergebnis wie das MiniMax-Verfahren. Allerdings wird der Spielbaum nicht komplett durchsucht, was Zeit spart, die in eine

tiefere Suche reinvestieren werden kann. Der Algorithmus entscheidet selbstständig, ob es sich lohnt einen Zweig weiterhin zu untersuchen. Um diese Entscheidung zu treffen wird davon ausgegangen, dass beide Spieler ihre Bewertung für das Spielfeld maximieren wollen. Wenn nun unterhalb eines MIN-Knotens, also eines gegnerischen Zuges, einer der möglichen Züge einen Wert liefert, der kleiner oder gleich dem bisherigen Wert ist, kann die Suche dort beendet werden. An einem MAX-Knoten, also einem eigenen Zug, kann die Suche beendet werden, wenn einer der nachfolgenden einen größeren oder gleichen Wert liefert, wie der MAX-Knoten bisher hatte. [Lug02, S. 176 ff]

> Ein anschauliches Beispiel für die Funktionsweise [des AlphaBeta-Pruning] ist ein Zweipersonenspiel, bei dem der erste Spieler eine von mehreren Taschen auswählt und von seinem Gegenspieler den Gegenstand mit geringstem Wert aus dieser Tasche erhält.
>
> Der MiniMax-Algorithmus durchsucht für die Auswahl alle Taschen vollständig und benötigt somit viel Zeit. Die AlphaBeta-Suche hingegen durchsucht zunächst nur die erste Tasche vollständig nach dem Gegenstand mit minimalem Wert. In allen weiteren Taschen wird nur solange gesucht, bis der Wert eines Gegenstands dieses Minimum unterschreitet. Ist dies der Fall, wird die Suche in dieser Tasche abgebrochen und die nächste Tasche untersucht. Andernfalls ist diese Tasche eine bessere Wahl für den ersten Spieler und ihr minimaler Wert dient für die weitere Suche als neue Grenze.
>
> Ähnliche Situationen sind jedem Schachspieler vertraut, der gerade einen konkreten Zug darauf prüft, ob er ihm vorteilhaft erscheint. Findet er bei seiner Analyse des Zuges eine für sich selbst ungünstige Erwiderung des Gegners, dann wird er diesen Zug als „widerlegt“ ansehen und verwerfen. Es wäre völlig sinnlos, noch weitere Erwiderungen des Gegners zu untersuchen, um festzustellen, ob der Gegner noch effektivere Widerlegungen besitzt und wie schlecht der geplante Zug tatsächlich für den Spieler ist. [Bew10, S. 184]

Pseudocode Nachstehend wird der Pseudocode des AlphaBeta-Pruning vorgestellt. Zu Beginn gibt es ein Hauptprogramm (Algorithmus 2.1), das den eigentlichen Algorithmus aufruft. Es gibt zwei wesentliche Implementierungsarten. Bei der einen wird die Bewertung des Spielfelds durchgehend aus der Sicht eines Spielers durchgeführt. Bei der anderen Variante wird die Bewertung immer aus der Sicht des Spielers

durchgeführt, der in der die Simulation am Zug ist. Da hierbei der Zug des Gegner negiert werden muss, nennt man diese Variante auch NEGAMAX-Vorgehen. Die letzte Variante soll für den Rest der Arbeit weiter betrachtet werden (Algorithmus 2.2), da bei dieser der Implementierungsaufwand geringer ist.

Algorithmus 2.1 HAUPTPROGRAMM ALPHABETA-PRUNING

Eingabe: $targetDepth$, $actualPlayer$

Ausgabe: nothing

```
1: savedTurn ← NULL
2: rating ← NEGAMAX(actualPlayer, targetDepth, −∞, ∞)
3: if savedTurn == NULL then
4:     print no more moves possible
5: else
6:     do savedTurn
7: end if
```

Im Hauptprogramm wird überprüft, ob die künstliche Intelligenz sofort gewinnen oder einen gegnerischen Sieg sofort verhindern kann. Trifft beides nicht zu, wird für jede erlaubte Spalte die NEGAMAX-Funktion aufgerufen, um eine Bewertung der Spalten zu erhalten. Die NEGAMAX-Funktion stellt den zweiten wesentlichen Teil dar. In ihr wird das eigentliche ALPHABETA-Pruning durchgeführt. An dieser Stelle besteht auch das größte Potenzial den Algorithmus zu beschleunigen. Die letzte wesentliche Stelle ist die Bewertung des Spielfelds, die von der NEGAMAX-Funktion aufgerufen wird, wenn die gewünschte Tiefe im Spielbaum erreicht wurde.

Hauptprogramm Zu Beginn wird das Spielfeld geklont und nur noch auf dem Klon gearbeitet. Theoretisch besteht auch die Möglichkeit permanent auf demselben Spielfeld zu arbeiten, allerdings traten bei der Implementierung, bei der Züge wieder rückgängig gemacht wurden, Seiteneffekte auf, die nicht genau zurück verfolgt werden konnten, weshalb die Variante des Spielfeldklons beibehalten wurde.
Für jede Spalte, in die geworfen werden darf, wird überprüft, ob beim Wurf in diese sofort gewonnen oder eine Niederlage verhindert werden kann. Ist dies nicht der Fall wird für die Spalte mittels der NEGAMAX-Funktion ein Wert ermittelt. Die Spalte, die den höchsten Wert erhält, wird von der künstlichen Intelligenz für den nächsten Zug gewählt.

Negamax-Funktion Die NEGAMAX-Funktion erhält als Übergabe, die Spalte, die bewertet werden soll, die obere und untere Grenze, das geklonte Spielfeld, der Spieler,

Algorithmus 2.2 NEGAMAX

Eingabe: $actualPlayer$, $targetDepth$, $alpha$, $beta$

Ausgabe: $maxValue$

```
1: if targetDepth == 0 or areMovesPossible(actualPlayer) then
2:   return rate(actualPlayer)
3: end if
4: maxValue ← alpha
5: generateNextMoves(actualPlayer)
6: while moves to go do
7:   doNextMove()
8:   value ← -Negamax(−actualPlayer, targetDepth − 1, −beta, −maxValue)
9:   undoLastMove()
10:  if value > maxValue then
11:    maxValue ← value
12:    if maxValue ≥ beta then
13:      BREAK
14:    end if
15:    if targetDepth == startingDepth then
16:      savedTurn ← actualTurn
17:    end if
18:  end if
19: end while
20: return maxValue
```

für den die Bewertung durchgeführt und bis zu welcher Tiefe gesucht werden soll. Der Stein wird in die gewünschte Spalte geworfen und überprüft, ob entweder das Spiel vorbei oder die gewünschte Tiefe erreicht wurde. Ist das der Fall wird die Bewertungsfunktion aufgerufen und der Wert zurück gegeben.
Ansonsten wird für alle noch erlaubten Spalten die NEGAMAX-Funktion, mit dem anderen Spieler, einer geringeren Tiefe und vertauschten Grenzen aufgerufen. Der Rückgabewert wird negiert und überprüft, ob der Wert größer der oberen Schranke ist. Falls das der Fall ist, wird dieser Wert die neue obere Grenze. Ist der Wert größer als die untere Grenze, werden die restlichen Spalten nicht mehr betrachtet. Die obere Grenze wird am Ende zurück gegeben.

Bewertungsfunktion Die Bewertungsfunktion bewertet ein Spielfeld aus Sicht eines übergebenen Spielers. Berücksichtigt werden dabei Siege, Unentschieden, Niederlagen, gesetzte Steine und „Bedrohungen“. Bei den gesetzten Steinen erhalten die Steine eine höhere Wertung, umso weiter sie in der Mitte liegen, da mittigplatzierte Steine eine bessere Ausgangsposition bedeuten. Unter eine Bedrohung werden drei Steine eines Spielers verstanden, sodass noch ein vierter Stein in Reihe gelegt werden kann.
Von der positiven Bewertung des betrachteten Spielers wird zusätzlich die Bewertung der Steine des Gegners abgezogen.

Beschleunigung Bei der praktischen Anwendung fiel auf, dass das AlphaBeta-Pruning zur Berechnung von Zügen relativ lange braucht. Deswegen wurden Möglichkeiten gesucht, die Algorithmus zu beschleunigen:

- *Vorsortierung der Züge*
  Die Suchreihenfolge spielt beim AlphaBeta-Pruning eine wesentliche Rolle, da die Suche früher abgebrochen wird, umso kleiner das Suchfenster, symbolisiert durch die obere und untere Grenze, wird. Da eine vorherige Sortierung für *Connect Four* schwierig erscheint, ohne das Spielfeld intensiv zu betrachten und eine maßgebliche Änderung der Zugdurchführung notwendig wäre, wurde diese Möglichkeit nicht weiter betrachtet.

- *Principal-Variation-Suche*
  Die Knoten bei der Suche können in drei wesentliche Kategorien unterteilt werden: Alpha-Knoten, bei denen die Suche immer kleinere Werte als Alpha zurück liefern wird, weshalb hier keine guten Züge möglich sind; Beta-Knoten, bei denen ein Wert größer oder gleich Beta gefunden wird und ein Cut-Off ausgelöst wird; Principal-Variation-Knoten liefern einen Kindknoten, der größer Alpha ist, und alle Kindknoten sind kleiner Beta.
  Liefert ein Knoten einen Wert zwischen Alpha und Beta besteht die Möglichkeit, dass es ein Principal-Variation-Knoten ist. In dem Fall kann eine Suche in einem verkleinerten Fenster Sinn machen, um einen vorzeitigen Cut-Off zu erreichen. Wird dieser nicht erreicht, muss die Suche mit dem vergrößerten Fenster erneut durchgeführt werden. Es muss folglich zwischen diesen beiden gegenläufigen Effekten abgewägt werden, ob sich der Einsatz lohnt.

- *Iterative Tiefensuche*
  Bei der iterativen Tiefensuche wird die Suchtiefe Schritt für Schritt erhöht,

bis eine definierte obere Schranke erreicht wird. Dadurch kann eine an die Ressourcen des Rechners angepasste Suche durchgeführt werden, wenn zum Beispiel die Zeit als obere Grenze gewählt wird. Dies bedeutet allerdings auch, dass auf unterschiedlichen Rechnern, unterschiedliche Ergebnisse entstehen. Da für die aktiven Lernalgorithmen deterministische künstliche Intelligenzen benötigt werden, wird diese Möglichkeit nicht weiter betrachtet.

- *Bewertungen speichern*
  Eine weitere Möglichkeit besteht darin, die Berechnungen für Spielfelder in einer Tabelle zu speichern. Dazu wird eine Tabelle angelegt, die für jedes Spielfeld den Wert speichert, den es bei einer bestimmten Tiefe erhält. Es wird zusätzlich festgehalten, ob es sich um eine exakte Berechnung oder um eine obere oder untere Grenze handelt. Hierdurch wird nicht nur die Berechnung eingespart, sondern auch der Rekursion muss nicht weiter gefolgt werden, wenn das Spielfeld in der entsprechenden Tiefe schon betrachtet wurde.

2.3.1.3 Determinisierung durch festen Random-Seed

Eine Vielzahl der Lernalgorithmen verlangt eine deterministisch agierende Gegner-KI (vgl. Kapitel 4 und Kapitel 3). Intuitiv hat dies den Hintergrund, dass eine effektive Vorhersage nur dann erfolgen kann, wenn sich der Gegner gleich verhält, d.h. in einem Spielfeldzustand S immer dieselbe Aktion a durchführt.
Nicht-deterministische KIs treffen in einem Zustand S eine zufällige Entscheidung für Aktion a_1 oder Aktion a_2. Diese zufällige Entscheidung basiert letztendlich auf einem Zufallsgenerator, der mit einem Random-Seed initialisiert wird. Initialisiert man den Zufallsgenerator in jedem Lerndurchlauf mit einem fixen Wert, so trifft der Zufallsgenerator immer dieselben Entscheidungen, d.h. es ist bei jedem Lerndurchlauf gewährleistet, dass immer Aktion a_1 oder a_2 im Zustand S gewählt wird.
Zu beachten ist hierbei jedoch, dass ein fester Random-Seed nicht zwingend zu einem Determinismus führt. Falls der Lerner innerhalb eines Lerndurchlaufes merhmals den Zustand S sieht, muss die KI innerhalb dieses Durchlaufes auch mehrmals eine Entscheidung für Aktion a_1 oder a_2 treffen. Der Random-Seed garantiert dann nur, dass diese Entscheidung zwischen zwei Lerndurchläufen immer gleich ist. Es wird jedoch nicht gewährleistet, dass diese Entscheidung innerhalb eines Lerndurchlaufes immer gleich ist.
Im Falle von *Connect Four* stellt dies keinerlei Einschränkung dar, da bei *Connect Four* keinerlei Spielsteine verschwinden oder sich ihre Position ändert. Somit kann jeder Zustand maximal einmal während eines Lerndurchlaufes observiert werden.

Im Falle von *Chain Reaction* kann jedoch ein und derselbe Zustand mehrmals innerhalb eines Lerndurchlaufes observiert werden, da hierbei Spielsteine ihre Position verändern und konsumiert werden können (vgl. Abschnitt 2.3.2).

2.3.2 Chain Reaction

Chain Reaction ist ein Zweipersonenspiel in dem alle Spielsteine des Gegners auf dem Spielbrett zu erobern sind. Ein normales *Chain Reaction* Spielfeld besteht aus 5×5 Feldern, wobei innenliegende Felder maximal vier, außenliegende Felder drei Spielsteine und Eckfelder zwei Spielsteine aufnehmen können. Jeder Spieler legt abwechselt einen Spielstein auf ein freies Spielfeld oder ein Spielfeld mit Steinen seiner Farbe. Sobald die maxixmale Anzahl der Spielsteine auf einem Feld erreicht ist, beginnt eine Kettenreaktion. In dieser Reaktion verteilen sich die Spielsteine auf einem Feld auf die vertikal und horizontal benachbarten Felder. Ist bei einem benachbarten Feld die maximale Anzahl erreicht, so wird auch dort eine Reaktion ausgelöst, bis auf keinem Feld mehr die maximale Anzahl der Spielsteine zu finden ist.
Bei einer Kettenreaktion werden Steine des Gegners konsumiert und in die eigene Farbe umgewandelt. Ziel des Spiel ist es, alle Felder des Spiels mit Hilfe einer Kettenreaktion in seine eigene Spielfarbe zu bringen.
Chain Reaction bietet somit interessante Eigenschaften für Lernalgorithmen an. So ist das komplexe Zusammenspiel der realtiv einfachen Regeln bereits nach einigen Zügen schwer vorherzusagen. Hinzu kommt, dass die Auswertungsreihenfolge der einzelnen Felder eine entscheidende Rolle für die Kettenreaktion spielen kann (vgl. Abbildung 2.2), sodass diese im Laufe des Spieles zunächst gelernt werden muss.
Dadurch bietet *Chain Reaction* neben einem größeren Zustandsraum auch eine Fülle an Konfigurationsmöglichkeiten um die Algorithmen in verschiedensten Konfigurationen zu testen. Es lässt sich damit nicht nur die Spielfeldgröße, sondern auch die maximale Steinanzahl auf einzelnen Feldern frei konfigurieren. Zusätzlich kann durch Anpassen der Auswertungsreihenfolge einzelner Felder (z.B. zufällig) Determinismus simuliert werden.

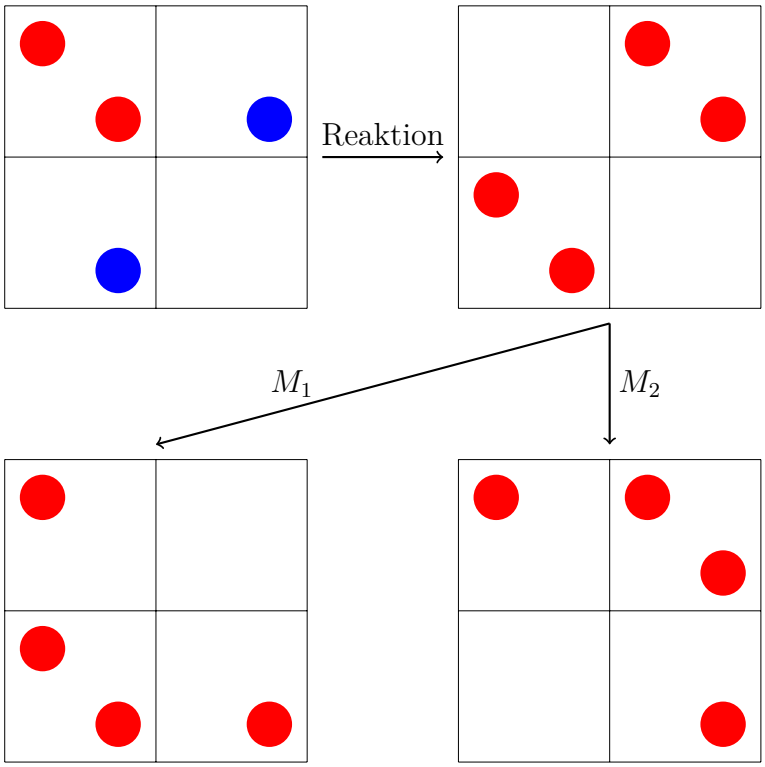


Abbildung 2.2: Eine einfache *Chain Reaction* Kettenreaktion. Der rote Spieler hat seinen zweiten Spielstein in das obere linke Feld gelegt, sodass es zu einer Reaktion kommt. Nach dieser Reaktion sind zwei weitere Reaktionen M_1 und M_2 möglich. Hierbei ist es nun implementierungsabhängig, ob zunächst M_1 oder M_2 ausgeführt wird. Beide Reaktionen führen zu anderen Spielfeldzuständen.

3 | Passives Lernen

Eine große Familie der (Automaten-)Lernverfahren bilden die sogenannten *passiven Lernverfahren*. Der grundlegende Ansatz dieser Verfahren ist es, auf Basis von endlich vielen Beobachtungen eine möglichst gute Approximation des tatsächlichen Systems zu erstellen. Die verwendeten Beobachtungen sind jedoch losgelöst von der konkreten Ausführung des Systems (beispielsweise über Log-Dateien) – insbesondere können also keine direkten Anfragen an das System gestellt werden, um konkrete Beobachtungen zu erhalten.
Im Rahmen dieser Projektgruppe werden zum einen für das Automatenlernen *klassische*, modellbasierte Lernverfahren betrachtet, zum anderen aber auch allgemeine Klassifikationsverfahren, welche beispielsweise auf einer numerischen Basis arbeiten.

3.1 Theoretische Grundlagen

Wie zuvor erwähnt, suchen passive Lernverfahren eine Approximation des tatsächlichen Systems. Als formale Basis, welche das Verhalten eines Systems modelliert, verwenden die hier behandelten passiven Lernverfahren sogenannte Deterministic Finite Automata (DFAs) (deut.: Deterministische Endliche Automaten (DEAs)) [HMU03]. Diese sind im Folgenden lediglich soweit vorgestellt, wie sie in der dieser Ausarbeitung Verwendung finden.

Definition 1 (Deterministischer endlicher Automat: Syntax) *Ein deterministischer endlicher Automat* $\mathcal{A} = (S, \Sigma, \delta, s, F_A, F_R)$ *besteht aus*

- *einer endlichen Menge* S *von Zuständen*
- *einem Eingabealphabet* Σ
- *einer Überführungsfunktion* $\delta : S \times \Sigma \to S$
- *einem Startzustand* $s \in S$
- *den Mengen* $F_A \subseteq S$ *von akzeptierenden und* $F_R \subseteq S$ *von ablehnenden Zuständen mit* $F_A \cap F_R = \emptyset$

Häufig wird statt der Mengen F_A, F_R auch nur eine Menge F von akzeptierenden Zustände angegeben. Implizit gilt dann $F_A = F, F_R = S \setminus F$.

Definition 2 (Deterministischer endlicher Automat: Semantik) *Sei* $\mathcal{A} = (S, \Sigma, \delta, s, F_A, F_R)$ *ein DFA, so ist seine erweiterte Überführungsfunktion* $\delta^* : S \times \Sigma^* \to S$ *wie folgt definiert:*

- $\delta^*(q, \epsilon) = q, \quad \forall q \in S$
- $\delta^*(q, u\sigma) = \delta(\delta^*(q, u), \sigma), \quad \forall q \in S, \forall u \in \Sigma^*, \forall \sigma \in \Sigma$

wobei $\epsilon \in \Sigma^*$ *das leere Wort bezeichnet. Der Automat* $\mathcal{A}$ *akzeptiert ein Wort* w *genau dann, wenn* $\delta^*(s, w) \in F_A$ *gilt, d.h. wenn nach Eingabe von* w *der Automat in einem der akzeptierenden Endzustände endet. Die von* $\mathcal{A}$ *akzeptierte Sprache wird dann mit* $L(\mathcal{A}) = \{w \in \Sigma^* \mid \mathcal{A} \text{ akzeptiert } w\}$ *beschrieben.*

Als abkürzende Schreibweise wird vereinzelt ebenfalls q_a für den Zustand $\delta^*(s, a), a \in \Sigma^*$ verwendet.
DFAs bilden im Vergleich zu Mealy-Automaten (siehe Kapitel 4) ein vergleichsweise einfaches Modell. Im Kontext von Spiel-Strategien erlauben DFAs jedoch eine sehr direkte Verbindung von Spiel-Strategien und formalen Modellen, da sich gewinnende Strategien auf akzeptierende und verlierende Strategien auf ablehnenden Zustände abbilden lassen. Desweiteren kann die geringe Modellkomplexität vorteilhaft für die Generalisierungsleistung der Modelle sein [DBL10], welches gerade beim passiven Lernen ein wichtiges Kriterium darstellt. Konkrete Messungen dazu werden in Kapitel 6 behandelt.

Samples Passive Lernverfahren erhalten ihre Informationen mittels Beobachtungen – im Folgenden auch Samples genannt. Da das Verhalten eines Systems mittels deterministischer Automaten modelliert wird, definiert jedes Systemverhalten ebenfalls eine reguläre Sprache. Ein Sample verbindet dabei Wörter einer Sprache mit Zusatzinformationen. Wie bereits zuvor nahegelegt, ist es sinnvoll, Eingaben in das System (Spielstrategien) mit der Information zu versehen, ob diese in einem akzeptierenden Zustand (Sieg) oder ablehnenden Zustand (Niederlage) enden.
Als Schreibweise wird $\mathcal{S} = (\mathcal{S}_+, \mathcal{S}_-)$ verwendet, wobei $\mathcal{S}$ das gesamte Sample beschreibt, $\mathcal{S}_+$ die positiv markierten Eingaben und $\mathcal{S}_-$ die negativ markierten Eingaben. Als kompakte Schreibweise können die Eingaben auch in Form eines Tupels geschrieben werden, bei dem jede Eingabe einen Indikator enthält, ob es sich um eine positive oder negative Eingabe handelt. Ein Beispiel dafür wäre $\{(aa, 1), (ab, 0), (bb, 0), (aba, 1), (bba, 1)\}$

Konsistenz Mit DFAs und Samples stehen zwei Mittel zur Verfügung, mit denen Systemverhalten und Beobachtungen dieses Verhaltens modelliert werden können. Es besteht allerdings der Wunsch, diese in gewisser Weise aneinander zu koppeln. So soll beispielsweise das letztendlich inferierte Modell nicht vollständig von dem zuvor beobachtetem Verhalten abweichen. Dazu wird an dieser Stelle der Begriff der Konsistenz eingeführt. Genauer werden die Begriffe der *schwachen Konsistenz* und der *starken Konsistenz* eingeführt.

