Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Knowledge Engineering
Prof. Dr. Johannes Fürnkranz

KNOWLEDGE ENGINEERING

Meta-Lernen einer Evaluierungs-Funktion für einen Regel-Lerner

Diplomarbeit

Sven Burges

Betreuung durch
Prof. Dr