Definition 3 (schwache Konsistenz) *Ein Automat* $\mathcal{A} = (S, \Sigma, \delta, s, F_A, F_R)$ *ist schwach-konsistent mit einem Sample* $\mathcal{S} = (\mathcal{S}_+, \mathcal{S}_-)$ *wenn gilt:*

- $\forall x \in \mathcal{S}_+ : \delta^*(s, x) \in F_A$ *und*
- $\forall x \in \mathcal{S}_- : \delta^*(s, x) \notin F_A$

Das bedeutet, der inferierte Automat muss für alle positiv-markierten Samples in einem akzeptierenden Zustand terminieren und darf für alle negativ-markierten Samples nicht in einem akzeptierendem Zustand terminieren.
Gegenüber der schwachen Konsistenz verschärft die starke Konsistenz die Bedingung für negativ-markierte Samples.

Definition 4 (starke Konsistenz) *Ein Automat* $\mathcal{A} = (S, \Sigma, \delta, s, F_A, F_R)$ *ist stark-konsistent mit einem Sample* $\mathcal{S} = (\mathcal{S}_+, \mathcal{S}_-)$ *wenn gilt:*

- $\forall x \in \mathcal{S}_+ : \delta^*(s, x) \in F_A$ *und*
- $\forall x \in \mathcal{S}_- : \delta^*(s, x) \in F_R$

Im Gegensatz zur schwachen Konsistenz wird hier gefordert, dass alle negativ-markierten Samples explizit in einem ablehnendem Zustand terminieren.

3.1.1 Modellbasierte Verfahren

Der grundlegende Ansatz modellbasierter Lern-Verfahren besteht darin, die übergebenen Samples zunächst in eine Automatenstruktur zu überführen. Auf Basis dieser Struktur werden dann weitere Operationen durchgeführt, welche die Approximationsgüte an das reale System möglichst verbessern.
Als Vertreter der modellbasierten Lernverfahren wird in diesem Kapitel RPNI (Regular Positive Negative Inference, [OG92]) vorgestellt. RPNI bildet eine Art Algorithmen-Framework, mit welchem sich unter Anderem Ansätze und Algorithmen wie MDL (Maximum Description Length, u.a. [AJ06]) und EDSM (Evidence-Driven State Merging, [LPP98]) umsetzen lassen. Zwischenmessungen haben jedoch gezeigt,

dass MDL und EDSM sich zumindest für Spielstrategien nicht signifikant von dem ursprünglichen RPNI-Ansatz unterscheiden, sodass auf deren Vorstellung verzichtet wird. RPNI wurde jedoch benutzt, um die neuentwickelten Heuristiken HCSM und LCSM (High/Low Coverage State Merging, siehe Abschnitt 3.1.1) zu realisieren. Eine weitere Grundlage dieses Kapitels – unter Anderem in Form von Beispielen – bildet [Hig10].

Vorverarbeitung Wie zuvor erwähnt, werden vor dem eigentlichen Start des Algorithmus' die übergebenen Samples in eine Automatenstruktur überführt. Es existieren verschiedene Ansätze, die beobachteten Informationen in einen Automaten umzuwandeln. RPNI verwendet dazu die Konstruktion eines *Prefix Tree Acceptor*s (PTAs).

Die Methode erstellt zu Beginn einen einelementigen Baum und arbeitet daraufhin die Elemente des Samples einzeln ab. Pro Element beginnt die Methode einen Durchlauf – beginnend in der Wurzel – des Baumes und arbeitet dabei die jeweilige Eingabe zeichenweise ab. Das aktuell eingelesene Zeichen der Eingabe bestimmt dabei, welchen Kind-Knoten die Methode dem Pfad des jeweils aktuellen Baumdurchlaufes hinzufügt. Besitzt ein Knoten für ein eingelesenes Zeichen keinen Kind-Knoten, wird dieser erstellt und in den bereits existierenden Baum eingehängt. Erreicht ein solcher Baumdurchlauf das Ende einer Eingabe wird der aktuelle Knoten entsprechend der Zugehörigkeit der Eingabe entweder als akzeptierend oder ablehnend markiert.

Der $PTA(\mathcal{S})$ für das Beispiel aus dem Paragraphen *Samples* ist in Abb. 3.1 gegeben. Akzeptierende Zustände haben eine doppelte Umrandung, ablehnende Zustände eine breitere.

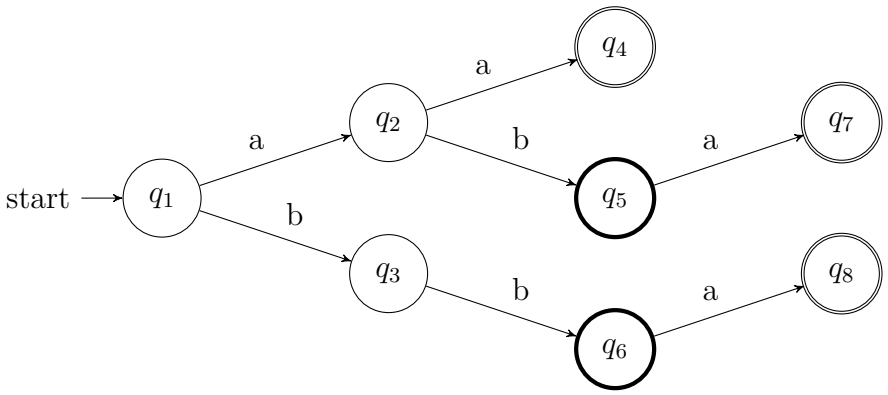


Abbildung 3.1: Ein beispielhafter $PTA(\mathcal{S})$.

Weiterhin ist es möglich, PTAs nur auf Basis der positiven oder der negative Eingaben zu konstruieren. Die Schreibweisen dafür sind im Folgenden $PTA(\mathcal{S}_+)$ respektive $PTA(\mathcal{S}_-)$.

RPNI Die Vorgehensweise des RPNI-Ansatzes wird im Folgenden informell anhand eines Beispieles erklärt, da dieses das beste Kompromiss aus Verständlichkeit und Länge darstellt. Eine formalere Darstellung – beispielsweise in Form von Pseudo-Code – ist unter Anderem in [Hig10] zu finden.
Für den weiteren Verlauf wird das folgende Sample verwendet.

$$\mathcal{S}_+ = \{aaa, aaba, bba, bbaba\}$$
$$\mathcal{S}_- = \{a, bb, aab, aba\}$$

Der Algorithmus beginnt damit, den $PTA(\mathcal{S}_+)$ zu erstellen. Für das zuvor erwähnte Sample ist der entsprechende $PTA(\mathcal{S}_+)$ in Abb. 3.2 dargestellt.

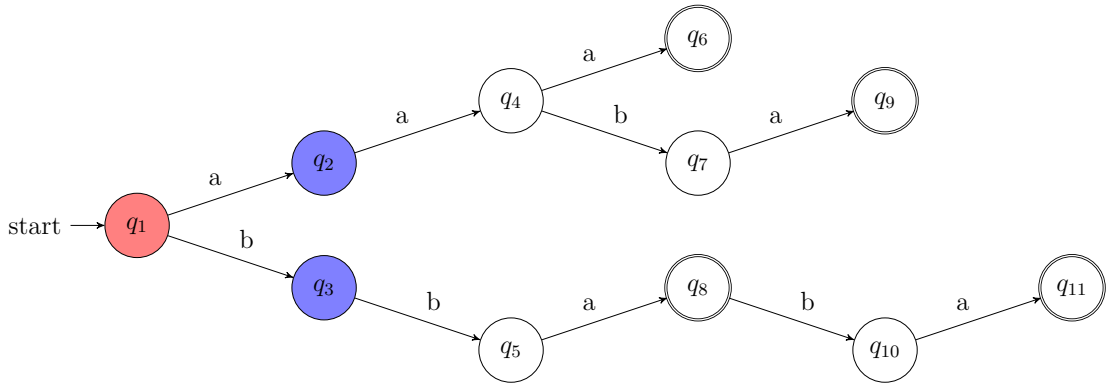


Abbildung 3.2: $PTA(\mathcal{S}_+)$ für das RPNI-Beispiel.

Abb. 3.2 stellt zudem eine weitere Information dar. Der Algorithmus verwaltet für jeden Knoten des Baumes einen eigenen Zustand beziehungsweise Einfärbung. Insgesamt gibt es rote, blaue und weiße Zustände.
Rote Zustände beschreiben Zustände, die von der weiteren Betrachtung des Algorithmus' ausgeschlossen werden, da sie bereits vollständig abgearbeitet sind. Blaue Zustände sind die aktuell betrachteten Zustände, aus denen der Algorithmus einen für seine nächsten Schritte auswählt. Weiße Zustände sind Zustände, welche der Algorithmus noch nicht betrachtet hat und in unmittelbarer Zukunft auch noch nicht betrachtet werden. Nachdem der Algorithmus den $PTA(\mathcal{S}_+)$ erstellt hat, färbt er die Wurzel des Baumes rot und alle zur Wurzel adjazenten Knoten blau ein.

Zunächst macht man sich klar, dass der $PTA(\mathcal{S}_+)$ bereits ein DFA ist, welcher (schwach) konsistent für das gegebene Sample ist. Allerdings akzeptiert dieser DFA nur genau die Elemente aus $\mathcal{S}_+$, sodass dieser DFA keine besonders guten Inferenzeigenschaften besitzt. Der Algorithmus versucht daher, den initialen DFA zu verallgemeinern indem er Zustände sucht, welche sich zusammenfassen lassen. Dazu betrachtet er ein Tupel (q_r, q_b) zweier Knoten, von denen q_r eine rote Markierung und q_b eine blaue Markierung besitzt und fasst diese zusammen, indem die in q_b eingehende Kante – per Konstruktion eindeutig – auf q_r umgebogen wird. Die Reihenfolge in der die Zustände betrachtet werden, delegiert RPNI an eine von außen festgelegt Ordnung auf den Knoten. Für das gegebene Beispiel wird im Folgenden (q_1, q_2) gewählt. Nach der Vereinigung dieser beiden Zustände, entsteht der Automat in Abb. 3.3.

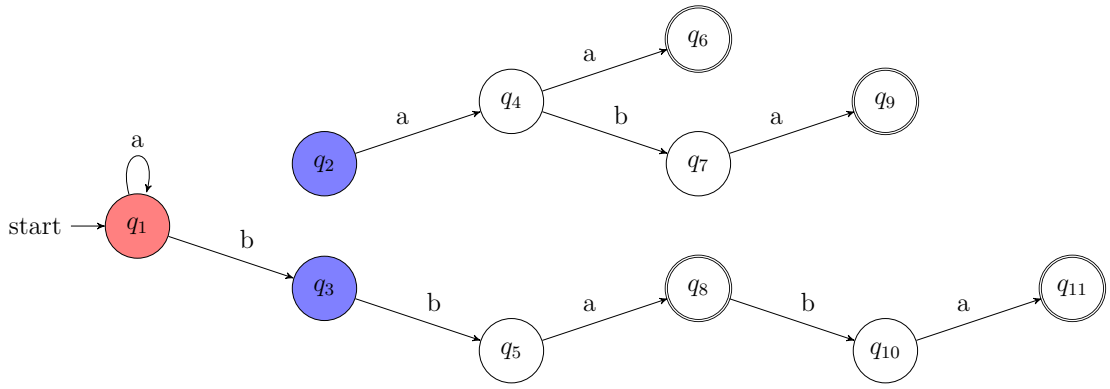


Abbildung 3.3: Zwischenzeitlicher DFA nach Verschmelzung von q_1 und q_2.

Nach der initialen Verschmelzung ergibt sich das Problem, dass der Baum nicht mehr zusammenhängend ist und damit bereits gewonnene Informationen nicht mehr erreichbar sind. Um dieses Problem zu lösen, wird daher im Folgenden der abgetrennte Teilbaum – beginnend mit der Wurzel q_2 und im Folgenden „Teil-Automat“ genannt – in den Hauptbaum – beginnend mit q_1 und im Folgenden „Haupt-Automat“ genannt – eingeflochten. Dazu werden die beiden Ausführungen parallel betrachtet und Informationen bei divergierendem Verhalten in den neuen Haupt-Automaten übernommen.

Die beiden betrachteten Ausführungen starten in den Wurzeln der beiden Teilbäumen, also q_1 und q_2. Die Eingabe b für Knoten q_2 ist undefiniert, sodass diese keine weiteren Informationen bereitstellt und daher nicht weiter betrachtet werden muss – bei Eingabe a hingegen wird in den Zustand q_4 gewechselt. Wird in q_1 ein a eingegeben, verbleibt der Haupt-Automat im Zustand q_1. In einem rekursiven Schritt wird

daher die Informationen des Teilbaums beginnend in q_4 in den Teilbaum beginnend in q_1 integriert.
Wird in q_4 ein a eingegeben, wird der akzeptierende Zustand q_6 erreicht – der Haupt-Automat verbleibt im Zustand q_1. In einem weiteren Rekursionsschritt müsste also q_5 in q_1 integriert werden. Um die Information über den akzeptierenden Zustand zu übernehmen, muss q_1 daher ebenfalls ein akzeptierender Zustand werden. Wird in q_4 ein b eingegeben, wechselt der Teil-Automat von q_4 nach q_7 und der Haupt-Automat von q_1 nach q_3. Damit führt auch dieses Eingabesymbol zu einem weiteren Rekursionsschritt, in dem die Informationen aus q_7 in q_3 integriert werden müssen. q_7 hat erstmals Informationen, welche zu einer Strukturänderung im Haupt-Automaten führen. Für q_3 ist die Eingabe a nicht definiert, sodass beim Einpflegen der Information von q_7 dessen Nachfolger q_9 neuer Nachfolger von q_3 bei Eingabe b wird. Damit sind alle Informationen aus dem abgespalteten Teil-Automaten in den Haupt-Automaten übernommen worden.
Nach der kompletten Verschmelzung der beiden Automaten entsteht der Automat, welcher in Abb. 3.4 dargestellt ist.

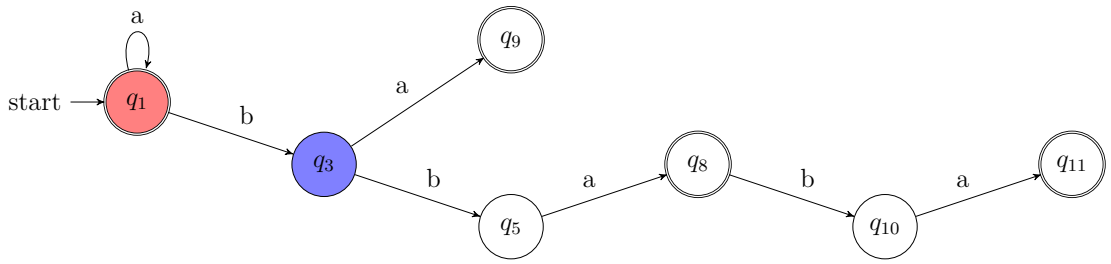


Abbildung 3.4: Automat nach Verschmelzung der beiden Teil-Automaten.

Bei dem neu erhaltenen Automaten fällt auf, dass dieser ebenfalls die Eingabe a akzeptiert. Rückblickend auf das betrachtete Sample ist die Eingabe a jedoch in $\mathcal{S}_-$ enthalten und bildet damit eine Eingabe, die nicht akzeptiert werden sollte. Die Entscheidung, q_1 und q_2 zusammenfassen zu wollen war damit falsch und muss rückgängig gemacht werden. Für q_2 existieren damit keine weiteren rot eingefärbten Knoten, welche mögliche Partner für eine Verschmelzung sind, sodass q_2 selbst eine rote Färbung erhalten kann. Ähnlich wie in der Initialisierungsphase des Algorithmus, werden alle zu q_2 adjazenten Knoten blau eingefärbt. Nach der Rücknahme der Verschmelzung wird mit dem Automaten in Abb. 3.5 weitergearbeitet.
In weiteren Durchläufen des Algorithmus könnte nun versucht werden, der Knoten q_3 mit den Knoten q_1 oder q_2 zu verschmelzen. Diese Versuche würden jedoch ebenfalls aufgrund akzeptierter Gegenbeispiele fehlschlagen, sodass letztendlich q_3

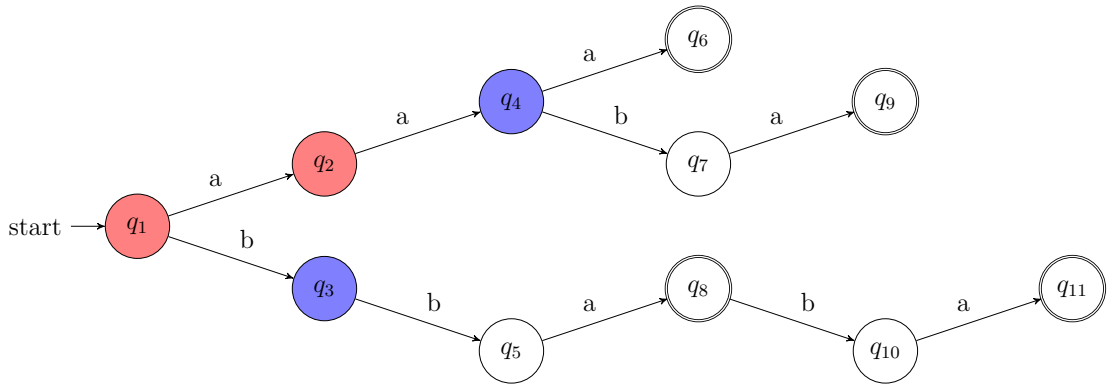


Abbildung 3.5: $PTA(\mathcal{S}_+)$ mit aktualisierten Einfärbungen.

ebenfalls rot eingefärbt werden würde und entsprechend q_5 eine blau Einfärbung erhalten würden. Die weiteren einzelnen Schritte können in dieser Ausarbeitung zugrundeliegenden Literatur [Hig10] in Kapitel 12.4 eingesehen werden. Nachdem alle möglichen Verschmelzungen und anschließende Validierung durchgeführt wurden, gibt der Algorithmus als vorläufiges Ergebnis den Automaten aus Abb. 3.6 zurück.

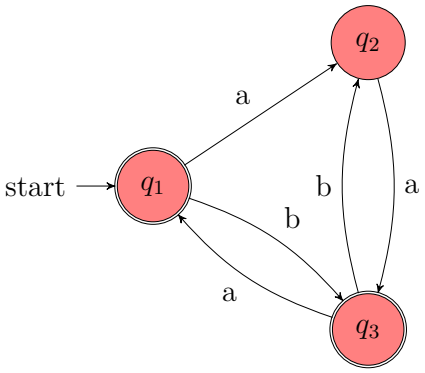


Abbildung 3.6: Vorläufiges Ergebnis des RPNI-Algorithmus'.

Als finaler Schritt muss dieser Automat noch mit den negativen Beispielen des Samples abgeglichen werden, um einen stark-konsistenten Automaten zu erhalten. Das bedeutet, der Zustand q_2 muss als ablehnender Zustand markiert werden.

Zusammenfassung Der RPNI-Algorithmus beschreibt eine sehr allgemeine Vorgehensweise für die Propagation und Zusammenführung von Zuständen. Die grundlegende Idee besteht darin, initial einen sehr spezifischen Automaten zu erstellen – den PTA – und diesen durch eine Reihe von Zustandsverschmelzungen zu generalisieren. RPNI verfolgt dabei eine *Greedy State Merging* Heuristik, da der erste Merge gewählt wird, welcher lediglich die schwache Konsistenz aufrechterhält. Die zuvor angesprochenen MDL und EDSM Ansätze erweitern RPNI dahingehend, dass sie

für die Validierung eines Merges weitere Gütemaße hinzuziehen. Für RPNI lassen sich ebenfalls weitere interessante Eigenschaften – wie beispielsweise Laufzeitverhalten – zeigen. Diese werden jedoch an dieser Stelle ausgelassen, da sie über die Grundlagen des Algorithmus' hinaus gehen und bei Interesse in der entsprechenden Literatur nachgeschlagen werden können.

[H|L]CSM

Mit RPNI und MDL beziehungsweise EDSM sind bisher Algorithmen vorgestellt worden, welche die zwei komplementären Paradigmen des gierigen und vorsichtigen Mergens von Zuständen verfolgen. Direkt damit verbunden ist allerdings auch die Gefahr des Übergeneralisierens beziehungsweise des Untergeneralisierens, welches beides für die Klassifikationsleistung auf bisher unbeobachteten Daten schlecht ist. Daher ist auf Basis von RPNI in dieser Projektgruppe ebenfalls eine eigene Heuristik entwickelt worden, welche versucht, die Kernideen der zuvor genannten Paradigmen zu vereinigen.
Als neues Konzept führt diese Heuristik die sogenannte *coverage* von Zuständen ein. Bei der Erstellung des PTAs wird zusätzlich gespeichert, für wie viele Elemente der Stichprobe der aktuelle Zustand ein Präfix bildet. Es existiert also eine Coverage-Funktion c mit $c(q_a) = |\{w \in S \mid a \text{ is prefix of } w\}|$. Der PTA des RPNI Beispiels würde also die in Abb. 3.7 dargestellten zusätzlichen Informationen beinhalten.

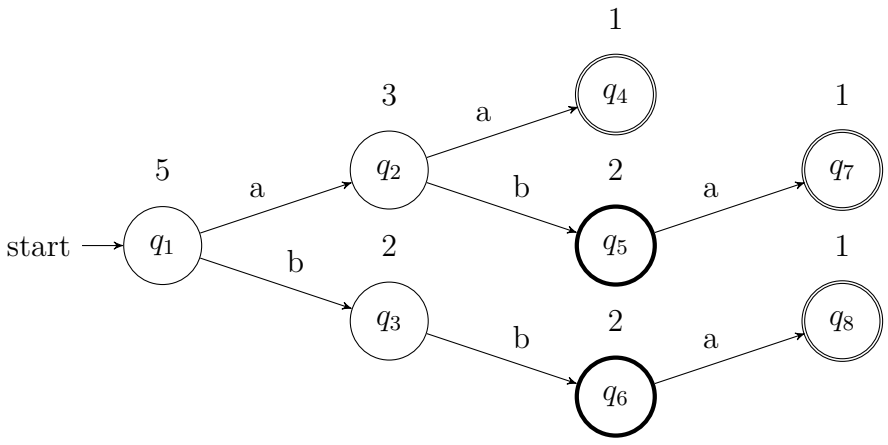


Abbildung 3.7: Ein beispielhafter $PTA(\mathcal{S})$ mit Coverage-Informationen.

Die Idee hinter diesem Maß ist es, etwas über die Sicherheit der Knoten auszusagen, da Knoten mit einer hohen Coverage von vielen Elementen der Stichprobe abgedeckt sind und damit repräsentativ für das beobachtete Verhalten sind. Anhand der Coverage können nun sowohl rote als auch blaue Knoten sortiert werden und in dieser

Reihenfolge für mögliche Merge-Versuche im klassischen RPNI-Kontext betrachtet werden. In Abhängigkeit davon, ob die Knoten aufsteigend – geringste Coverage zuerst – oder absteigend – höchste Coverage zuerst – sortiert, erhält man die Algorithmen LCSM – Low Coverage State Merging und HCSM – High Coverage State Merging.

3.1.2 Nicht-modellbasierte Verfahren

Neben den zuvor vorgestellten modellbasierten Lernverfahren, existiert ebenfalls eine große Menge anderer Lern- und Klassifikationsverfahren [HTF09, Bis06, Nie03]. Grundlegender Unterschied zu den bisherigen Verfahren ist, dass diese nicht direkt auf Automaten arbeiten, sondern ihre Entscheidungen beispielsweise auf Basis von Datenpunkten im $\mathbb{R}^p$ treffen.
Im Rahmen dieser Projektgruppe wird eine Auswahl dieser Verfahren untersucht, da sie einen interessanten, weil sehr alternativen Ansatz für das klassische Automatenlernen darstellen. Des Weiteren ist bei diesen Verfahren nicht direkt ersichtlich, ob sie sich für das Automatenlernen eignen, sodass ebenfalls die empirischen Ergebnisse dieser Verfahren, welche in den weiteren Kapiteln dieser Ausarbeitung vorgestellt werden, von Interesse sind.

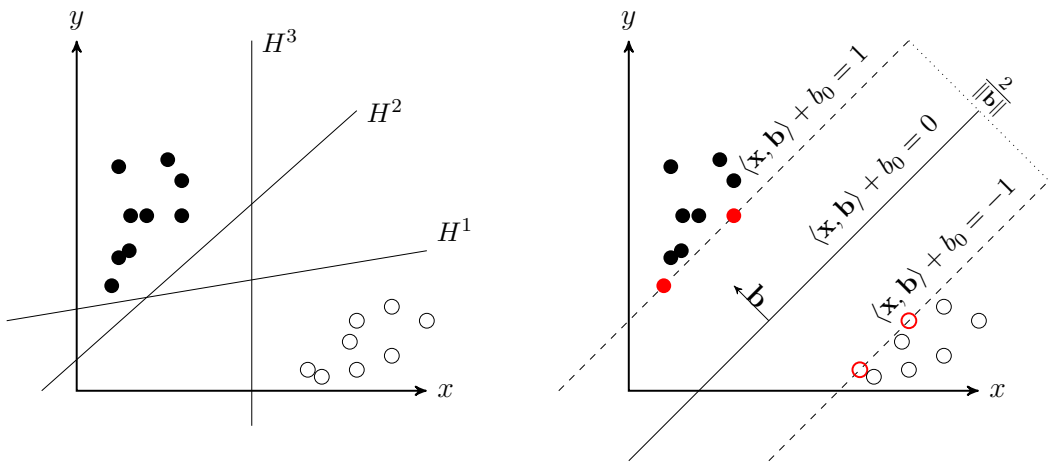
SVM

Support Vektor Maschinen [CV95] – im folgenden durch SVM abgekürzt – beschreiben ein Klassifikationssystem für 2-Klassen Probleme. SVMs arbeiten dabei auf Daten, welche mittels numerischer Vektoren, also Elementen $\mathbf{x} \in \mathbb{R}^p$, dargestellt werden. Als Eingabe erhält eine SVM die Menge $\omega = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_n, y_n)\}$, wobei $\mathbf{x}_i \in \mathbb{R}^p$ das beobachtete Datum und $y_i \in \{-1, 1\}$ das jeweilige Klassen-Label darstellt. Wie sich die bisherigen Trainingsdaten in Form von Automateneingaben in dieses Format überführen lassen, wird in Abschnitt 3.2.2 vorgestellt. Nachdem die SVM ein Modell – welches im Folgenden erklärt wird – gelernt hat, gibt sie für neue, unbekannte Daten die Klasse aus, welche das gelernte Modell vorhersagt.

Funktionsweise Eine SVM trifft ihre Entscheidung mittels einer orientierten Hyperebene, welche den Merkmalsraum $\mathbb{R}^p$ in 2 Teilräume aufteilt. In Abhängigkeit davon, in welchem dieser beiden Teilräume ein Merkmal liegt, wird die entsprechende Klasse gewählt. Eine Hyperebene ist definiert als

$$\begin{aligned} H :=& \{\mathbf{x} \in \mathbb{R}^p | \langle \mathbf{x} - \mathbf{a}, \mathbf{b} \rangle = 0\} \\ =& \{\mathbf{x} \in \mathbb{R}^p | \langle \mathbf{x}, \mathbf{b} \rangle + b_0 = 0\} \end{aligned}$$

wobei $\mathbf{b} \in \mathbb{R}^p$ den Normalenvektor der Ebene, $\mathbf{a} \in \mathbb{R}^p$ den Stützvektor der Ebene und $\langle \mathbf{x}, \mathbf{y} \rangle$ das Skalarprodukt der Vektoren $\mathbf{x}, \mathbf{y}$ beschreibt. Häufig wird ebenfalls die zweite Notation verwendet, welche den Abstand der Hyperebenen zum Nullpunkt in $b_0 = \frac{\langle \mathbf{x}, \mathbf{a} \rangle}{\|\mathbf{b}\|}$ kodiert. Das Skalarprodukt für einen konkreten Punkt $\mathbf{x}' \in \mathbb{R}^p$ liefert den Abstand zur Ebene in Richtung des Normalenvektores. Dieser ist entweder positiv oder negativ, welches genau der Aufteilung in zwei Teilräume entspricht – falls ein Punkt genau auf der Ebene liegt, benötigt es entsprechendes Tie-breaking[1]. Die SVM versucht nun, möglichst *gute* Vektoren $\mathbf{b}, \mathbf{a}$ zu finden. Das Optimierungsproblem sei zunächst für den sehr einfachen Fall einer linear separierbaren Stichprobe veranschaulicht.



(a) H^1, H^2, H^3 **(b)** H^*

Abbildung 3.8: Bestimmen einer Hyperebene mit maximaler Breite

Abbildung 3.8a zeigt eine mögliche Stichprobe von 2-dimensionalen Daten sowie 3 mögliche Trennfunktionen H^1, H^2 und H^3. Jede dieser Trennfunktionen trennt die Stichprobe korrekt, allerdings hat jede Trennfunktion ein unterschiedliches Generalisierungsverhalten. Sind die bisher unbeobachteten Daten beispielsweise positiv linear korreliert, bildet H^2 eine weitaus bessere Trennfunktion als H^3. Eine SVM bestimmt nun jene Hyperebene H^*, welche den minimalen Abstand zu den Stichprobenelementen maximiert, wie in Abb. 3.8b dargestellt ist. Man kann zeigen, dass sich

[1] Tie-breaking beschreibt eine Regel, wie Daten, welche das Modell nicht eindeutig klassifizieren kann, ausgewertet werden

durch entsprechende Skalierung der Parameter zwei parallele Ebenen finden lassen, welche sowohl die Punkte mit minimalem Abstand schneiden als auch den Abstand $\frac{1}{\|\mathbf{b}\|}$ zur Hyperebene H^* haben. Die beiden parallelen Ebenen, welche in Abb. 3.8b durch gestrichelte Linien dargestellt sind, bilden damit eine Art „Korridor“ in dem keine Stichprobenelemente liegen. Die SVM bestimmt nun die Parameter der Ebene so, dass dieser „Korridor“ maximale Breite erlangt.
Die Trainingsphase der SVM lässt damit als constraint-behaftetes quadratisches Optimierungsproblem formulieren:

$$\begin{aligned} &\min_{\mathbf{b}} \frac{1}{2}\|\mathbf{b}\|^2 \\ &\text{s.t.} \\ &\quad y_i(\langle \mathbf{x}_i, \mathbf{b}\rangle + b_0) \geq 1 \quad i = 1, ..., n \end{aligned}$$

Man beachte, dass in dieser Form das quadratische Optimierungsproblem des inversen Problems betrachtet wird. Die Nebenbedigungen sind eine Zusammenfassung der Constraints

$$\begin{aligned} &\langle \mathbf{x}_i, \mathbf{b}\rangle + b_0 \geq 1 && \text{wenn } y_i = 1 \\ &\langle \mathbf{x}_i, \mathbf{b}\rangle + b_0 \leq -1 && \text{wenn } y_i = -1 \end{aligned}$$

Dieses Optimierungsproblem kann nun mittels Lagrange-Multiplikatoren und den Karush-Kuhn-Tucker Bedingungen umgeformt und vereinfacht werden. Das genaue Verfahren wird an dieser Stelle nicht vorgestellt, da es den Rahmen dieses Kapitels überschreiten würde. Detailierte Beschreibungen können unter Anderem in [HTF09, Bis06, Nie03] gefunden werden. Das letztendliche (duale Lagrange-)Optimierungsproblem, welches die SVM lösen muss besteht aus:

$$\begin{aligned} &\max_{\alpha_i} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \langle \mathbf{x_i}, \mathbf{x}_j \rangle \\ &\text{s.t.} \\ &\quad \alpha_i [y_i(\langle \mathbf{x}_i, \mathbf{b}\rangle + b_0) - 1] = 0 \quad i = 1, ..., n \\ &\quad \sum_{i=1}^{n} \alpha_i y_i = 0 \\ &\quad \alpha_i \geq 0 \quad i = 1, ..., n \end{aligned}$$

wobei α_i die Lagrange-Multiplikatoren bezeichnen, welche die SVM verändern kann. Die entsprechenden Parameter für die Ebenengleichung erhält man durch folgende Berechnungen:

$$\mathbf{b} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i$$

b_0 kann – gegeben $\mathbf{b}$ – mittels eines Support Vektors (Element, welches eine parallele Hyperebene schneidet) berechnet werden.

Linear nicht separierbare Stichprobe In der Realität ist die Annahme über die lineare Separierbarkeit der Stichprobe häufig nicht zutreffend. Nichtsdestotrotz lässt sich der zuvor vorgestellte Ansatz erweitern, um ebenfalls mit nicht linear trennbaren Stichproben umzugehen. Dazu werden die Nebenbedingungen des Optimierungsproblems durch das Einfügen von Schlupfvariablen abgeschwächt. Das aktualisierte Optimierungsproblem sieht damit wie folgt aus:

$$\min_{\mathbf{b},\xi_i} \frac{1}{2}\|\mathbf{b}\|^2 + C \cdot \sum_{i=1}^{n} \xi_i$$

s.t.

$$y_i(\langle \mathbf{x}_i, \mathbf{b} \rangle + a_0) \geq 1 - \xi_i \quad i = 1, ..., n$$
$$\xi_i \geq 0 \quad i = 1, ..., n$$

wobei die erste Nebenbedingung wieder eine zusammengefasste Version folgender Constraints ist:

$$\langle \mathbf{x}_i, \mathbf{b} \rangle + b_0 \geq 1 - \xi_i \qquad \text{wenn } y_i = 1$$
$$\langle \mathbf{x}_i, \mathbf{b} \rangle + b_0 \leq -1 + \xi_i \qquad \text{wenn } y_i = -1$$

Die Schlupfvariablen erlauben es also, die Merkmale im Merkmalsraum zu „verschieben“. Wählt man für eine Schlupfvariable einen Wert > 1 ist es damit sogar möglich, Merkmale auf der falschen Seite der Hyperebenen zu modellieren. Durch die Schlupfvariablen alleine, kann eine Stichprobe so verändert werden, dass der Korridor um die Trennfunktion unendlich groß wird. Um diesen Effekt in einem sinnvollen Maße zu beschränken, geht die Summe über die Schlupfvariablen ebenfalls in das zu optimierende Kriterium mit ein. Die Summe der Schlupfvariablen wird dabei mit einem

Faktor C gewichtet, über den man bestimmen kann, ob man eine große Änderung der Daten zugunsten eines großen Korridors (kleines C) erlaubt, oder größeren Wert auf Separierbarkeit legt ($C \to \infty$).
Das umgeformte Optimierungsproblem, welches die SVM lösen muss, hat die angenehme Eigenschaft, bis auf n Constraints, die gleiche Form zu haben:

$$\max_{\alpha_i} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \langle \mathbf{x_i}, \mathbf{x}_j \rangle$$

s.t.

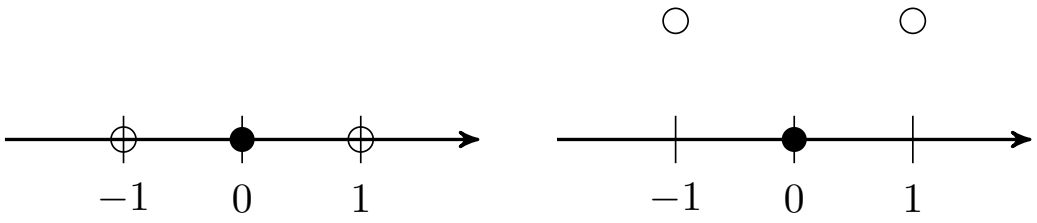
$$\alpha_i [y_i(\langle \mathbf{x}_i, \mathbf{b} \rangle + b_0) - 1] = 0 \quad i = 1, ..., n$$

$$\sum_{i=1}^{n} \alpha_i y_i = 0$$

$$0 \leq \alpha_i \leq C \quad i = 1, ..., n$$

Die restliche Struktur der SVM behält die bereits bekannte Form. Man beachte, dass durch das Hinzufügen von Slack-Variablen keine schwache und starke Konsistenz mehr mit dem Trainings-Samples garantiert werden kann.

Nicht lineare SVMs Eine weitere Möglichkeit, mit nicht linear separierbaren Stichproben umzugehen ist statt (oder auch in Kombination mit) Schlupfvariablen nichtlineare Kernel-Funktionen zu benutzen. Die Idee dahinter lässt sich leicht an dem Beispiel in Abb. 3.9 illustrieren.



(a) XOR-Problem **(b)** Transformiertes XOR-Problem

Abbildung 3.9: Nicht-Lineare Transformation in höher-dimensionalen Raum

In Abb. 3.9a ist ein einfaches XOR-Beispiel im 1-Dimensionalen dargestellt. Man überzeugt sich, dass dieses Punkte nicht linear separierbar sind. Allerdings können die Punkte über eine nicht-lineare Transformation in einen höher dimensionalen Raum transformiert werden, wie es in Abb. 3.9a geschehen ist. Es ist ersichtlich, dass die Punkte nach der Transformation linear trennbar sind. Als Transformation in diesem Beispiel wurde

$$\phi(x) = \begin{pmatrix} x \\ x^2 \end{pmatrix}$$

gewählt.
Auf das eigentliche Optimierungsproblem, welches die SVM lösen muss, hat diese Transformation keine großen strukturellen Einwirkungen, da die SVM lediglich in einem höher-dimensionalen Raum nach einer orientierten Hyperebene sucht. Anstatt die einzelnen $\mathbf{x}_i$ direkt zu verwenden, werden die transformierten $\phi(\mathbf{x}_i)$ verwendet. Das einzige Problem dieses Ansatzes ist die Laufzeit, da in vielen Anwendungen die Zieldimension um hunderte oder gar tausende Dimensionen größer ist als die originale Stichprobe. Um dieses Problem jedoch zu umgehen, kann man sich den sogenannten Kernel-Trick zu Nutzen machen. Dieser basiert auf der Tatsache, dass die SVM zum finden der optimalen Parameter lediglich Skalarmultiplikationen auf Vektoren durchführen muss. Sei im Folgenden die Transformation $\phi(\mathbf{c}) : \mathbb{R}^3 \to \mathbb{R}^6$ betrachtet, mit

$$\phi(c_1, c_2, c_3) = (c_1^2, \sqrt{2}c_1c_2, c_2^2, \sqrt{2}c_2c_3, c_3^2, \sqrt{2}c_1c_3)$$

Betrachtet man nun die Skalarmultiplikation im höher-dimensionalem Raum erhält man

$$\langle \phi(\mathbf{a}), \phi(\mathbf{b}) \rangle = a_1^2b_1^2 + 2a_1a_2b_1b_2 + 2a_2a_3b_2b_3 + a_3^2b_3^2 + 2a_1a_3b_1b_3$$

Durch scharfes Hinsehen kann man erkennen, dass das Ergebnis auch mittels einer geeigneten Kernel-Funktion $K(\mathbf{a}, \mathbf{b})$ berechnet werden kann, mit

$$K(\mathbf{a}, \mathbf{b}) = (\langle \mathbf{a}, \mathbf{b} \rangle)^2$$

Über solch eine Kernel-Funktion kann also das Ergebnis einer Skalarmultiplikation in einem höher-dimensionalen Raum berechnet werden, ohne dass die Vektoren tatsächlich transformiert werden müssen. Die Kernel-Funktion kann dabei direkt in die Zielfunktion des Optimierungsproblems übernommen werden, welches sich zu

$$\max_{\alpha_i} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j K(\mathbf{x_i}, \mathbf{x}_j)$$

ändert. Es existieren eine Reihe von unterschiedlichen Kernel-Funktionen, wie polynomiale Kernel, Sigmoid-Kernel oder RBFs, welche alle unterschiedliche Trennfunktionen darstellen. Damit es also ebenfalls möglich, eine nicht-lineare Trennfunktion zu benutzen um die Daten zu klassifizieren, wobei man sich dennoch vor Augen halten sollte, dass die Kernel-Funktionen lediglich ein Abkürzung für eine lineare Trennfunktion in einem höher-dimensionalen Raum darstellen.

3.2 Realisierung

Um die zuvor erwähnten Lernverfahren nutzen zu können, spielt zum einen die Implementierung der Algorithmen eine wichtige Rolle und zum anderen die konkrete Anbindung an das entwickelte Framework. Im Folgenden wird daher betrachtet, welche Bibliotheken für die verschiedenen Probleme existieren und welche letztendlich für die Umsetzung verwendet wurden sowie deren Integration in das Framework.

3.2.1 Implementierung

Für das passive, modellbasierte Automatenlernen existieren eine Reihe verschiedener Bibliotheken. Im Rahmen der Projektgruppe wurden die Bibliotheken *libalf* sowie *GIToolbox* betrachtet.

libalf [2] ist eine quelloffene Bibliothek für das Automatenlernen und bietet verschiedene Algorithmen zum Automatenlernen an. Die Bibliothek ist in C++ geschrieben, lässt sich aber über das Java Native-Interface aus Java heraus anwenden. Entsprechende Java Interfaces für die native Anbindung werden bereits mitgeliefert, jedoch muss die Bibliothek zunächst plattformspezifisch kompiliert und anschließend in das Programm gelinkt werden. Damit ließ sich die Bibliothek nur schwer in die verwendeten Build-Management Strukturen der Projektgruppe integrieren, da verschiedene Architekturen verwendet werden. Weiter zeigte sich, dass libalf zwar quelloffen, aber nur mäßig dokumentiert ist, was die Verwendung zunehmend erschwerte.

GIToolbox [3] ist eine quelloffene Skriptsammlung zum Automatenlernen für die Matheumgebungen Octave beziehungsweise Matlab. Sowohl für Octave als auch Matlab existieren Möglichkeiten, einzelne Skripte beziehungsweise Funktionen

[2] `http://libalf.informatik.rwth-aachen.de`
[3] `https://code.google.com/p/gitoolbox/`

aus Java heraus aufzurufen [4][5]. Durch die Verwendung einer Skriptsprache kann die Formulierung von Algorithmen vereinfacht werden, da man auf die bereitgestellten Funktionen der jeweiligen Umgebungen zurückgreifen kann. Bei der Integration in das entwickelte Framework erwies sich dies jedoch als Hürde, da zum einen die verwendeten Datenstrukturen der Skripte entsprechend transformiert werden müssen, um in den unterschiedlichen Komponenten verwendet werden zu können und zum anderen das entwickelte Framework die Laufzeiumgebung für die Skripte bereitstellen müsste.

Innerhalb der Projektgruppe wurde daher entschieden, die modellbasierten Lernverfahren eigenständig zu implementieren. Hinsichtlich der Interaktion mit den anderen Komponenten des Frameworks erlaubt diese Entscheidung die Integration der *AutomataLib* (siehe unten) als gemeinsame Grundlage der Automatenmodelle. Dadurch kann eine nahtlose Kommunikation zwischen den Komponenten erreicht werden, da alle Komponenten die gleiche Bibliothek verwenden. Desweiteren erlaubt eine eigenständige Implementierung einen besonderer Fokus auf Aspekte wie Speicherverbrauch und Parallelisierung zu legen, welches die Performanz der Lernverfahren verbessert.

AutomataLib [6] ist eine in Java geschriebene Open-Source-Bibliothek für Automaten, Graphen und Transitionssysteme. Die AutomataLib ist das Automatenframework der LearnLib (siehe Abschnitt 4.2.1) und wird ebenfalls als Modellgrundlage in der Inferenz-Komponente (siehe Kapitel 5) verwendet. Durch die bereitgestellten Interfaces und einer Reihe von vorimplementierten Automatenmodellen, lassen sich dadurch sehr einfach Automaten erstellen, welche unter den einzelnen Komponenten kompatibel sind. Desweiteren bringt die AutomataLib eine Reihe von Graphenalgorithmen wie beispielsweise „Single Source Shortest Path“, „All Pairs Shortest Path“, „Breadth First Search“ oder Minimierungsalgorithmen mit, welche für die Aufgaben der einzelnen Komponenten verwendet werden können. Aus diesen Gründen wurde entschieden, die AutomataLib als Modellgrundlage in der Projektgruppe zu verwenden.

Rapidminer [GMB14a] Für die Umsetzung der SVM wurde das Machine-Learning Framework Rapidminer verwendet. Rapidminer ist eine in Java geschriebene Umgebung zur Lösung maschineller Lern- und Data-Mining-Aufgaben. Es

[4] `http://www.cs.virginia.edu/~whitehouse/matlab/JavaMatlab.html`
[5] `https://kenai.com/projects/javaoctave/pages/Home`
[6] `http://www.automatalib.net`

wurde ursprünglich unter dem Namen YALE (Yet Another Learning Environment) am Lehrstuhl 8 [Dor14] der TU Dortmund entwickelt, ist aber seit längerem unter dem Namen RapidMiner der Firma Rapid-I als eigenständiges Produkt verfügbar. Rapidminer bietet Zugriff auf viele gängige statistische Lernverfahren sowie Algorithmen zur Datenvor- und Nachbearbeitung. Die Programmierung erfolgt in der Regel mittels grafischer Programmierung über den mitgelieferten Editor. Einzelne Operatoren – zu denen unter Anderem auch die SVM zählt – können jedoch ebenfalls direkt aus Java-Code heraus aufgerufen werden. Als besonders nützlich erweist sich dabei, dass Rapidminer damit eine Java-Schnittstelle für eine SVM bietet – intern verwendet Rapidminer für die SVM die C-Bibliothek *libsvm*.

Für die Verwendung der SVM in einer automatenlastigen Umgebung wurde ein SVM-Wrapper geschrieben, welcher Anfragen an einen Automaten auf die SVM abbildet. An einigen Stellen können diese Anfragen nicht vollständig beantwortet werden, da eine SVM beispielsweise über keine Zustände verfügt, dessen Gesamtanzahl zurückgegeben werden könnte. Essentielle Anfragen anderer Komponenten, ob beispielsweise eine Eingabe akzeptiert oder abgelehnt wird, können jedoch in ausreichendem Maße behandelt werden.

3.2.2 Anbindung an das Framework

Zuletzt wird behandelt, wie die Algorithmen an das Framework angebunden werden. Im Falle der SVM bleibt die zusätzliche Frage, wie sich das numerische Klassifikationsverfahren für das Automatenlernen nutzen lässt.

Verarbeiten von Samples Wie zuvor erwähnt, besteht das grundlegende Konzept des passiven Automatenlernens daraus, auf Basis von gelabelten Trainingsdaten ein Automatenmodell zu erstellen. Um diesen Anwendungsfall zu bewältigen, erhält die passive Lernkomponente einen Parser für CSV-Dateien, über den die Trainingsdaten spezifiziert werden können. Dieser liest die bereitgestellten Beobachtungen ein und transformiert sie zu Samples, welche den Lernalgorithmen übergeben werden können. Betrachtet man, was der Benutzer des Frameworks für die anderen Komponenten bereitstellen muss, nimmt das GUL jedoch eine weitaus größere Rolle ein. Um den Workflow des passiven Lernens zu verschlanken, existiert ebenfalls ein GULWrapper, welcher anstatt mittels einer CSV-Datei, mittels eines GULs Samples erstellen kann. Dazu werden zufällige Eingaben für das GUL erstellt und auf Basis der akzeptierenden und ablehnenden Symbole, entsprechend gelabelte Trainingsdaten erzeugt.

Der Benutzer kann dabei zum einen bestimmen, wie viele Trainingsdaten erzeugt werden sollen und zum anderen angeben, ob diese in eine CSV-Datei für zukünftige Verwendung exportiert werden soll.

Numerische Klassifikation Die SVM – oder numerische Klassifikationsverfahren allgemein – erwartet Eingaben aus einem Raum $\mathbb{R}^p$. Samples bestehen im Allgemeinen jedoch aus Sequenzen von Eingabesymbolen, sodass diese für für die SVM vorverarbeitet werden müssen. Dazu wird zunächst eine Ordnung auf den Eingabesymbolen definiert, sodass für ein Eingabesymbol über seine Ordinalzahl ein numerischer Repräsentant gewählt werden kann. Das Framework wählt im Falle von CSV-Dateien eine Ordnung gemäß des Beobachtungszeitpunktes aus. Das bedeutet das erste eingelesene Eingabesymbol erhält die Ordinalzahl 1, das zweite die Ordinalzahl 2, etc. Im Falle des GUL-Wrappers wird die Ordnung gewählt, die durch das GUL vorgegeben wird.
Im Allgemeinen können Traces eine unterschiedliche Länge besitzen, die SVM erwartet jedoch Eingaben mit konstanter Dimensionalität. Daher wird zunächst die Dimension p durch die Länge des längstens Traces bestimmt, sodass jedes Eingabesymbol einer eigenen Dimension entspricht. Die restlichen Dimensionen kürzerer Traces werden vor der Übergabe an die SVM mit den Werten 0 aufgefüllt, sodass *unbekannte* Daten/Dimension bei dem inneren Produkt (siehe duales Lagrange-Problem) keinen Einfluss haben. Wird eine Anfrage an die SVM mit größerer Dimensionalität gestellt, werden die Eingabesymbole ab Positionen $p + 1$ ignoriert.
Mit dieser Transformation können Traces für beliebige numerische Klassifikationsverfahren verwendet werden. Lernt man mit dem gewählten Klassifikator lediglich ein 2-Klassen Problem (wie in diesem Fall mit der SVM), kann die vorhergesagte Klasse direkt auf die Semantik von akzeptierenden beziehungsweise ablehnenden Zuständen der DFAs abgebildet werden.

4 | Aktives Lernen

Aktives Automatenlernen wurde im Rahmen dieser Projektgruppe verwendet um das Verhalten von künstlichen Intelligenzen auf analysierbare Modelle abzubilden. Diese Modelle können anschließend dazu verwendet werden um künstliche Intelligenzen anhand ihres Verhaltens zu identifizieren und Gewinnstrategien gegen sie zu erstellen.

Um ein besseres Verständnis für die aktive Lernkomponente entwickeln zu können wird im folgenden kurz der theoretische Hintergrund erläutert. Auf dieser Basis wird die Implementierung und Umsetzung der Komponente vorgestellt, bevor die Anbindung und Interaktion mit dem restlichen Framework beschrieben wird.

4.1 Theoretische Grundlagen

Im Gegensatz zu passivem Lernen, bei dem ein Model anhand von *Samples* (welche bspw. *Traces* des Systemverhalten entsprechen) extrahiert wird (siehe Kapitel 3), wird beim aktiven Lernen mit dem System interagiert. Den Grundstein hierfür legte der L^* Algorithmus von Dana Angluin [Ang87], der ursprünglich dafür entwickelt wurde eine unbekannte reguläre Sprache anhand ihrer enthaltenen und nicht enthaltenen Worte exakt und effizient zu identifizieren. Das zu lernenden System wird auch als System Under Learning (SUL) bezeichnet und entspricht in unserem Szenario der *GUL*-Klasse. Das theoretische Fundament hierfür liefert die Nerode-Relation [Ner58] bzw. der Satz von Myhill-Nerode, welche eine Charakterisierung der Zustände des minimalen DFAs einer Sprache als Äquivalenzklassen auf Σ^* erlaubt. Zwei Präfixe entsprechen hierbei genau dann dem gleichen Zustand, wenn das Verhalten für alle „Zukünfte“ (Suffixe) gleich ist. Es gibt allerdings noch folgende Restriktionen, die von einem System zwingend eingehalten werden müssen, damit ein Modell von diesem mittels aktiven Lernens inferiert werden kann:

- das System muss ein *Interface* oder eine Abstraktion bereitstellen, über das mit ihm kommuniziert werden kann und seine Reaktion auf Aktionen muss sichtbar sein
- ein *endliches* festes Eingabealphabet[1] für das Interface muss bekannt sein
- das Verhalten des Systems muss *deterministisch* sein und Anfragen dürfen sich nicht gegenseitig beeinflussen. Insbesondere dürfen sich Ausgaben für identische Eingabesequenzen zu verschiedenen Zeitpunkten nicht unterscheiden

Um die grundlegenden Konzepte eines aktiven Lernalgorithmus genauer zu erklären, folgt nun eine genauere Erläuterung der gerade genannten Punkte im Kontext von L^*.

Endliches Eingabealphabet Ein häufiges Hindernis ist, dass das Eingabealphabet durch Verwendung von Daten wie z.B. Userdaten (Login und Passwort) unbeschränkt ist. Um das Verhalten eines solchen Systems in einen endlichen Automaten abbilden zu können, ist es zwingender maßen notwendig, das Eingabealphabet zu abstrahieren, um es für den Lerner endlich erscheinen zu lassen. Das Ausgabealphabet muss hingegen so abstrahiert werden, dass es deterministisch in Verbindung mit dem Eingabealphabet erscheint. Man unterscheidet deshalb zwischen einem konkreten und einem abstrakten Eingabe- bzw. Ausgabealphabet. Am konkreten Beispiel müsste eine Login-Aktion bereitgestellt werden, die mit validen Userdaten ausgestattet ist und danach alle Aktionen für eingeloggte User ermöglicht. Das erstellte Modell ist daher immer eine Approximation des Zielsystems. Diese Abstraktionen sind offensichtlich individuell und müssen in praktischen Anwendungen für jedes Problem neu definiert werden. Zwar gibt es Ansätze für eine automatische Inferenz der Abstraktion [HSM11], diese sind jedoch weiterhin zumindest auf Gegenbeispiele mit den korrekten Userdaten angewiesen.
Die für einen Lernprozess notwendigen Rahmenbedingungen wurden von Dana Angluin im sogenannten Minimally Adequate Teacher (MAT) Modell [Ang87] etabliert, welches die mögliche Kommunikation mit einem sogenannten *Teacher* definiert. Dieses besteht im Wesentlichen aus zwei Arten von Anfragen (*queries*) an den MAT.

Definition 5 (Membership Query) *Eine* Membership Query *ist eine Anfrage an einen MAT mit einer Eingabesequenz, auf die die Ausgabe des SULs zurückgeliefert wird, nachdem alle Symbole der Eingabesequenz abgearbeitet wurden.*

Definition 6 (Equivalence Query) *Eine* Equivalence Query *ist eine Anfrage an den MAT mit einer vom Algorithmus gelernten Hypothese des SUL. Sie liefert ent-*

[1] Aktionen die auf dem System ausgeführt werden können

weder die Ausgabe zurückliefert, dass die Hypothese äquivalent zum SUL ist oder ein Gegenbeispiel in Form einer Eingabesequenz, die unterschiedliche Ausgaben erzeugt.

Der grobe Verlauf eines aktiven Lernalgorithmus lässt sich nun in zwei alternierende Phasen aufteilen. In der ersten Phase werden *Membership Queries* an den MAT gestellt um eine Hypothese des Systems zu erstellen. Die zweite Phase beginnt sobald der Algorithmus genug Informationen gesammelt hat, um eine konsistente Hypothese zu erzeugen. Diese Hypothese wird anschließend benutzt um zu testen, ob das jetzige Modell schon mit dem zu lernenden System übereinstimmt (*Equivalence Query*). Ist dies nicht der Fall, wird anhand eines Gegenbeispiels in Form einer Sequenz von Eingabesymbolen (die in beiden Systemen unterschiedliche Ausgaben verursachen) die Hypothese verfeinert, indem mindestens ein neuer Zustand der Hypothese hinzugefügt wird. Diese beiden Phasen werden so lange alterniert, bis die Hypothese das SUL korrekt beschreibt.
Obwohl dies in der Theorie in polynomieller Zeit möglich ist das Modell zu inferieren, treten in der Praxis oftmals Probleme auf, die sich darauf zurückführen lassen, dass es in der Regel keinen MAT für das SUL gibt. Es bedarf deshalb einer praktikablen Umsetzung des MAT Modells für das Erlernen von realen Systemen.

Praktische Adaption des Minimally Adequate Teacher Ein MAT nimmt in einem aktiven Lernalgorithmus die zentrale Rolle des Kommunikators zwischen dem SUL und dem Lernalgorithmus ein. Es wird angenommen, dass er für den Algorithmus *Membership Queries* und *Equivalence Queries* korrekt beantworten kann. Da diese beiden Typen von Anfragen sehr unterschiedlich sind, kann ein MAT auch in zwei alleinstehende Komponenten geteilt werden:

- einem *Membership Oracle*, welches die *Membership Queries* beantwortet
- und einem *Equivalence Oracle*, welches entsprechend die *Equivalence Queries* beantwortet

Da Anfragen laut Definition unabhängig voneinander getätigt werden müssen, muss zwischen dem Auswerten der einzelnen Queries ein sogenannter *reset* erfolgen, der das System in seinen Ursprungszustand zurück versetzt. Ein weiteres Hindernis bei realen Systemen ist außerdem dass diese in Echtzeit laufen. Wird mit dem laufenden System interagiert, bedeutet das, dass man nicht sofort eine Antwort bekommt. Die Ausgaben treten stattdessen mit einer gewissen Verzögerung auf, die im Algorithmus abstrahiert werden muss. Es sollte also eine gewisse Zeit gewartet werden, bis sicher alle Konsequenzen einer bestimmten Eingabe eingetreten sind bevor eine neue

Eingabe erfolgen kann. Die Ausgaben müssen korrekt auf die Eingaben abgebildet werden, damit kein falsches Verhalten angenommen wird.
Die Annahme der Existenz eines *Equivalence Oracle* machte es für Angluin möglich, die Korrektheit des aktiven Lernens sehr kurz und elegant zu beweisen. Es ist jedoch leicht ersichtlich, dass die praktische Umsetzung des *Equivalence Oracle* bei einem unbekannten System nur approximativ möglich ist, was ein Hindernis für die Anwendbarkeit aktiver Lernalgorithmen in der Praxis darstellt.
Üblicherweise approximiert man *Equivalence Queries* durch mehrere *Membership Queries*, bspw. durch zufälliges Samplen von Wörtern oder Random Walks. Diese Verfahren sind jedoch nicht vollständig, d.h. die Tatsache dass kein Gegenbeispiel gefunden wurde sagt nichts über die Äquivalenz zwischen Hypothese und SUL aus. Das Modell, welches während des aktiven Lernvorgangs erstellt wird, ist daher niemals garantiert korrekt, da nicht sichergestellt werden kann, dass das *Equivalence Oracle* alle Zustände des SUL durch Sampling erreicht.
Da die interne Struktur der Lernalgorithmen für unseren Anwendungszweck unerheblich ist, verzichten wir an dieser Stelle darauf und verweisen den Leser auf [Ang87] und [SHM11].

4.2 Realisierung

Im folgenden wird die Realisierung der aktiven Lernkomponente vorgestellt. Dabei wird zuerst auf das unterliegende Framework eingegangen und die schon vorhandenen Komponenten werden erläutert. Anschließend werden kurz die Ideen der ersten Implementierung aus dem ersten PG-Semester vorgestellt und die Einschränkungen erläutert, die in der zweiten Implementierung, welches die finale Implementierung dieser PG darstellt, behoben wurden. Im letzten Abschnitt wird noch auf die Anbindung an das restliche Framework eingegangen, welche unter anderem Speichermethoden und Konfigurationsdateien beinhaltet.

4.2.1 Verwendetes Framework

Für aktives Automatenlernen stehen zwei frei verfügbare Frameworks zur Auswahl. Zum einen *LearnLib*[2], ein in Java geschriebenes Open-Source-Framework ausschließlich für aktives Automatenlernen, und zum anderen libalf [3], ein in C++ geschriebenes Open-Source-Framework für passive und aktive Lernalgorithmen. Da *LearnLib*

[2] `http://www.learnlib.de`
[3] `http://libalf.informatik.rwth-aachen.de`

nicht nur in derselben Programmiersprache wie unser Framework geschrieben ist, was die Anbindung deutlich erleichtert, sondern bezüglich der aktiven Lernalgorithmen auch eine größere Auswahl bietet, ist die Wahl schnell auf *LearnLib* gefallen.

LearnLib stellt neben vielen verschieden Lernalgorithmen für Mealy-Automaten und DFA eine Vielzahl von Konfigurationsmöglichkeiten für diese bereit. Das Framework ist so konstruiert, dass alle Lernalgorithmen für Mealy-Automaten (bzw. für DFA) ein einheitliches Interface implementieren. Dasselbe gilt für *Membership Oracles*, welche ineinander geschachtelt werden können, und *Equivalence Oracle*. Dies hat den Vorteil, dass ein Lernvorgang sehr abstrakt formuliert werden kann und die Orakel und implementierenden Algorithmen in diesem Konstrukt beliebig ausgetauscht werden können. Der einzige Unterschied sind oft die unterschiedlichen Parameter der Konstruktoren für Lernalgorithmen und Orakel, was sich leicht durch ihre unterschiedlichen Datenstrukturen und Funktionsweisen erklären lässt. Eine Abstraktion von den individuellen Konstruktoren ist daher leider nicht möglich.

Um vom zu lernenden System ebenfalls abstrahieren zu können, wurde die SUL-Schnittstelle (entspricht dem in Abschnitt 2.3 vorgestellten *GUL*) eingeführt. Diese definiert alle nötigen Methoden um die in Abschnitt 4.1 vorgestellten Bedingungen zu erfüllen (soweit sie über ein Interface erzwungen werden können).

Die von den Lernalgorithmen erzeugten Modelle werden auch innerhalb des Frameworks weiter verwendet. Um auf einen einheitlichen Standard zu kommen und keine Datenstruktur neu implementieren zu müssen, werden die Datenstrukturen des *AutomataLib* Frameworks (siehe Abschnitt 3.2.1) verwendet. Die Datenstrukturen werden bereits von den in der *LearnLib* implementierten Lernalgorithmen verwendet und müssen deshalb nicht konvertiert werden. Es sind keine Modifikationen der Datenstruktur notwendig. Da die gelernten Modelle gespeichert werden müssen, wurde für das `MealyMachine`-Interface eine Serialisierung nach XML implementiert.

4.2.2 Erste Implemenierung und Allgemeines

Da im ersten Semester das Framework *LearnLib* zuerst kennengelernt werden musste, wurden für das Spiel *Connect Four* zwei unterschiedliche Ansätze auf Basis der *LearnLib* implementiert. Diese unterscheiden sich hauptsächlich in der Implementierung des sogenannten *Test-Driver* [MIH+12], welcher in Abbildung Abbildung 4.1 dargestellt ist.

Den Komponenten sind dabei folgende Aufgaben zugeordnet:

Mapper Der Mapper ist verantwortlich für das Übersetzen vom abstrakten zum konkreten Alphabet und umgekehrt (vgl. Abschnitt 4.1)

Wertekontext Wenn die Symbole des konkreten Alphabets Datenwerte erfordern, bedient sich der Mapper des Wertekontexts um diese zu erfragen. Dieser ist in der Lage, Werte zu speichern und dem Mapper zur Verfügung zu stellen, wenn er diese benötigt. Dabei kann es sich um konstante Werte handeln, die zum Aufbau der Testverbindung nötig sind (wie zum Beispiel Login-Daten), aber auch um Werte, die aus der Ausgabe des SUL ausgelesen werden (zum Beispiel eine nach dem Login zurückgegebene Session-ID).

Proxy Der Proxy ist die direkte Schnittstelle zum SUL (in unserem Fall das SUL-Interface). Er macht direkten Gebrauch von der Zielsprache, zum Beispiel von Java-Methoden.

Eine mögliche Umsetzung ist dabei die Implementierung der SUL-Klasse der *LearnLib*, in welcher alle Komponenten des Testdrivers selbst umgesetzt werden müssen. Eine andere Möglichkeit ist die Benutzung der *Driver Architektur*, in der eine separate Klasse für den *Mapper* bereits vorhanden ist und die Eingaben aus ausführbaren Symbolen bestehen. Der Code wird an dieser Stelle also vereinfacht, da eine bestehende Struktur benutzt werden kann. Auf genauere Ausführungen wird an dieser Stelle verzichtet, da die Implementierungen der *GUL*-Klasse (welches dem SUL entspricht) in Abschnitt 2.3 genauer vorgestellt wird.

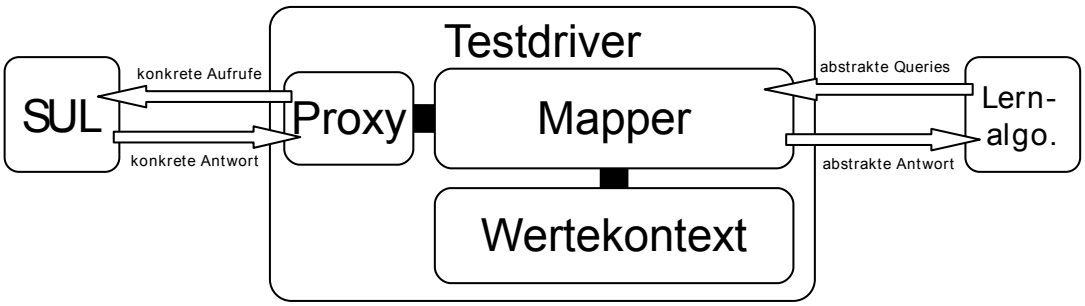


Abbildung 4.1: Graphische Darstellung eines Testdrivermodells [MIH+12]

4.2.2.1 Allgemeiner Lernablauf

Der allgemeine Ablauf eines Lernvorgangs ist hingegen unabhängig von der Realisierung der SUL Schnittstelle. Er folgt immer demselben Schema, welches in Abbildung 4.1 dargestellt ist, und wurde auf ähnliche Weise umgesetzt.

Algorithmus 4.1 Grundlegender Lernablauf

Eingabe: input alphabet *alphabet*, SUL adapter

Ausgabe: learned Mealy machine $\mathcal{M}$

```
1: learner = createLStar(alphabet, SUL)
2: learner.start()
3: eqOracle = createRandomWords(alphabet, SUL)
4: while ( ce = eqOracle.findCE(learner.hypothesis()) != null) do
5:     learner.refine(ce)
6: end while
7: return  M
```

Mit einer Anbindung an das zu lernende System und dem zugrundeliegenden Eingabealphabet kann so ein beliebiger Lernalgorithmus des Frameworks erzeugt werden. Mit ihm wird dann eine initiale Hypothese des Systems erstellt. Anschließend muss ein geeignetes *Equivalence Oracle* erstellt werden, welches die Hypothese mit dem Zielsystem vergleicht und bei festgestellter Ungleichheit eine trennende Sequenz, das Gegenbeispiel *ce*, zurückliefert. Solange ein Gegenbeispiel gefunden werden kann, wird der Lernalgorithmus mit dessen Hilfe die Hypothese verfeinern.

Obwohl man in der *LearnLib* die *Membership Oracle* und den internen Automatentyp zur Speicherung der Hypothese beliebig wählen kann, haben wir uns im Rahmen der PG für Mealy-Automaten entschieden, da diese besonders geeignet sind um reaktive Systeme, wie künstliche Intelligenzen in *Connect Four* und *Chain Reaction*, abzubilden.

4.2.2.2 Vor- und Nachteile

Diese Version einer Lernkomponente war sehr schnell zu implementieren und designtechnisch nicht besonders aufwendig. Das sorgte auch für eine sehr übersichtliche Implementierung in weniger als 1000 Zeilen Code. Lernalgorithmus oder Orakel konnten auf Code-Ebene schnell und einfach ausgetauscht werden. Aufwendigere Änderungen, wie zum Beispiel eine komplizierte Schachtelung der Orakel, waren allerdings nicht elegant umzusetzen und machten den Code schnell unübersichtlich, zumal immer nur eine Implementierung existierte und keine eigenständige Konfiguration in bspw. Subklassen für unterschiedliche Ausführungen möglich war. Stattdessen wurde der Code kopiert und angepasst. Die entspricht nicht dem Gedanken eines Frameworks, welches von Benutzern einfach verwendet werden kann.

4.2.3 Finale Implementierung

Die erste Implementierung war nur schwer konfigurierbar und für Außenstehende sehr unverständlich. Für die Neuimplementierung wurde daher beschlossen vom eigentlichen Lernablauf zu abstrahieren und nur noch die Komponenten zu definieren, die benutzt werden sollen. Der Lernablauf an sich sollte dann automatisch aus den definierten Komponenten zusammengesetzt werden.
Wie aus Abschnitt 4.1 hervorgeht sind die variablen Komponenten eines aktiven Lernalgorithmus das zu lernende System, der Lernalgorithmus, unterschiedliche *Membership Oracles* (vgl. Abschnitt 4.2.3.1) und die *Equivalence Oracles*, die benutzt werden sollen. Bei einer genaueren Analyse der zu lernenden Spiele im ersten Semester ist außerdem noch der Wunsch auf Unterbrechung und vorzeitigen Abbruch des Lernvorgangs geäußert worden. Grund dafür waren nicht endende oder sehr lange dauernde Lernvorgänge.
Da unterschiedliche Lernalgorithmen auch unterschiedliche interne Datenstrukturen verwalten, ist es nicht einfach möglich eine generelle Speichermethode für alle Lernalgorithmen zu schreiben, damit diese unterbrochen und zu einem späteren Zeitpunkt fortgeführt werden können. Es müsste für jeden Algorithmus und jede Datenstruktur eine eigene Serialisierung implementiert werden. Um dieses Problem zu umgehen wird stattdessen für jeden Lernvorgang ein Cache angelegt, welcher die Ein- und Ausgaben aller gestellten *Membership Queries* speichert. Der Cache wird vor alle *Membership Oracles* geschaltet und Anstelle der internen Datenstruktur gespeichert wird. So lässt sich zwar nicht der aktuelle Fortschritt speichern, allerdings kann bei einem Neustart zuerst beobachtetes Verhalten aus dem Cache gelernt werden, was je nach SUL eine deutliche Beschleunigung bedeutet.
Um einen Lernvorgang vorzeitig abzubrechen sobald ein bestimmtes Kriterium erreicht ist, wurde das `TerminationChecker`-Interface eingeführt. Dies enthält eine Methode `isTerminating`, welche eine Hypothese des Lernalgorithmus übergeben bekommt. Diese Hypothese kann dann auf gewünschte Eigenschaften untersucht werden und der Lernalgorithmus gegebenenfalls abgebrochen werden.
Ein verallgemeinerter Algorithmus, der selbstständig eine Verkettung von *Membership Oracles* erstellt, mehrere *Equivalence Oracle* befragt und die Hypothese regelmäßig auf erreichte Ziele untersucht, ist in Algorithmus 4.2 aufgeführt.
Der Algorithmus bekommt alle benötigten Komponenten für einen Lernvorgang übergeben, die sich aus dem SUL, dem Lernalgorithmus, einem evtl. schon vorhandenem Cache, evtl. einer Reihe bereits befundener Gegenbeispiele, der zu benutzenden *Membership Oracles*, der zu benutzenden *Equivalence Oracle* und dem zum SUL

Algorithmus 4.2 Flexibler Lernalgorithmus

Eingabe: GUL *gul*, Learning algorithm *learner*, Cache *cache*, Counterexamples *ces*, Membership Oracles *mqs*, TerminationChecker *tc*, Equivalence Oracles *eqs*

Ausgabe: learned Mealy machine M

```
1: Membership Oracle sulOracle ← createOracleChain(mqs, cache)
2: learner.startLearning(sulOracle)
3: for all ( ce ∈ces) do
4:    learner.refineHypothesis(ce);
5: end for
6: for all ( eqOracle ∈ eqs) do
7:    CounterExample counterexample;
8:    while (counterexample ← eqOracle.findCounterExample(
      learner.getHypothesisModel()) ≠ NULL do
9:       learner.refineHypothesis(counterexample);
10:      if tc.isTerminating(learner.getHypothesisModel()) then
11:         return learner.getHypothesisModel()
12:      end if
13:   end while
14: end for
15: return learner.getHypothesisModel();
```

gehörenden `TerminationChecker` besteht. Da *Membership Oracle* in der *LearnLib* schachtelbar sind, wird aus ihnen zuerst in der `createOracleChain()`-Methode ein einziges SUL zusammengesetzt, welches dann vom Lernalgorithmus befragt wird. Da der Lernalgorithmus unterbrochen werden können soll, wird zuerst die Schnittstelle zum zu lernenden System hinter einen Cache gehängt, welcher so alle an das reale System gestellten Anfragen mitschneiden kann. Dieser Cache wird dann in weitere Orakel geschachtelt (in vorgegebener Reihenfolge), die als *Membership Oracle* dem Algorithmus übergeben wurden. Anschließend kann der ausgewählte Lernalgorithmus eine erste Hypothese erstellen, die ggf. aus dem Cache gelernt werden kann. Anschließend werden von anderen Komponenten gefundene Gegenspiele dazu benutzt die Hypothese zu verfeinern. Dies geschieht bspw. wenn das gelernte Modell für die Inferenz (vgl. Kapitel 5) zu ungenau war. Im Anschluss an diese vorgestellten Maßnahmen beginnt der aus der ersten Implementierung schon bekannte Lernablauf. Der gravierende Unterschied ist, dass alle übergebenen *Equivalence Oracle* der Reihe nach abgearbeitet werden. Das bedeutet, dass ein Orakel so lange benutzt wird

wie es noch Gegenbeispiele findet, bevor zum nächsten Orakel übergegangen wird. Dies ermöglicht es zum Beispiel erst ein auf randomisierter Suche basiertes Orakel zu benutzten, bevor ein strategisches (siehe Abschnitt 4.2.3.1) angewandt wird. Dies ist sehr sinnvoll, da strategische Orakel in der Regel deutlich langsamer und teurer bzgl. der Rechenzeit sind. Nach jeder Behandlung eines Gegenbeispiels wird außerdem die Hypothese mittels des `TerminationChecker`s auf Problem-spezifische Eigenschaften überprüft. Wird eine gesuchte Eigenschaft erfüllt führt dies zu einem vorzeitigen Abbruch des Lernvorgangs und die gefundene Hypothese wird zurück gegeben. Das System ist so meistens noch nicht vollständig gelernt, im Kontext unserer Anwendung kann es jedoch schon genügen einen einzigen Gewinnzustand zu finden, da mit diesem eine Zugfolge gefunden werden kann, die das Spiel sicher gewinnt. Wird kein `TerminationChecker` übergeben oder bis zuletzt keine Abbruchbedingung erfüllt, terminiert der Algorithmus sobald jedes *Equivalence Oracle* benutzt und vom letzten kein Gegenbeispiel für die derzeitige Hypothese gefunden wurde.

4.2.3.1 Konfigurationsmöglichkeiten

Neben den selbst zu implementierenden Komponenten wie dem `GUL` und dem `TerminationChecker` gibt es auch noch eine Reihe von Komponenten der *LearnLib* die für einen Lernvorgang ausgewählt werden können. Im folgenden werden für jede dieser Komponenten alle in der aktiven Lernkomponente vorhandenen Konfigurationsmöglichkeiten genannt und ihre Besonderheiten vorgestellt.

Membership Oracles In der *LearnLib* ist es vorgesehen, dass funktionale *Membership Oracle* (kein Orakel welches direkt mit dem SUL verbunden ist) geschachtelt werden können. Das unterste Orakel ist dabei immer ein das SUL-Interface implementierende Adapter für das zu lernende System. Dieses kann dann in weitere Orakel verpackt werden, die Anfragen des Lernalgorithmus, die sie selbst nicht beantworten können an das nächste Orakel in der Hierarchie weiter reichen.

MealyCacheOracle Dies ist ein Cache, welcher die Ein- und Ausgaben gestellter *Membership Queries*, die durch den Cache geleitet werden, speichert. Er implementiert außerdem einen *„Prefix-Closure“-Filter* [MRS05]. Ein *„Prefix-Closure“-Filter* sorgt dafür, dass beim Auftauchen von definierten Ausgabesymbolen des SUL stattdessen vom Benutzer definierte Ausgaben zurückgegeben werden und die Anfrage danach abgeschnitten wird. Intern wird dies durch ein Mapping realisiert. Dies hat den Vorteil, dass

sobald ein im Mapping enthaltenes Symbol in einer Anfrage zurückgeben wird, alle Anfragen nach diesem Symbol abgeschnitten und auf das vordefinierte Symbol gemappt werden. So können unnötige Systemanfragen vermieden werden. Ein sinnvoller Einsatz ist das frühe Erkennen von Senken im Modell, die so nicht exploriert werden müssen.

CounterOracle Ein *CounterOracle* ist ein Statistikorakel, welches alle Anfragen zählt, die durch es hindurch geleitet werden. Es kann dazu benutzt werden Statistiken über den Lernvorgang zu sammeln indem vor oder hinter bestimmte Orakel geschachtelt wird, die ausgewertet werden sollen.

Da die *LearnLib* sich ständig weiter entwickelt wird können weitere Orakel in das Framework integriert werden indem die Enumerationsklasse `de.tudo.heiss.activelearning.ActiveLearner.MembershipOracle` erweitert wird, wobei jedes Element in dieser Enumeration eine `create`-Methode definieren muss.

Equivalence Oracles Bei *Equivalence Oracle* die auf Random-Orakel basieren, ist es oft sinnvoll Parameter mit Blick auf die derzeitige Hypothesengröße zu wählen um zu kurze Queries, die vielleicht kein Gegenbeispiel finden oder für den derzeitigen Zweck zu lange Queries zu vermeiden, da diese einen kostenspieligen Overhead erzeugen können.

RandomWordsEQOracle Bei diesem Orakel werden die Automaten auf zufällig generierte Wörter überprüft. Die Länge der Wörter lässt sich nach oben und unten beschränken. Die maximale Anzahl an Wörtern ist einstellbar.

RandomWalkEQOracle Dieses Orakel versucht durch einen Random Walk ein Gegenbeispiel im System zu finden. Die maximale Anzahl an Schritten sowie die Wahrscheinlichkeit eines Neustarts lassen sich konfigurieren.

WMethodEQOracle Die W-Methode [Cho78] ist ein sogenannter *conformance test*. Es kann eine Tiefe k bestimmt werden, die verwendet wird um alle möglichen Wörter der Länge k über Σ zu generieren. Jedes generierte Wort wird dazu benutzt um es zwischen jedes Paar von Zustandszugangssequenz mit einem trennenden Suffix zu einem neuen Wort zu konkatenieren, welches dann auf dem SUL ausgeführt wird. Die Anzahl der *Membership Queries* dieser Methode sind exponentiell in k, sie kann

aber oft noch Gegenbeispiele finden, wenn die Random-Orakel versagen. Hier gibt es zwei benutzbare Implementierungen, zum einen das `MealyWMethodEQOracle` und das `MealyIncrementalWMethodEQOracle`, welches nicht direkt alle möglichen Kombinationen an Erweiterungen der Tabelle berechnet, sondern beim ersten gefundenen Gegenbeispiel die Suche unterbricht.

Lernalgorithmen In der *LearnLib* gibt es eine Reihe von Lernalgorithmen, die auch zum Lernen von Mealy-Automaten verwendet werden können. Die importierten Varianten sind in diesem Fall L^* [Ang87], welcher schon in Abschnitt 4.1 vorgestellt wurde, der DHC-Algorithmus [Mer13], der Algorithmus von Kearns und Vazirani [KV94] und der Discrimination-Tree-Algorithmus [How13]. Der erst kürzlich erschiene T^3 [IHS14] wurde noch nicht portiert, da er noch nicht standardmäßig in der *LearnLib* verfügbar ist. Bei vollständiger Inferenz sind die gelernten Modelle aller Algorithmen identisch, allerdings haben die Algorithmen unterschiedliche Stärken und Schwächen, welche die Dauer des Lernvorgangs beeinflussen können.

4.2.3.2 Adaptionen der LearnLib

Um den vorgestellten abstrakten Lernablauf realisieren zu können müssen für viele Klassen der *LearnLib* Wrapper geschrieben werden, die es erlauben die Objekte nach und nach zusammen zu setzen. Dies erleichtert die Benutzung der Komponente, da nicht im Voraus komplizierte Objekte erzeugt werden müssen. Die Verknüpfung der einzelnen Elemente bleibt vor dem Benutzer verborgen. Wie im vorherigen Abschnitt beschrieben, werden für viele Komponenten schon Wrapper bereitgestellt.
Dabei wurde für Lernalgorithmen und *Equivalence Oracle* ein Interface definiert, welches erlaubt nach Erzeugung eines Wrapper-Objektes das benutzte Alphabet und das SUL zu übergeben, sodass später mit einer `create`-Methode das ursprüngliche *LearnLib* Objekt erzeugt werden kann. Dies ist einfach möglich, da diese beiden Objekte standardmäßig keine außergewöhnlichen Abhängigkeiten zu Systemspezifischen Komponenten außer dem Alphabet und dem SUL besitzen, welche einfach an alle Komponenten weitergeben werden, ob sie gebraucht werden oder nicht. Im Falle der *Membership Oracle* musste von dieser Lösung abgesehen werden, da zum Beispiel mit einem *Prefix-Closure-Filter* Abhängigkeiten zum Eingabealphabet des SULs bestehen, welches erst zur Laufzeit bekannt ist. Für *Membership Oracle* wurde deshalb eine Enumerationsklasse `de.tudo.heiss.activelearning.ActiveLearner.MembershipOracle` definiert,

welche die verschiedenen Orakel abstrakt behandelbar macht. Die Instanzen der Enumerationsklasse verfügen dann über eine `create`-Methode, welche die eigentlichen Orakel erzeugt. Um nachträglich unterschiedliche Parameter übergeben zu können wurde außerdem eine Klasse `Tuple` definiert, welche als Container für alle Zusatzeigenschaften genutzt wird. Diese kann im Falle eines neuen Orakels mit besonderen Abhängigkeiten noch um weitere Attribute erweitert werden ohne dass der Algorithmus, welcher die Orakel schachtelt und mit den richtigen Parametern versorgt, an sich verändert werden muss. Für die Schachtelung der Orakel enthält `Tuple` außerdem das innerliegende Orakel und einen String der eine Beschreibung des Orakels beinhalten kann, welcher zur Referenzierung bei statistischen Auswertungen verwendet wird.

4.2.4 Anbindung an das Framework

Die einzelnen Komponenten des Frameworks sollen auch unabhängig voneinander verwendet werden können. Die gesamte Kommunikation basiert auf kurzen Nachrichten, die neue Prozesse starten oder anhalten (vgl. Abschnitt 2). Die Konfigurationsdateien für Lern-Setups oder inferierte Ergebnisse müssen so für andere Prozesse zur Benutzung abgespeichert werden, was in den folgenden beiden Abschnitten genauer erläutert wird.

4.2.4.1 Konfigurationsdatei

Für jede Komponente gibt es eine eigene Konfigurationsdatei mit deren Hilfe sie gestartet werden kann. In der Konfigurationsdatei für aktives Lernen muss deshalb eine komplette Konfiguration eines Lernvorgangs gespeichert werden können. Um diese Konfiguration auch von Hand vornehmen oder modifizieren zu können, wurde deshalb ein XML Schema entwickelt, welches in Listing 4.1 dargestellt ist. Mit Hilfe von JAXB [4] können aus diesem Schema automatisch Java-Klassen generiert werden, die man mit mitgelieferten Methoden von JAXB nach XML serialisieren und wieder deserialisieren kann. Die fixen Elemente eines Lernvorgangs sind dabei eine Liste von abzuarbeitenden *Equivalence Oracles*, optionalen *Membership Oracles*, dem Lernalgorithmus, einem optionalen `TerminationChecker` und dem Pfad an dem der Ausgabeautomat, und alle Zwischenhypothesen, gespeichert werden soll.
Für jedes *Equivalence Oracle* muss zum Speichern ein `eqOracle`-Objekt erzeugt werden, welches mindestens den absoluten Klassenpfad des originalen Orakels ent-

[4] https://jaxb.java.net

hält. Da jedes Orakel wie zum Beispiel ein `RandomWalkEQOracle` auch noch spezifische Parameter hat, die nur es selbst betreffen wurde im XML-Schema für diesen Zweck der Platzhalter `anyAttribute` hinzugefügt, der das Definieren von beliegen Eigenschaften erlaubt. Dabei sind auch arithmetische Ausdrücke erlaubt um variable Parameter, bzgl. der Hypothesengröße für z.B. die minimale Länge der *Random Walks*, definieren zu können. Dabei ist n als einzige definierte Variable erlaubt. Die arithmetischen Ausdrücke müssen allerdings später speziell geparst werden.
Für jedes *Membership Oracle* wird der Name des definierenden Enums (siehe Abschnitt 4.2.3.2) als absoluter Name gespeichert. Optional kann noch ein beschreibender String und ein Prefix-Closure-Mapping übergeben werden. Die Eingabesymbole werden dabei über ihre String-Repräsentation (`toString()`) identifiziert, sodass jedes über diese Methode zweifelsfrei unterscheidbar sein muss. Zum Auslesen wird ein Eingabealphabet benötigt.
Der Lernalgorithmus wird über den absoluten Namen des Wrappers identifiziert um ein entsprechendes Objekt zu erzeugen. Optionale und individuelle Parameter können über `anyAttribute` mit gespeichert werden.
Ein `TerminationChecker` stellt eine zum SUL gehörige händische Implementierung da, für dessen Speicherung nur der Name der Klasse notwendig ist.
Zum Speichern eines Lernsetups können so die einzelnen Elemente in die generierten Java-Klassen gekapselt werden um dann serialisiert zu werden. Dies funktioniert auch ohne vorher die eigentlichen Klassen, die für einen Lernvorgang benötigt werden, zu erzeugen. In einer GUI können diese Werte zum Beispiel über Dropdown-Boxen und Textfelder gesammelt werden. Zum Erstellen der Objekte aus den generierten Java-Klassen muss für jedes Objekt eine `fromXMLElement`-Methode geschrieben werden, die die aus XML generierte Klasse übergeben bekommt und so in der Lage ist alle sie betreffenden Parameter eigenständig auszulesen. Diese Methode wird mittels *Java Reflection API* auf dem bekannten Namen der Klasse aufgerufen.

Listing 4.1: XML Schema für die Konfiguration von aktiven Lernprozessen

```
<xs:complexType name="eqOraclesType">
  <xs:sequence minOccurs="1" maxOccurs="unbounded">
    <xs:element name="eqs" type="eqOracle"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="eqOracle">
  <xs:sequence><xs:any></xs:sequence>
  <xs:attribute name="class" type="xs:string" use="required" />
  <xs:anyAttribute />
</xs:complexType>
```

```
<xs:complexType name="entry">
  <xs:attribute name="key" type="xs:string" use="required" />
  <xs:attribute name="value" type="xs:string" use="required" />
</xs:complexType>

<xs:complexType name="prefixClosureMap">
  <xs:sequence minOccurs="1" maxOccurs="unbounded">
    <xs:element name="entries" type="entry"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="mqOracle">
  <xs:sequence minOccurs="0" maxOccurs="1">
    <xs:element name="mapping" type="prefixClosureMap"/>
  </xs:sequence>
  <xs:attribute name="oracle" type="xs:string" use="required" />
  <xs:attribute name="desc" type="xs:string" use="required" />
</xs:complexType>

<xs:complexType name="mqOraclesType">
  <xs:sequence minOccurs="1" maxOccurs="unbounded">
    <xs:element name="mqs" type="mqOracle"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="learnerType">
  <xs:sequence minOccurs="0" maxOccurs="unbounded">
    <xs:any processContents="skip" />
  </xs:sequence>
  <xs:attribute name="class" type="xs:string" use="required" />
  <xs:anyAttribute />
</xs:complexType>

<xs:complexType name="learnSetup">
  <xs:sequence  maxOccurs="1">
    <xs:element name="terminationChecker" type="xs:string" minOccurs="0"/>
    <xs:element name="automatonFile" type="xs:string" minOccurs="1" />
    <xs:element name="learner" type="learnerType" minOccurs="1"/>
    <xs:element name="eqOracles" type="eqOraclesType" minOccurs="1"/>
    <xs:element name="mqOracles" type="mqOraclesType" minOccurs="1" />
  </xs:sequence>
</xs:complexType>
```

4.2.4.2 Speichern der Ergebnisse

Auf den ersten Blick ist die einzige Ausgabe des aktiven Lernvorgangs ein Mealy Modell des gelernten Systems, welches über ein definiertes XML Schema (ähnlich des Schemas in Abschnitts 4.2.4.1) geladen und gespeichert werden kann. Um allerdings das aktive Lernen in den Gesamtablauf integrieren zu können, muss ein Lernvorgang wieder aufgenommen oder mit Hilfe von gefundenen Gegenbeispielen die Hypothese

verfeinert werden können. Aus diesem Grund wurde wie vorher beschrieben ein Cache direkt vor das SUL geschaltet, welcher alle gestellten Anfragen speichert.
Um einen Lernvorgang später weiterführen oder mit Gegenbeispielen verfeinern zu können, muss dieser Cache gespeichert werden. Da er allerdings auf uns unbekannten Objekten operiert (das Eingabealphabet ist erst zur Laufzeit bekannt) wird, muss der Cache als ganzes Java-Objekt serialisiert werden, was allerdings mit einem einzigen Aufruf über die Bibliothek XStream [5] möglich ist.
Beim Laden eines Lernvorgangs muss allerdings darauf geachtet werden, dass das über den Cache gespeicherte Alphabet verwendet wird, da die Java-Objekte ansonsten inkompatibel mit denen aus dem geladenen SUL sind. Dies muss auch beim Laden von Gegenbeispielsequenzen oder eines *Prefix-Closure-Filters* beachtet werden.
Wird ein Lernalgorithmus über eine Konfigurationsdatei geladen, kann er anhand der im Arbeitsverzeichnis liegenden Dateien entscheiden, ob es sich um einen neuen Lernvorgang oder um einen fortgeführten Lernvorgang handelt. In letzterem Fall müssen das GUL und der Cache zum weiter Lernen wieder hergestellt werden. Wurden von der Inferenz-Komponente Gegenbeispiele für das Problem gefunden, werden diese ebenfalls im Voraus geladen und an den Algorithmus übergeben.

[5] http://xstream.codehaus.org

5 | Inferenz

Die Gegnerinferenz stellt den zweiten wesentlichen Schritt der Projektgruppe dar. Nachdem durch die unterschiedlichen Lernalgorithmen die verschiedenen Strategien diverser KI erlernt wurden, soll durch die Simulation eines Spiels die beteiligte KI identifiziert werden. Auf Grundlage dieser Erkenntnis kann beispielsweise eine passende Gegenstrategie gewählt werden, um die Gewinnchancen zu verbessern.

5.1 Theorie

Die Inferenz gliedert sich in zwei wesentliche Bestandteile auf. Im ersten Teil (Algorithmus 5.1) wird der nächste Zug bestimmt, der am wahrscheinlichsten zu einer Reduktion der potentiellen Gegner führt.
Hierzu wird für jedes Inputsymbol bei jedem Automaten ermittelt, welches Outputsymbol dieser zurück gibt. Dabei wird gezählt, wie oft jedes Outputsymbol bei welchem Inputsymbol auftaucht. Im Anschluss daran wird ermittelt, welches Outputsymbol bei welchem Inputsymbol am seltensten vorkommt und das entsprechende Inputsymbol zurück gegeben. Dabei ist zu beachten, dass Inputsymbol gewählt wird, bei dem das entsprechende Outputsymbol minimal, aber auch größer 0 ist.
Der zweite Teil (Algorithmus 5.2) führt die eigentliche Inferenz durch. Dabei wird die Reaktion der KI auf die bisherigen Züge, mit den erwarteten Reaktionen der hinterlegten erlernten Automaten verglichen. Dazu wird dem GUL eine Zugfolge übergeben und das letzte Outputsymbol der Rückgabe mit der Rückgabe aller Automaten verglichen. Unterscheidet sich das Outputsymbol des GUL von einem der Automaten, so kann dieser Automat ausgeschlossen werden. Dies wird solange fortgesetzt, bis nur noch ein Automat übrig ist, der somit für die betrachteten Inputsymbole der KI in der GUL entspricht.
Für diese Algorithmen besteht noch Optimierungspotential in Hinblick die notwendige Anzahl an durchzuführenden Zügen bis eine KI identifiziert werden konnte. Der Algorithmus schaut derzeit nur einen Schritt in die Zukunft und entscheidet

Algorithmus 5.1 optimalTurn

Eingabe: $potEnemies$, $moveHistory$

Ausgabe: $optimalTurn$

```
1: optimalTurn ← randomTurn()
2: countPotTurns ← DICTIONARY[input, DICTIONARY[output, count]]
3: for all potEnemy ← potEnemies do
4:    for all input ← inputs do
5:       nextTurn ← append(moveHistory, input)
6:       countPotTurns[input][nextMove(potEnemy, nextTurn)] + +
7:    end for
8: end for
9: optimalTurn ← minTurn(countTurns)
10: return optimalTurn
```

Algorithmus 5.2 EnemyInference

Eingabe: $potEnemies$, $moveHistory$

Ausgabe: $quantityPotEnemies$

```
1: quantityPotEnemies ← 0
2: for all potEnemie ← potEnemies do
3:    if GUL.step(moveHistory) EQUALS potEnemie.step(moveHistory)
      then
4:       quantityPotEnemies ++
5:    else
6:       REMOVE potEnemie FROM potEnemies
7:    end if
8: end for
9: return quantityPotEnemies
```

auf Grundlage dessen, welcher nächster Zug gemacht wird. Ein Ansatz zur Verbesserung wäre es mehrere Züge in die Zukunft zu schauen und somit die Anzahl der notwendigen Züge zu reduzieren, da die Wahrscheinlichkeit mehr KI in kürzer Zeit auszusortieren steigt. Denkbar wäre der Einsatz der Konzepte aus der Routenplanung, wie beispielsweise dem Dijjkstra Algorithmus, oder aus denen der künstlichen Intelligenz, wie AlphaBeta-Pruning. Hierdurch lassen sich vor allem bei den passiven Automaten bessere Ergebnisse erwarten.

5.2 Realisierung

Die Inferenz benötigt neben einer GUL, bei der eine KI gesetzt worden sein muss, auch eine Menge von gelernten Automaten. Diese Automaten müssen entweder direkt oder über einen Wrapper das Interface „Inferable“ implementieren. Dieses Interface beinhalten einen Getter und Setter für den Parameter „Name“, um den Automaten identifizieren zu können, eine Methode, die für eine Verkettung von Inputsymbolen ein Outputsymbol liefert, und einen Getter, der Auskunft darüber gibt, mit welcher Vorgehensweise[1] der Automat gelernt wurde.

Die Inferenz lässt sich in zwei Modi betreiben. Zum einen kann sie als ein normales Javaprogramm über die Konsole gestartet werden, wobei dem Programm eine GUL und eine beliebige Anzahl an Automaten als Argumente mitgegeben werden müssen. Zum anderen kann die Inferenz im Zusammenspiel mit dem JMS betrieben werden, sodass über eine Startnachricht angegeben wird, welche GUL mit welchen Automaten inferiert werden soll.

Bei der Implementation der Inferenz wurde darauf geachtet, dass diese leicht erweiterbar ist. So wurden die Umsetzung der Algorithmen 5.1 und 5.2 in eigene Klassen ausgelagert, die durch eine Schablonenmethode [GHJV10, S. 366ff] in den Inferenzablauf eingebunden werden. Dadurch wird es möglich Besonderheiten bei bestimmten Automatentypen ohne großen Aufwand zu berücksichtigen oder beispielsweise einen anderen Algorithmus für die Bestimmung des optimalen Zuges zu verwenden.

Konkret wurde dieser Mechanismus dafür benutzt die Besonderheiten zwischen Mealy Automat (MA) und Deterministic Finite Automaton (DFA) abbilden zu können. So liefert der DFA gegenüber dem MA als Outputsymbole nur eine boolsche Variable zurück, die die Werte „true“oder „false“ annehmen kann. Die GUL liefert nicht dieselben Outputsymbole, weshalb ein Wrapper genutzt wird, der die Outputsymbole zu den entsprechenden Outputsymbolen des DFA übersetzt. Dabei werden die akzeptierenden Zustände auf „true“ und alle anderen Zustände auf „false“ übersetzt.

Eine weitere Möglichkeit die Inferenz zu erweitern stellt die Verarbeitung von Automaten dar, die für die einzelnen Transitionen Wahrscheinlichkeiten hinterlegt haben. Dies ist in erster Linie für nicht-deterministische Strategien erforderlich, die das Framework derzeit noch nicht unterstützt.

Die Inferenz liefert als Ergebnis nicht nur den oder die Automaten, die der KI in der GUL entsprechen, sondern auch die Verkettung aller Inputsymbole, die unter-

[1] Derzeit unterscheiden wir zwischen aktiv- und passivgelernten Automaten.

sucht wurden. Diese Verkettung kann als Pfad visualisiert werden, der durch den Spielbaum aller möglichen Züge des Spiels führt, bis die KI inferiert wurde. Diese Informationen werden an die GUI gesendet, sodass sie dem Nutzer dargestellt werden können.
Die aktiven Automaten müssen nicht vollständig bzw. vollständig korrekt gelernt sein. Aus diesem Grund ist es möglich, dass ein Automat fälschlicherweise aussortiert wird. Da innerhalb der Inferenz jedoch nicht bestimmt werden kann, ob es sich an der Stelle um ein Gegenbeispiel für den entsprechenden Automaten handelt oder die KI im GUL einfach nicht zum Automaten passt, werden die verketteten Inputsymbole in eine Datei geschrieben und das aktive Lernen über JMS[2] darüber informiert, dass es möglicherweise ein Gegenbeispiel gibt. Das aktive Lernen entscheidet dann, ob es sich hierbei um ein korrektes Gegenbeispiel handelt und verfeinert gegebenenfalls den Automaten. Dieser verfeinerte Automat wird allerdings in der Inferenz nicht mehr berücksichtigt.
Die Gruppe um das passive Lernen hat auf die Übermittlung dieser potentiellen Gegenbeispiele verzichtet, da hier nicht die Möglichkeit besteht bestehende Automaten zu verfeinern, sondern der Automat vollständig neu gelernt werden müsste.

5.3 Beispiel

Um die Funktionsweise der beiden Algorithmen 5.2 und 5.1 und deren Interaktion untereinander besser zu verstehen, soll in diesem Abschnitt eine Inferenz an einem fiktiven Beispiel durchgespielt werden. Hierzu werden drei verschiedene MA angenommen, die jeweils eine KI repräsentieren. Die Automaten operieren auf dem Inputalphabet, das A, B und C enthält. Das Outputalphabet besteht aus X,Y und Z.

Initialisierung Zu Beginn enthält die Variable *potEnemies* die drei in Abbildung 5.1 dargestellten Automaten, von denen der Automat A der in der GUL gesetzten KI entspricht. Da noch kein Zug gemacht wurde ist *moveHistory* leer und alle Automaten befinden sich im Zustand q_0.

Schritt 1 Zu Beginn des ersten Schrittes wird das Inputsymbol ermittelt, das dafür sorgt, dass Automaten aussortiert werden, aber mindestens einer übrig bleibt. Da

[2]Wurde die Inferenz ohne die Kommunikation über JMS gestartet, wird nur in die Datei geschrieben und keine Nachricht an das aktive Lernen gesendet.

sich alle drei Automaten identisch für den Übergang von q_0 zu q_1 verhalten, wird ein beliebiges der Inputsymbole, hier soll es A sein, ausgewählt und zu $moveHistory$ hinzugefügt.
Da die GUL als KI den Automaten A enthält, gibt dieser für das Inputsymbol A das Outputsymbol Y zurück, weshalb keiner der Automaten aussortiert werden kann. Die Automaten befinden sich nach diesem Schritt alle im Zustand q_1, wie in Abbildung 5.2 skizziert.

Schritt 2 Die Ermittlung der besten Zugfolge ergibt für das Inputsymbol A, dass nur der Automat C das Outputsymbol Z ausgibt. Die beiden anderen Automaten geben das Outputsymbol Y aus.
Die neue $moveHistory$ wird der GUL übergeben und diese gibt als Outputsymbol Y zurück. Daher kann der Automat C ausgeschlossen werden. Automat A befindet sich, wie in Abbildung 5.3 nach dem zweiten Schritt in q_2 und Automat B in Zustand q_3.

Schritt 3 Im dritten Schritt liefert die Betrachtung der beiden verbleibenden Automaten das Ergebnis, das für jedes Inputsymbol je Automat ein unterschiedliches Outputsymbol ausgegeben wird. So kann als Inputsymbol beispielsweise das A zu $moveHistory$ hinzugefügt werden.
Wird $moveHistory$ der GUL übergeben, wird das Outputsymbol X zurückgegeben, weshalb Automat B ausgeschlossen werden kann und Automat A als Äquivalent zur hinterlegten KI in der GUL angenommen werden kann. Dies wird zurückgegeben und die Inferenz ist beendet.

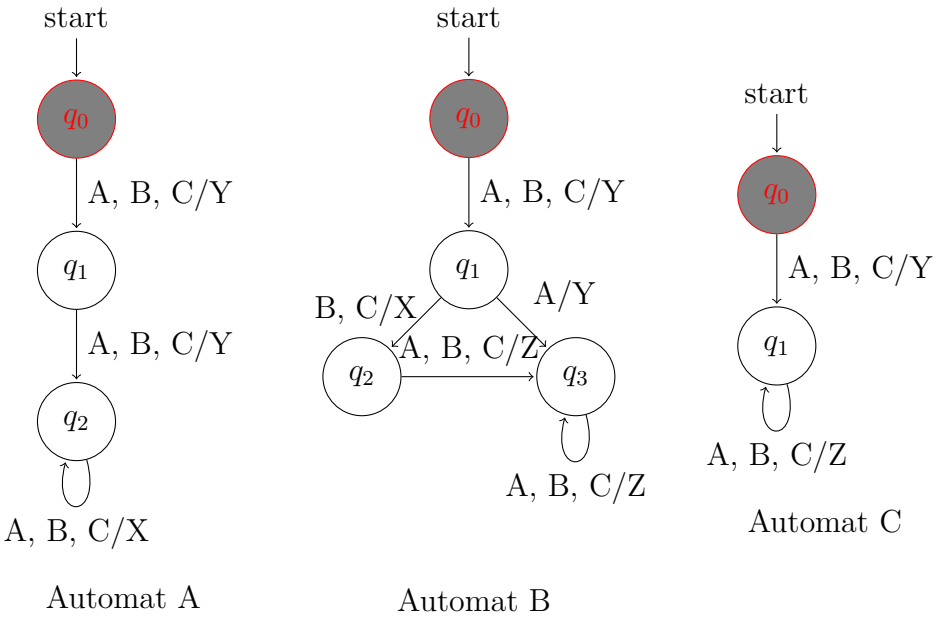


Abbildung 5.1: Inferenzbeispiel - Initialisierung

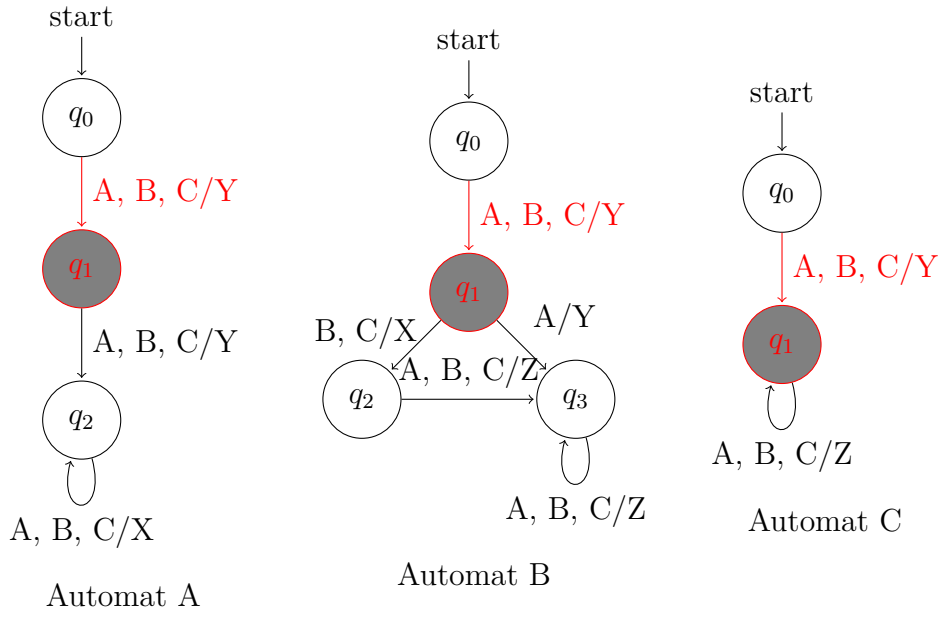


Abbildung 5.2: Inferenzbeispiel - Schritt 1

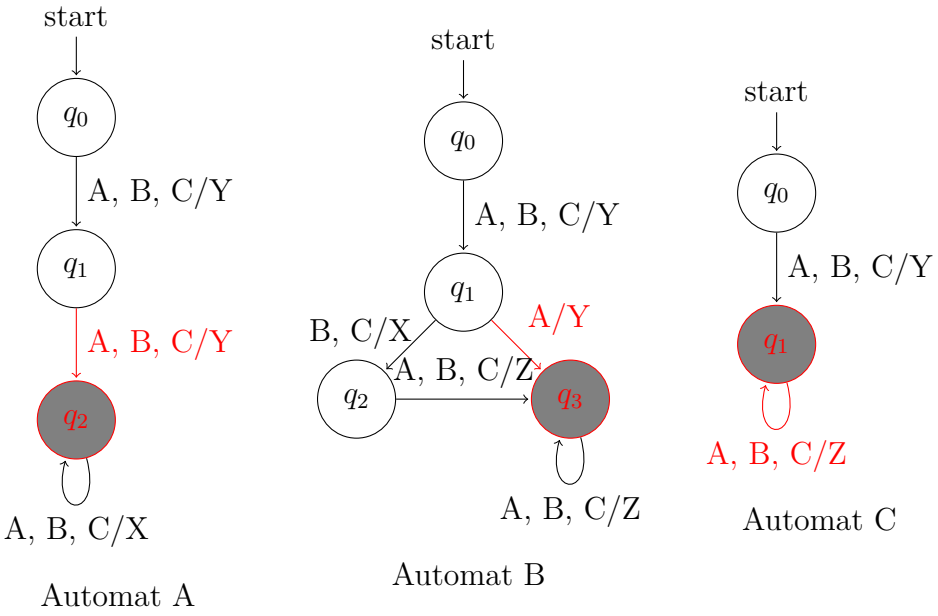


Abbildung 5.3: Inferenzbeispiel - Schritt 2

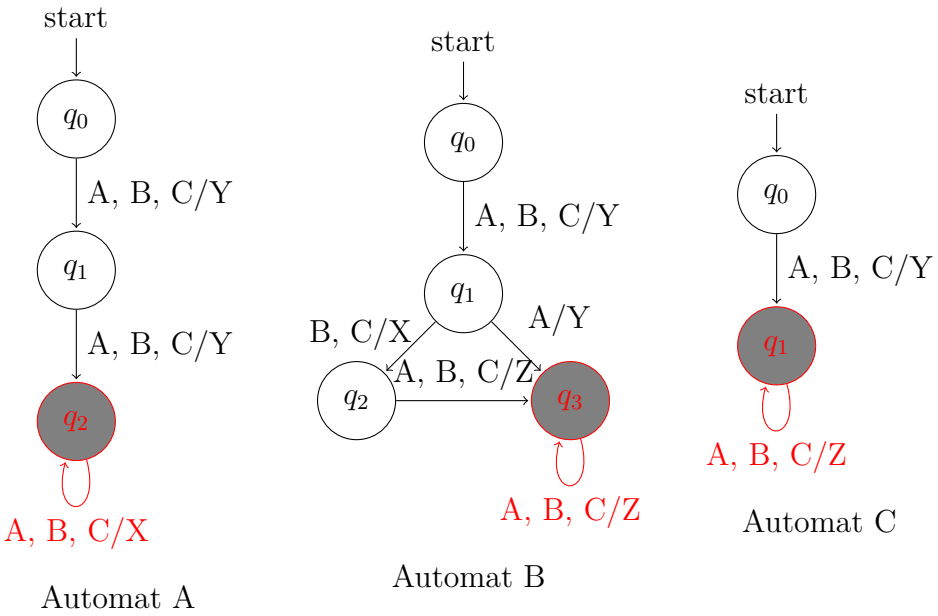


Abbildung 5.4: Inferenzbeispiel - Schritt 3

6 | Auswertung

Für die Anwendung in realen Applikationen, ist die Qualität der vorgestellten Verfahren (siehe Kapitel 3 bis 5) von großer Relevanz. In diesem Kapitel werden daher die Verfahren unter verschiedenen Gesichtspunkten analysiert und ausgewertet. Konkret werden im Folgenden zwei unterschiedliche Aspekte betrachtet:

- Die Qualität der inferierten Modelle.
  Es ist ersichtlich, dass eine gute Approximation eines Systemes für viele Aufgaben besser geeignet ist, als eine schlechte. Es existieren jedoch noch viele weitere Dimensionen, in denen sich ein Modell untersuchen lässt, wie beispielsweise die Größe eines Modelle oder die Dauer, die es zum Erstellen des Modells gebraucht hat. So ist unter Umständen ein Modell einem anderen vorziehen, wenn dieses nur unwesentlich ungenauer ist, aber in einer sehr viel geringeren Zeit erstellt werden kann. Die einzelnen Aspekte werden in Abschnitt 6.1 diskutiert.
- Die Anwendung im Falle der Gegnerinferenz.
  Das Anwendungszenario dieser Projektgruppe ist das Inferieren von gegnerischen KIs auf Basis vergangener Beobachtungen. Inwieweit die Inferenz erfolgreich ist, wird in Abschnitt 6.2 behandelt. Auch hier lassen sich verschiedene Aspekte, wie Vorhersagegüte und -dauer betrachten.

6.1 Modellqualität

Für die Auswertung der Lernverfahren wurden für verschiedene Konfigurationen Daten erhoben. Im Folgenden wird jedoch nur eine Auswahl der Ergebnisse vorgestellt. Die vollständigen Daten können im Repository der Projektgruppe eingesehen werden[1].

[1] `ssh://git@projekte.itmc.tu-dortmund.de/informatik-ls5/pg-heiss.git`

Versuchsaufbau Für die passiven Lernverfahren besteht der erste Schritt darin, Trainingsdaten zu erheben. Dazu wurden n Partien (siehe unten) gegen den jeweiligen Gegner – eine Kombination aus GUL und KI – simuliert, bei denen jeder Zug zufällig gewählt wurde. Wurde eine Partie gewonnen, wurde das gespielte Wort als positives Sample verwendet, wurde eine Partie verloren, wurde das gespielte Wort entsprechend als negatives Sample gewertet. Partien, welche in einem Unentschieden oder sonstigem Zustand (bspw. Exception) endeten, wurden verworfen.

Um ähnliche Rahmenbedingungen bei den aktiven Lernverfahren zu schaffen, wurde in den Tests ein RandomWordEQOracle verwendet, welches n Anfragewörter der Länge 1 bis |„Längstes Sample der passiven Verfahren“| stellen darf. Hierbei entsteht jedoch eine Diskrepanz, da die aktiven Lernverfahren mit zwei Äquivalenztests im schlechtesten Fall bereits das Wissen von $2n$ verschiedenen Samples erhalten und damit einen Vorteil gegenüber den passiven Verfahren erhalten. Desweiteren wird die Anzahl der MembershipQueries, welche während der Hypothesenkonstruktion gestellt werden, nicht berücksichtigt. Diese Probleme werden in gewisser Weise abgeschwächt, da es passieren kann, dass Partien nicht bis zu Ende gespielt werden und ein Query somit nicht notwendigerweise den gleichen Informationsgehalt wie ein Sample/Trace im passiven Fall besitzt. Bei der Interpretation der Ergebnisse sollte dieser Umstand jedoch berücksichtigt werden.

Ein weiterer wichtiger Punkt ist das Verfahren zur Erhebung der Kenngrößen. Im Falle der passiven Lernverfahren könnte für die Ermittlung der Modellgüte beispielsweise der Wiedereinsetzungsfehler betrachtet werden. Man überlegt sich jedoch, dass ein „Lern“-Verfahren, welches lediglich das Trainingsset auswendig lernt, stets einen Wiedereinsetzungsfehler von 0 besitzt. Insbesondere gilt dies auch für alle schwachkonsistenten Lernverfahren. Daher wurden bei der Analyse die Mittelwerte einer 10-fachen Kreuzvalidierung [Koh95] verwendet. Die aktiven Lernverfahren wurden jeweils auf den gleichen Testdaten, wie die passiven Lernverfahren ausgewertet.

Das zuvor erwähnte n wurde auf den Wert 10000 gesetzt. Im Falle einer 10-fachen Kreuzvalidierung bedeutet dies, dass die Trainingsmengen jeweils eine Größe von 1000 besitzen. Verschiedene Untersuchungen [WK91] zeigen, dass mit 1000 Testdaten der tatsächliche Fehler hinreichend genau geschätzt werden kann. In einigen Fällen kann das Erheben von 10000 Trainingsdaten jedoch zu teuer sein, sodass ebenfalls Konfigurationen mit $9000, 8000, ..., 1000$ Trainingssamples betrachtet wurden, um einen Eindruck davon zu erhalten, wie die (passiven) Verfahren mit geringen Daten umgehen können.

Getestet wurden die zuvor vorgestellten passiven Verfahren `RPNI`, `LCSM`, `HCSM` und `SVM`. Im Falle des aktiven Lernens wurden die Verfahren `DHC` und `KearnsVazirani` untersucht. Die aktiven Verfahren wurden einmal auf eine „simple“ und eine „complex“ Weise getestet. Eine Ungleichheit zwischen den aktiven und passiven Verfahren ist, dass die passiven Verfahren in der hier verwendeten Form einen DFA anstatt einen Mealy-Automaten lernen. Das bedeutet, die passiven Verfahren klassifizieren ein Beispiel korrekt, wenn lediglich der finale Zustand einer Eingabe korrekt ist. Daher wird in der aktiven, *simplen* Variante lediglich getestet, ob das letzte Symbol des Ausgabewortes zu den akzeptierenden Ausgabesymbolen gehört oder nicht. In der *komplexen* Variante wird ein Testdatum nur dann als richtig klassifiziert gewertet, wenn das Ausgabewort komplett übereinstimmt.

Ergebnisse Zunächst werden die Ergebnisse für das Spiel *Connect Four* auf einem 5×4 Spielfeld gegen die KI Downy, welche in Abschnitt 2.3.1.1 näher erläutert worden ist, vorgestellt. Die Klassifikationsleistung der einzelnen Verfahren ist in Abbildung 6.1 dargestellt.

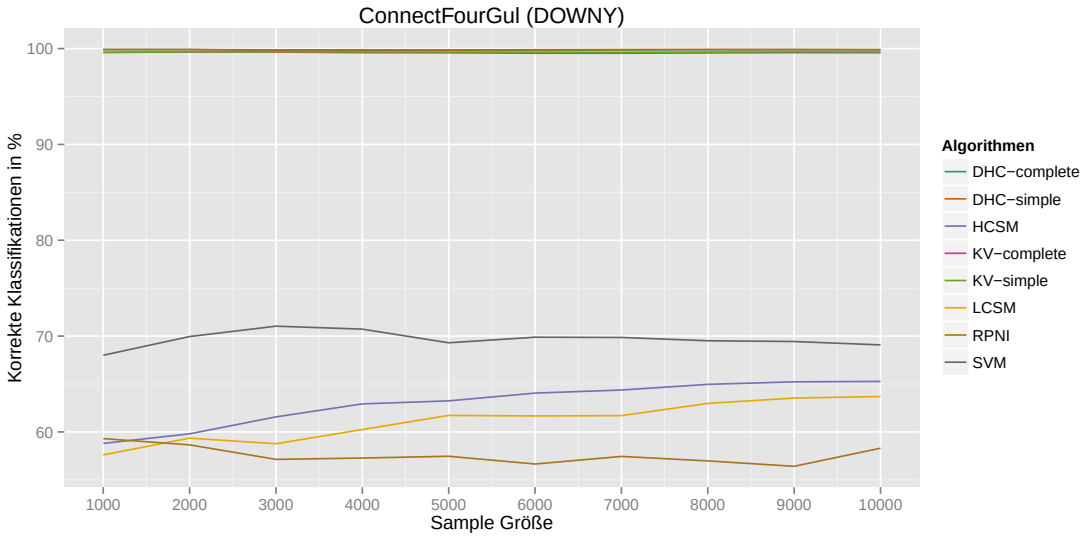


Abbildung 6.1: Modellqualität für *Connect Four* mit der KI Downy.

Wie zu erwarten, schneiden die aktiven Verfahren mit fast perfekten Klassifikationsleistungen sehr gut ab. Selbst in der komplexen Variante sind die Ergebnisse nahezu perfekt, was für eine sehr genaue Modellapproximation spricht. Vergleichsweise weit abgeschlagen, mit bis zu 30%-Punkten schlechteren Leistungen, liegen dagegen die passiven Lernverfahren. Überraschend hierbei ist jedoch, dass die SVM, welche sich aufgrund ihres numerischen Hintergrundes doch sehr deutlich von den klassischen,

modellbasierten Verfahren unterscheidet, sichtlich besser als diese abschneidet. Dieser Trend ist im Mittel auch bei den übrigen KIs festzustellen.

Der Verlauf über die Sample-Größen zeigt, dass die Verfahren nicht signifikant schlechter werden, wenn ihnen nur eine geringe Anzahl an Trainingsdaten zur Verfügung stehen. Es lässt sich zwar ein aufsteigender Trend erkennen, dieser beschränkt sich jedoch – auch bei den übrigen KIs – auf ein Ausmaß von ca. 5%-Punkten.

Es stellt sich die Frage, wie die aktiven (komplexen) Lernverfahren so einen großen Vorsprung gegenüber den passiven Verfahren erreichen können. Dazu lassen sich weitere Dimensionen der Modellbildung betrachten. In Abbildung 6.2 ist dazu die Anzahl der Zustände der erstellten Hypothesen dargestellt.

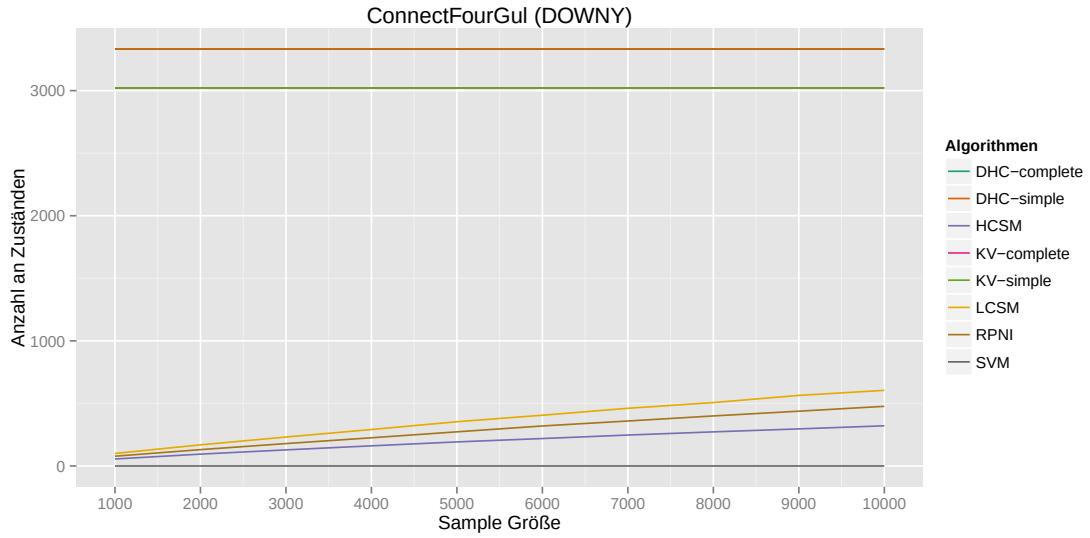


Abbildung 6.2: Modellgröße für *Connect Four* mit der KI Downy.

Es zeigt sich, dass die Modelle der aktiven Lernverfahren um eine Größenordnung größer sind als die der passiven Lernverfahren. Es ist naheliegend, dass dadurch weitaus mehr strukturelle Informationen modelliert werden können, welche eine bessere Modellapproximation erlauben.

Hinsichtlich der passiven Verfahren zeigt die Grafik ebenfalls interessante Eigenschaften der selbstentwickelten Lernverfahren. So war die Idee hinter dem HCSM-Ansatz, dass stark-überdeckte Knoten zuerst gemergt werden. Dies resultiert darin, dass die inferierten Modell kleiner sind, als beispielsweise beim klassischen RPNI-Ansatz, da starke Strukturen sehr früh in der Modellbildung manifestiert werden und ein Großteil der Informationen damit in sehr wenigen Zuständen ausgedrückt werden kann.

Die hohe Modellkomplexität der aktiven Verfahren ist jedoch mit weiteren Eigenschaften verbunden. Abbildung 6.3 die benötigte Zeit, die die Algorithmen für die Modellbildung benötigt haben.

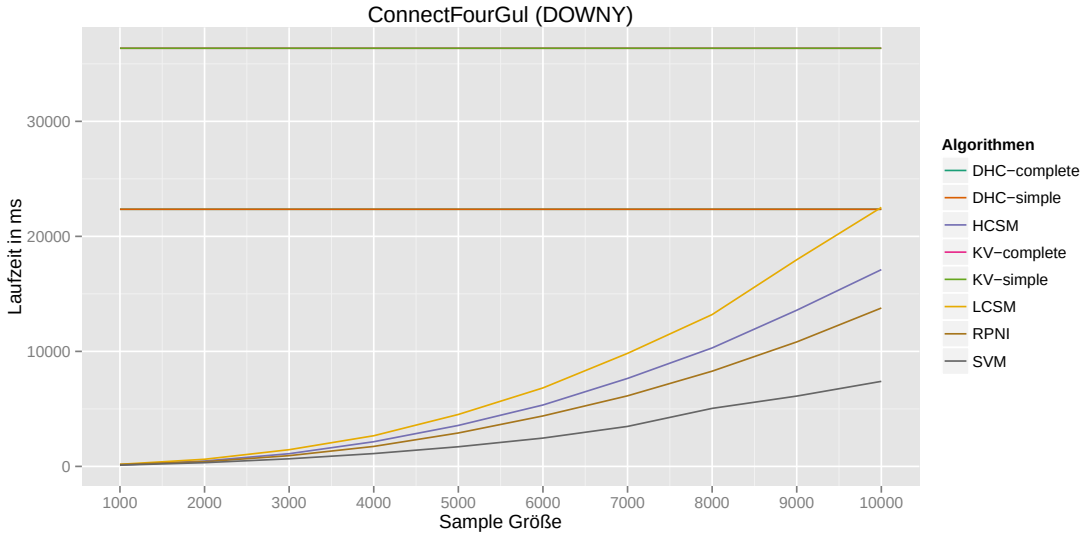


Abbildung 6.3: Modelldauer für *Connect Four* mit der KI Downy.

Die aktiven Verfahren benötigen die längste Zeit, um ihr Modell zu erstellen, wobei der DHC-Ansatz in etwa $\frac{2}{3}$ der Zeit des KV-Ansatzes benötigt. Im Falle der passiven Verfahren ist zu sehen, dass insbesondere mit zunehmender Sample-Größe die Laufzeit immer schlechter wird. Dieser Trend ist so stark, dass bei 10000 Samples die Laufzeit des LCSM-Verfahrens in etwa identisch mit der des DHC-Verfahrens ist. Die Unterschiede zwischen den passiven Verfahren lassen sich auf die unterschiedlichen Implementierungen zurückführen. Der Kern der SVM ist in einer C-Bibliothek (Single-Threaded) geschrieben, wohingegen die modellbasierten Verfahren in Java (Multi-Threaded) implementiert sind. Bei den [H|L]CSM Ansätzen kommt hinzu, dass die Knoten vor den Merge-Versuchen sortiert werden müssen, sodass weiterer Overhead entsteht.

Bei der Auswertung der Daten muss jedoch ebenfalls die Dimensionierung des Problemes betrachtet werden. Kein Lernverfahren benötigt länger als eine Minute zum Erstellen seines Modells, was für eine sehr kleine Problemkomplexität spricht. Insbesondere die Länge einer Partie *Connect Four* auf einem 5×4 ist sehr gering. Desweiteren hängt die Laufzeit der aktiven Verfahren maßgeblich von der Performanz des zu untersuchenden Systemes ab, da dieses während des Lernvorganges sehr häufig angefragt wird. Im Falle von *Connect Four* wurde das Spiel in der Projektgruppe

selbst implementiert, sodass eine sehr auf Performanz ausgelegte Implementierung gewählt werden konnte.
Um daher weitere Aspekte untersuchen zu können, wurde ebenfalls das externe Spiel *Chain Reaction* betrachtet. Dort zeigen sich bereits im Aufbau der Versuche erste Unterschiede. Um überhaupt sinnvoll mit der Komplexität des Spieles umgehen zu können, wurde im Falle der aktiven Lernverfahren ein *WinningSymbol*-TerminationChecker verwendet. Das bedeutet, sobald die Hypothese des aktiven Lerners einen Zustand besitzt, in dem die Partie gewonnen ist, bricht der Lernvorgang ab. Wie in Abb. 6.4 zu sehen ist, ist die daraus resultierende Konsequenz, dass die aktiven Verfahren sehr schlecht abschneiden.

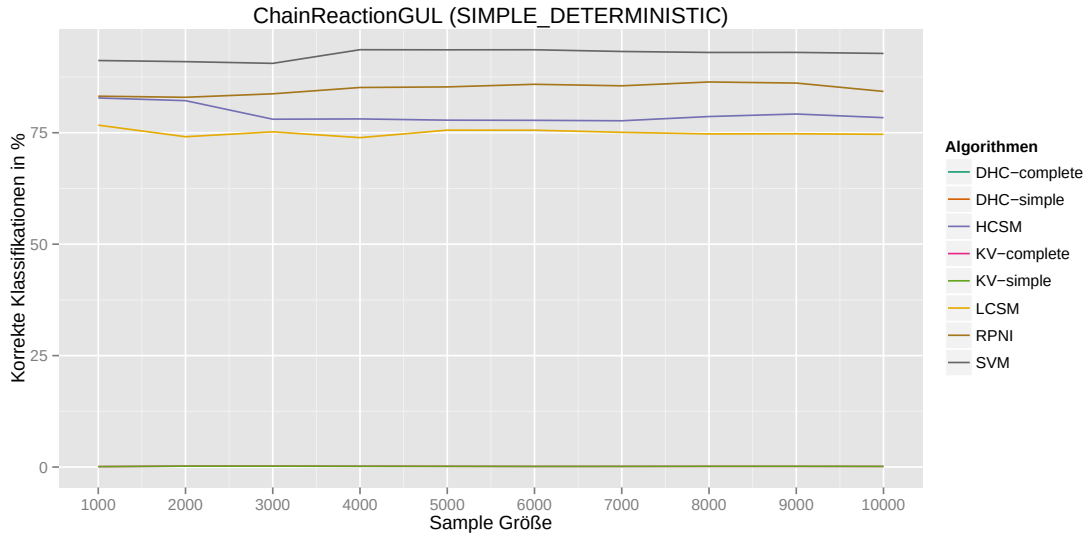


Abbildung 6.4: Modellqualität für *Chain Reaction*.

Erneut liegt die SVM vor den modellbasierten Verfahren. Im Vergleich zu dem *Connect Four*-Beispiel, schließen die passiven Verfahren insgesamt etwas besser ab. Diese Werte sind jedoch kritisch zu sehen. Im Falle von *Chain Reaction* sind nur etwa 1% der zufällig generierten Trainingsdaten positiv gelabelt. Das bedeutet insbesondere, ein Modell, welches lediglich aus einem ablehnenden Zustand bestehen würde, hätte eine erwartete Klassifikationsgüte von 99%. Durch die vielen negativen Samples wird im Falle der modellbasierten Verfahren das Mergen von Zuständen verhindert, sodass unter Umständen bis auf die positiven Samples des Trainingsdatensatzes keine weiteren Eingaben akzeptiert werden. Da jedoch ebenfalls viele negative Samples zum Testen verwendet werden, können diese Modelle dadurch eine hohe Güte erreichen. Zum Vergleich: Im Falle des *Connect Four* (Downy) Beispiels, war das Verhältnis zwischen positiven und negative Samples in etwa 40/60.

Das frühe Abbrechen des Lernvorganges sorgt dafür, dass das aktive Modell selbst in der *simple* Auswertung, kein Datum richtig klassifiziert. Obwohl das Modell damit gänzlich unbrauchbar ist, besitzt es dennoch die meisten Zustände, wie in Abbildung 6.5 dargestellt.

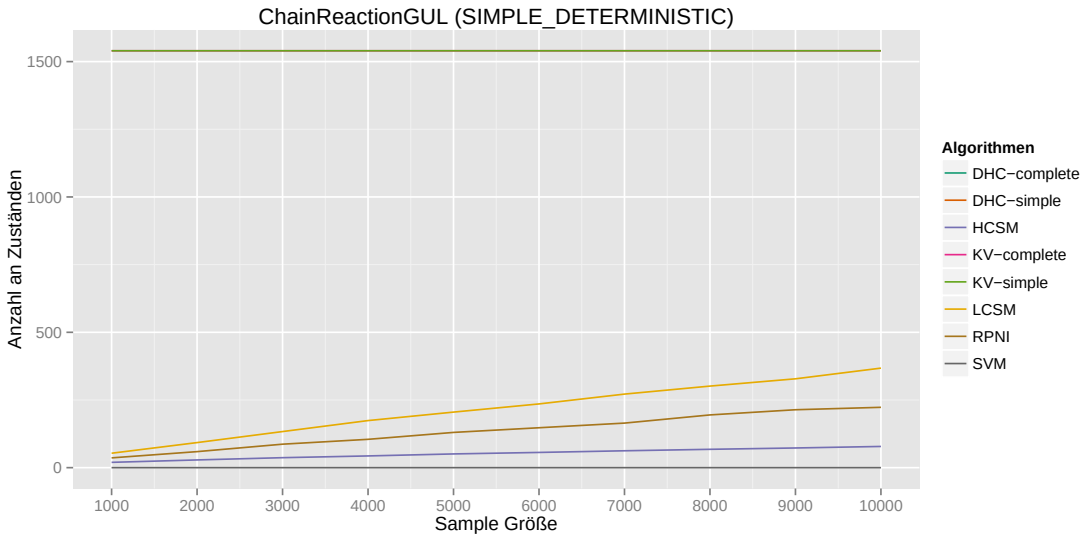


Abbildung 6.5: Modellgröße für *Chain Reaction*.

Wie schon im *Connect Four*-Beispiel sind zwischen den passiven Verfahren ähnliche Strukturen zu erkennen. Zuletzt liefert jedoch die Laufzeit in Abbildung 6.6 noch einmal interessante Einblicke.

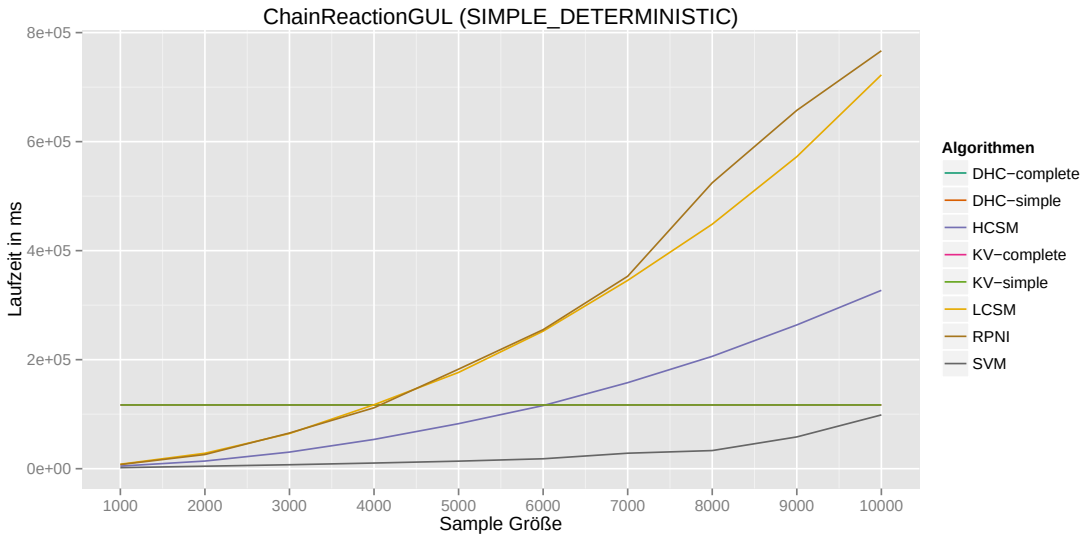


Abbildung 6.6: Modelldauer für *Chain Reaction*.

Wie schon im vorherigen Beispiel, erreicht die SVM erneut die beste Laufzeit der passiven Verfahren. Im Falle der modellbasierten Verfahren zeigt sich jedoch, dass der HCSM einen beachtlichen Vorsprung gegenüber den übrigen Verfahren hat. Bei großen und komplexen Problemen scheint also die Modellgröße eine stärkere Gewichtung zu erhalten, als der algorithmische Overhead (in Form von Sortieren). Damit ist also die Modellgröße, welche bisher einen scheinbar unwichtigen Faktor darstellte – von der Interpretierbarkeit der Modelle abgesehen – in manchen Anwendungsfällen durchaus relevant.
Vergleichsweise aussagelos ist die Messung der aktiven Verfahren. Durch die verfrühte Terminierung erreicht der KV-Ansatz eine Vergleichsweise gute Laufzeit – gegeben der Modellgüte ist aber selbst diese geringe Zeit verschwendet. Die Laufzeit des DHC-Algorithmus' ist gar nicht erst eingetragen, weil dieser nach 40 Stunden noch nicht terminierte und der Benchmark vorzeitig abgebrochen wurde. Dies wirkt – gegeben Abbildung 6.3 – zunächst unplausibel, aber bereits bei *Connect Four* gibt es KIs bei denen der DHC-Ansatz ein Vielfaches der Laufzeit des KV-Ansatzes benötigt.

6.2 Inferenzqualität

Waren die bisherigen Analysen eher abstrakt und allgemein gehalten, ist es im Kontext dieser Projektgruppe von besonderer Interesse zu betrachten, wie die unterschiedlichen Verfahren bei der Hauptaufgabe – dem Inferieren von Gegnern – abschneiden. Dazu wurden die gelernten Modelle aus dem vorherigen Abschnitt genommen (die passiven Verfahren haben dafür auf allen 10000 Beispielen gelernt) und der Inferenz-Komponente übergeben.
Im Falle von *Connect Four* standen die vier KIs AlphaBeta, Clocky, Downy sowie Lefty (s. Abschnitt 2.3.1.1 sowie Abschnitt 2.3.1.2) zur Verfügung. Es wurde jeweils eine Instanz des Spiels mit der entsprechenden KI initialisiert und den entsprechenden Modellen der Lernverfahren als mögliche Gegner der Inferenz übergeben. Die Ergebnisse sind in Abb. 6.7 dargestellt.
Im Falle der passiven Verfahren `RPNI, HCSM` und `LCSM`, konnte die Inferenz in einem von vier Fällen den richtigen Gegner bestimmen. Auffällig dabei war jedoch, dass die Inferenz-Komponente unabhängig von der tatsächlichen KI in allen vier Fällen jeweils die gleiche Antwort gegeben hat. Es scheint bei den DFAs das Problem zu existieren, dass diese sich zu ähnlich sind, um korrekt auseinander gehalten zu werden. Dieses Problem manifestiert sich bei den Modellen der SVM. Auf Basis dieser

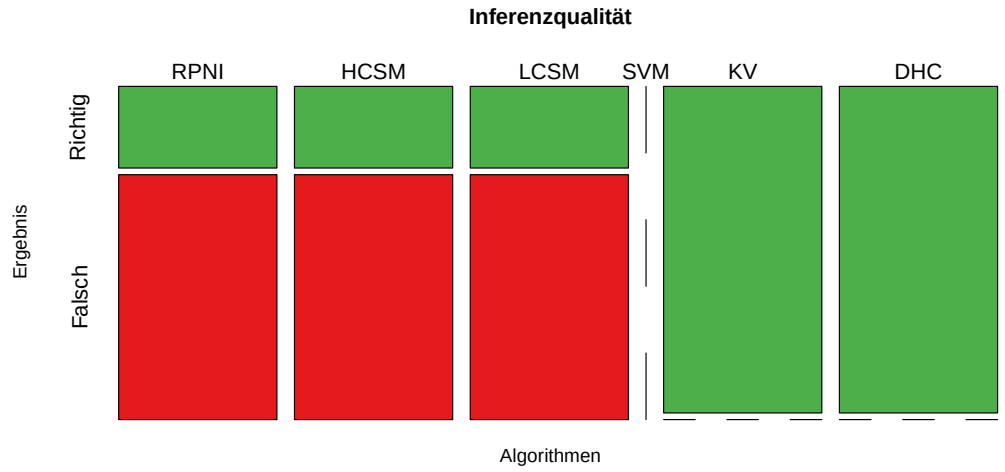


Abbildung 6.7: Inferenzqualität der Verfahren für *Connect Four*.

Modelle konnte die Inferenz-Komponente keine sinnvollen Entscheidungen treffen, da entweder alle möglichen Gegner-Kandidaten gleichzeitig eliminiert wurden, oder in einer Endlosschleife keine Modelle eliminiert werden konnten. Entgegen der guten Ergebnisse der SVM im vorherigen Kapitel zeigt sich also, dass sich die gelernten Modelle – zumindest in der hier gezeigten Art und Weise – nicht für die Inferenz eignen.

Die Modelle der aktiven Lernalgorithmen hingegen konnten in allen Fällen erfolgreich genutzt werden, den richtigen Gegenspieler zu identifizieren. Insgesamt war die Dauer, bis die Inferenz-Komponente eine Entscheidung getroffen hat, durchweg kurz. Es hat – mit Ausnahme der SVM – nie länger als fünf Iterationen benötigt, bis die Inferenz-Komponente mit einer Entscheidung terminierte.

Die Analyse für das Spiel *Chain Reaction* wurde ausgelassen, da zum einen – wie bereits erwähnt – einige Verfahren aufgrund der hohen Laufzeit vorzeitig abgebrochen wurden und damit die entsprechenden Modelle nicht zur Verfügung standen und zum anderen lediglich die Anbindung einer einzigen KI existiert, dessen Inferenz trivial ist.

7 | Fazit

Im Verlauf der Projektgruppe wurde das aktive und passive Lernen betrachtet. Das Ziel war es, ein Framework zu entwickeln, welches zum Lernen von Anwendungen und Erkennen von Gegnern genutzt werden kann. Dabei kommen passive und aktive Lernverfahren zum Einsatz. Sowohl beim passiven als auch beim aktiven Lernverfahren greift das Framework auf viele verschiedene Lernalgorithmen zurück (z. B. SVM, DHC, L*). Einerseits wird das aktive Lernen genutzt, um eine Anwendung zu lernen. Andererseits gibt es eine Implementierung, welche Samples generiert und diese Samples dann für das passive Lernen nutzt. Darüber hinaus wurde ein Interface, die GUL, entwickelt, welches genutzt wird, um jede Anwendung, die in eine GUL überführt werden kann, zu lernen. Dadurch wurde die Komplexität der Lernverfahren weitesgehend vereinfacht, so dass jeder Nutzer mit Hilfe des Frameworks seine Anwendungen auf einfache Weise lernen kann. Das gesamte Framework kommuniziert dabei über einen JMS Nachrichtensystem, welches die Komponenten so voneinander trennt, dass sie verteilt und asynchron ausgeführt werden können.

7.1 Zusammenarbeit im Team

Die Zusammenarbeit der einzelnen Teammitglieder lässt sich als sehr gut beschreiben. Für die Entwicklung des Frameworks wurden kleinere Gruppen gegründet, welche sich aus Spezialisten für die benötigten Bereiche zusammensetzten. Dabei wurde auch darauf geachtet, dass die Gruppenmitglieder großes Interesse an ihrem jeweiligen Bereich besaßen.
Hier wurde zu Beginn in den kleineren Gruppen ein Plan zur Implementierung der einzelnen Komponenten entwickelt und dieser dann den anderen Gruppen vorgestellt. Dieser Vorgehensplan wurde anschliessend von den übrigen Gruppen auf Korrektheit überprüft und für die Implementierung bewilligt. Der Nachteil lag allerdings darin, dass die einzelnen Teilgruppen zwar eine Menge Wissen in ihrem Gebiet besaßen, aber anderen Gebiete größtenteils keine Aufmerksamkeit schenkten. Dies führte

zu Problemen in Diskussionen, in denen Änderungen oder Implementierungen meist sehr detailliert erklärt werden mussten, damit jeder diese verstehen konnte. Dadurch wurde der Planungsprozess unweigerlich verlängert, da oft sehr viel Zeit zur Präsentation der vorläufigen Ergebnisse genutzt werden musste.
Grundsätzlich war jedes Gruppenmitglied bei Problemen oder Fragen über verschiedene Kommunikationsportale erreichbar. Hier wurde großen Wert auf die Kommunikation gelegt, so dass Probleme oder Fragen in angemessener Zeit gelöst oder beantwortet werden konnten.
Große Schwierigkeiten bereitete der Projektgruppe der Umgang mit Maven. Zu Beginn gab es einige Probleme aufgrund der Komplexität des Build-Management-Tools. Nach einiger Eingewöhnungszeit überwiegten die Vorteile, wie das Dependency Management, allerdings gegenüber der Komplexität. Rückblickend kann man sagen, dass es sich gelohnt hat, sich mit Maven zu beschäftigen, da es ein wichtiges Tool im Bereich des Projektmanagement ist.
Die Definition von Zielen für die Projektgruppe stellte sich als schwierig heraus, da die Problemstellung den Mitgliedern der Projektgruppe viel Freiraum gelassen hat. Daher konnte der konkrete Fokus selbst gewählt werden. Hier wurden zu Beginn die groben Ziele definiert, die dann im Verlauf der Projektgruppe angepasst wurden. Dieser Vorgang entspricht dem Paradigma der agilen Softwareentwicklung (Siehe [Gmb14b]). Die Projektgruppe hatte aus diesem Grund die Aufgabe, die neuen Ziele zu adaptieren und bereits erarbeitete Ergebnisse dementsprechend anzupassen. Hinzu kommt, dass alle definierten Ziele erfolgreich abgeschlossen werden konnten.

7.2 Evaluierte Techniken

Zu Beginn der Projektgruppe wurde von einer Teilgruppe recherchiert, in wie weit Fourier Transformationen zur Generierung von Entscheidungsbäumen genutzt werden können (Siehe [KP04]). Hierzu wurden Prototypen entwickelt und überprüft, ob ein sinnvoller Einsatz innerhalb des Frameworks möglich ist. Es stellte sich allerdings heraus, dass diese Methodik nicht praktikabel ist und wurde somit nicht weiter verwendet.
Des Weiteren wurde ein regelbasierter Ansatz mit CHR evaluiert (Siehe [Fru09]). Hierbei erfolgte zunächst eine Recherche mit anschließender Prototypentwicklung. Dabei wurde herausgefunden, dass bei sehr einfachen Anwendungen, wie zum Beispiel das Spiel Tic-Tac-Toe, die Regeln bereits sehr komplex werden, so dass ein effektiver Einsatz in dem Framework nicht möglich war.

7.3 Umgesetzte Funktionen im Framework

Das Framework bietet die Möglichkeit weitere Anwendungen zu adaptieren und diese beim Lernvorgang zu benutzen. Dies ist möglich, indem man eine GUL entwickelt, die der Anwendung entspricht. Hier wurde darauf geachtet, dass die GUL so allgemein wie möglich gehalten wurde um damit die größtmögliche Anzahl an Anwendungen zu unterstützen. Dadurch ist das Framework sehr vielseitig und unterstützt viele Anwendung. Hier muss darauf hingewiesen werden, dass zu dem Spiel auch mindestens eine Künstliche Intelligenz existieren muss, welche durch das Framework gelernt werden kann. Dies wird durch ein bereitgestelltes Interface ermöglicht, welches zur Generierung einer künstlichen Intelligenz genutzt werden muss. Nachdem man ein neues Spiel erstellt, bzw. eine GUL implementiert hat, benötigt man nur noch eine KI. Nun ist es möglich, mit Hilfe des Frameworks das Spiel zu lernen und die perfekte Gegnerstrategie zu erzeugen, so dass die KI immer besiegt wird. Die Tatsache, dass sich die Anwendungen und KIs immer im deterministischen Bereich befinden, sorgt dafür, dass eine 100% Gewinnrate erreicht werden kann.
Anstatt einen Nachrichtenserver mit Hilfe von Java Sockets zu erstellen, wurde JMS deshalb gewählt, da dadurch die Netzwerkprogrammierung weggefallen ist. Das führt dazu, dass die Kommunikation innerhalb des Frameworks auf einem JEE Standard basiert, welches den Zugriff auf diverse Messaging Services bietet. So musste man sich nicht mit Sicherheit, dem Öffnen und Schließen von Sockets und der korrekte Übertragung von Nachrichten befassen. Außerdem ermöglicht der Nachrichtenserver eine Plattform- und programmiersprachenunabhängige Kommunikation. Die Kommunikation ist asynchron, was zu einer zeitlichen Entkopplung zwischen Beauftragung und Bearbeitung einer Nachricht führt. Dies ermöglicht, dass die einzelnen Komponenten unabhängig voneinander ausgeführt werden können, weil keine konkrete Verbindung zwischen den Komponenten besteht. Hinzu kommt, dass das Framework die Auslagerung der einzelnen Komponenten auf unterschiedliche Computersysteme unterstützt, um dadurch einen Performancegewinn zu erzielen.

7.4 Ausblick

Die Frage, welches Lernverfahren das bessere ist, konnte im Verlauf der PG nicht geklärt werden, da beide Verfahren ihre Stärken und Schwächen haben. Wenn

man vom Standpunkt der Inferenz ausgeht, bietet das aktive Lernen einen Vorteil gegenüber dem Passiven. Das aktive Lernen ist hier deshalb besser, da die Inferenz schneller in der Lage ist, herauszufinden, um welchen Gegner es sich handelt. Allerdings berechnet die Inferenz immer nur den nächsten Zug, um zu einem Ergebnis zu gelangen. Hier wäre es möglich ein *AlphaBeta-Pruning* einzubauen, welches die Inferenz so erweitert, dass mehr als ein Zug beachtet wird (vgl. Kapitel 5.1). Diese Erweiterung bietet aber nur für das passive Lernen eine Geschwindigkeitsvorteil, da beim aktiven Lernen die Inferenz in den vorliegenden Anwendungen nach ein bis zwei Schritten bereits ein Ergebnis liefert. Beim passiven Lernen dauert dieser Teil länger und benötigt ebenfalls mehr Schritte.
Im Gegensatz zur Inferenz ist der Lernvorgang beim passiven Lernen sehr viel schneller abgeschlossen als beim aktiven Lernen. Dabei schließen die nicht-modellbasierten Lernverfahren (siehe 6.1) in den empirischen Experimenten überraschend gut ab. Eine genauere Untersuchung der Zusammenhänge zwischen den verschiedenen Modellrepräsentationen – im Konkreten: DFAs und Hyperebenen – bildet damit ein interessantes Thema für zukünftige Arbeiten.

Obwohl der Lernvorgang beim passiven Lernen schneller abläuft, kann das aktive Lernen dennoch besser bei einfachen Anwendungen wie *Connect4* eingesetzt werden, weil es genauere Ergebnisse bei der Inferenz bietet. Die Laufzeit ist dabei nur geringfügig höher. Bei komplexen Anwendungen, wie *Chain Reaction* bietet das aktive Lernen allerdings keinen Vorteil mehr. Die Laufzeit des Lernverfahrens wird zu hoch.
Hier wäre die Kombination aus beiden Lernverfahren ein guter Ansatz um ein performantes Lernverfahren zu entwickeln. Es wäre möglich, durch das passive Lernen einen Startpunkt zu schaffen, so dass viele irrelevante Züge bereits durch das passive Lernverfahren aussortiert werden. Das aktive Lernen wird dann dazu benutzt, das Ergebnis so weit zu verbessern, dass auch die Inferenz sehr schnell funktioniert. Auch kann das Framework als Startpunkt genommen werden, um ein nichtdeterministisches Lernverfahren zu implementieren. Hierfür müsste die Inferenz um Wahrscheinlichkeiten erweitert werden, so dass ein Zug nur dann gewählt wird, wenn dieser die größtmögliche Gewinnchance bietet. Leider kann dafür nur das passive Lernen genutzt werden. Beim aktiven Lernen ist dies nicht möglich.

Ein weiterer sehr wichtiger Punkt zur Verbesserung des Frameworks ist die Unterstützung von Parallelität beim Lernvorgang. Durch Parrallelität kann der Lernvorgang vorraussichtlich weiter beschleunigt werden, so dass das aktive Lernen noch schneller funktioniert. Die `Learnlib` bietet hierfür bereits einige Implementierungen.

Des Weiteren wäre ein zusätzlicher Schritt das Verändern der Abstraktionsebene. Mehrere Züge könnten zu einem Schritt zusammengefasst werden, um so den entstehenden Graphen zu verkleinern. So könnten die ersten sechs Züge bei der Anwendung `Connect4` zusammengefasst werden als ein möglicher Startpunkt, da der erste mögliche Gewinnzustand frühstmöglich im siebten Zug eintreffen kann.

Ein weiterer Schritt wäre der Einbau einer intelligenten `Reset`-Funktion. Anstelle eines Zurücksetzens des gesamten Spielfeldes nach einem Durchlauf könnte ein vordefinierter Startspielfeldzustand gewählt werden. So könnte man bereits begonnene Spiele lernen.

Abbildungsverzeichnis

1.1 Zusammenspiel der einzelnen Komponenten. 4

2.1 Darstellung der einzelnen Komponenten in dem Framework. 7
2.2 Eine einfache *Chain Reaction* Kettenreaktion 21

3.1 Ein beispielhafter $PTA(\mathcal{S})$. 26
3.2 $PTA(\mathcal{S}_+)$ für das RPNI-Beispiel. 27
3.3 Zwischenzeitlicher DFA nach Verschmelzung von q_1 und q_2. 28
3.4 Automat nach Verschmelzung der beiden Teil-Automaten. 29
3.5 $PTA(\mathcal{S}_+)$ mit aktualisierten Einfärbungen. 30
3.6 Vorläufiges Ergebnis des RPNI-Algorithmus'. 30
3.7 Ein beispielhafter $PTA(\mathcal{S})$ mit Coverage-Informationen. 31
3.8 Bestimmen einer Hyperebene mit maximaler Breite 33
3.9 Nicht-Lineare Transformation in höher-dimensionalen Raum 36

4.1 Graphische Darstellung eines Testdrivermodells [MIH+12] 48

5.1 Inferenzbeispiel - Initialisierung 64
5.2 Inferenzbeispiel - Schritt 1 64
5.3 Inferenzbeispiel - Schritt 2 65
5.4 Inferenzbeispiel - Schritt 3 65

6.1 Modellqualität für *Connect Four* mit der KI Downy. 69
6.2 Modellgröße für *Connect Four* mit der KI Downy. 70
6.3 Modelldauer für *Connect Four* mit der KI Downy. 71
6.4 Modellqualität für *Chain Reaction*. 72
6.5 Modellgröße für *Chain Reaction*. 73
6.6 Modelldauer für *Chain Reaction*. 73
6.7 Inferenzqualität der Verfahren für *Connect Four*. 75

Algorithmenverzeichnis

2.1 Hauptprogramm AlphaBeta-Pruning 16
2.2 Negamax 17

4.1 Grundlegender Lernablauf 49
4.2 Flexibler Lernalgorithmus 51

5.1 optimalTurn 60
5.2 EnemyInference 60

Quellcodeverzeichnis

2.1 GUL-Interface 11

4.1 XML Schema für die Konfiguration von aktiven Lernprozessen 56

Literaturverzeichnis

[AJ06] Adriaans, Pieter ; Jacobs, Ceriel: Using MDL for Grammar Induction. In: Sakakibara, Yasubumi (Hrsg.) ; Kobayashi, Satoshi (Hrsg.) ; Sato, Kengo (Hrsg.) ; Nishino, Tetsuro (Hrsg.) ; Tomita, Etsuji (Hrsg.): *Grammatical Inference: Algorithms and Applications* Bd. 4201. Springer Berlin Heidelberg, 2006, S. 293–306

[All88] Allis, Victor: A knowledge-based approach to connect-four. The game is solved: White wins. In: *Master's thesis, Vrije Universiteit*, 1988

[Ang87] Angluin, Dana: Learning regular sets from queries and counterexamples. In: *Information and Computation* 75 (1987), Nr. 2, 87 - 106. http://www.sciencedirect.com/science/article/pii/0890540187900526. – ISSN 0890–5401

[Bew10] Bewersdorff, Jörg: *Glück, Logik und Bluff. Mathematik im Spiel - Methoden, Ergebnisse und Grenzen.* Wiesbaden : Vieweg+Teubner Verlag, 2010

[Bis06] Bishop, Christopher M.: *Pattern Recognition and Machine Learning.* Secaucus, NJ, USA : Springer-Verlag New York, Inc., 2006

[Cho78] Chow, Tsun S.: Testing Software Design Modeled by Finite-State Machines. In: *IEEE Transactions on Software Engineering* 4 (1978), May, Nr. 3, S. 178–187

[CV95] Cortes, Corinna ; Vapnik, Vladimir: Support-Vector Networks. In: *Mach. Learn.* 20 (1995), September, Nr. 3, S. 273–297

[DBL10] Bias Variance Decomposition. In: Sammut, Claude (Hrsg.) ; Webb, Geoffrey I. (Hrsg.): *Encyclopedia of Machine Learning.* Springer, 2010. – ISBN 978–0–387–30768–8, S. 100–101

[Dor14] Dortmund, TU: *G-Dur berät erfolgreich: Data-Mining Spezialist "Rapid-I" erhält 50.000 Euro Preis*. http://www.cs.tu-dortmund.de/nps/de/Aktuelles/Newsarchiv/2008/2008_02_22_G-Dur/index.html, 2014

[Fru09] Fruehwirth, Thom: *Constraint Handling Rules*. Cambridge University Press, 2009. – ISBN 978–0–521–87776–3

[GHJV10] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. München, Deutschland : Addison-Wesley Verlag, 2010

[GMB14a] GMBH, RapidMiner: *Rapidminer*. http://rapidminer.com/, März 2014. – Homepage des Herstellers

[Gmb14b] GmbH it-agile: *Was ist agile Softwareentwicklung?* http://www.it-agile.de/wissen/methoden/agilitaet/, 2014

[Hig10] Higuera, Colin de l.: *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010

[HMU03] Hopcroft, John E. ; Motwani, Rajeev ; Ullman, Jeffrey D.: *Introduction to automata theory, languages, and computation*. Second Edition. Addison-Wesley, 2003. – I–XIV, 1–521 S. – ISBN 978–0–321–21029–6

[How13] Hower, Falk: *Active learning of interface programs*, TU Dortmund, Diss., June 2013. http://hdl.handle.net/2003/29486

[HSM11] Howar, Falk ; Steffen, Bernard ; Merten, Maik: Automata Learning with Automated Alphabet Abstraction Refinement. In: *Verification, Model Checking, and Abstract Interpretation* Bd. 6538. Springer Berlin Heidelberg, 2011. – ISBN 978–3–642–18274–7, S. 263–277

[HTF09] Hastie, Trevor ; Tibshirani, Robert ; Friedman, Jerome: *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. Springer, 2009 (Springer Series in Statistics)

[IHS14] Isberner, Malte ; Howar, Falk ; Steffen, Bernhard: The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. In: Bonakdarpour, Borzoo (Hrsg.) ; Smolka, Scott A. (Hrsg.): *Runtime*

Verification Bd. 8734, Springer International Publishing, 2014 (Lecture Notes in Computer Science). – ISBN 978–3–319–11163–6, 307-322

[Koh95] Kohavi, Ron: A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In: *International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 1995, S. 1137–1143

[KP04] Kargupta, H. ; Park, Byung-Hoon: A Fourier spectrum-based approach to represent decision trees for mining data streams in mobile environments. In: *Knowledge and Data Engineering, IEEE Transactions on* 16 (2004), Feb, Nr. 2, S. 216–229. `http://dx.doi.org/10.1109/TKDE.2004.1269599`. – DOI 10.1109/TKDE.2004.1269599. – ISSN 1041–4347

[KV94] Kearns, Michael J. ; Vazirani, Umesh V.: *An Introduction to Computational Learning Theory*. Cambridge, MA, USA : MIT Press, 1994. – ISBN 0–262–11193–4

[LCTD14a] LS5-CS-TU-DORTMUND: *Connect4*. `http://connectit.cs.uni-dortmund.de/maven-site/connectfour/`, 2014

[LCTD14b] LS5-CS-TU-DORTMUND: *PGHEISS*. `http://ls5-www.cs.tu-dortmund.de/cms/de/extra/HeISs/PG-Beschreibung/index.html`, 2014

[LPP98] Lang, Kevin J. ; Pearlmutter, Barak A. ; Price, Rodney A.: Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm. In: *Proceedings of the 4th International Colloquium on Grammatical Inference*. London, UK, UK : Springer-Verlag, 1998 (ICGI '98), S. 1–12

[Lug02] Luger, George F.: *Künstliche Intelligenz . Strategien zur Lösung komplexer Probleme*. 5. München : Pearson Studium, 2002

[Mat14] Mathematik.de: *Kapitel 2*. `http://www.mathematik.de/spudema/spudema_beitraege/beitraege/vorberger/kapitel2.htm`, 2014

[Mer13] Merten, Maik: *Active automata learning for real life applications*, TU Dortmund, Diss., January 2013. `http://hdl.handle.net/2003/29884`

[MH99] Mark Hapner, Rahul S. Rich Burridge B. Rich Burridge: *Java^{TM} Message Service*. http://docs.oracle.com/cd/E19957-01/816-5904-10/816-5904-10.pdf, November 1999

[MIH+12] Merten, Maik ; Isberner, Malte ; Howar, Falk ; Steffen, Bernhard ; Margaria, Tiziana: Automated Learning Setups in Automata Learning. Version: 2012. http://dx.doi.org/10.1007/978-3-642-34026-0_44. In: Margaria, Tiziana (Hrsg.) ; Steffen, Bernhard (Hrsg.): *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change* Bd. 7609. Springer Berlin Heidelberg, 2012. – ISBN 978–3–642–34025–3, 591-607

[MRS05] Margaria, Tiziana ; Raffelt, Harald ; Steffen, Bernhard: Knowledge-based Relevance Filtering for Efficient System-level Test-based Model Generation. In: *Innovations in Systems and Software Engineering* 1 (2005), July, Nr. 2, S. 147–156

[Ner58] Nerode, A.: Linear automaton transformations. In: *Proceedings of the American Mathematical Society* 9 (1958), Nr. 4, S. 541–544

[Nie03] Niemann, Heinrich: *Klassifikation von Mustern*. 2. Universität Erlangen, 2003. – http://www5.cs.fau.de/fileadmin/Persons/NiemannHeinrich/klassifikation-von-mustern/m00-www.pdf

[OG92] Oncina, José ; García, Pedro: Identifying regular languages in polynomial time. In: *Advances in Structural and Syntactic Pattern Recognition* 5 (1992), S. 99–108

[Pep05] Pepicelli, Glen: *Bitwise Optimization in Java: Bitfields, Bitboards, and Beyond*. http://www.onjava.com/pub/a/onjava/2005/02/02/bitsets.html, 2005

[SHM11] Steffen, Bernhard ; Howar, Falk ; Merten, Maik: Introduction to Active Automata Learning from a Practical Perspective. In: Bernardo, Marco (Hrsg.) ; Issarny, Valérie (Hrsg.): *Formal Methods for Eternal Networked Software Systems* Bd. 6659. Springer Berlin Heidelberg, 2011. – ISBN 978–3–642–21454–7, S. 256–296

[Wik13] Wikipedia: *Strategie*. http://de.wikipedia.org/wiki/Strategie, September 2013

[WIK14] WIKIPEDIA.DE: *Sogo*. `http://de.wikipedia.org/wiki/Sogo`, September 2014

[WK91] Weiss, Sholom M. ; Kulikowski, Casimir A.: *Computer Systems That Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1991. – ISBN 1–55860–065–5

[WNS+13] Windmueller, Stephan ; Neubauer, Johannes ; Steffen, Bernhard ; Hower, Falk ; Bauer, Oliver: Active continuous quality control. In: *In Proceedings of CBSE ACM* (2013)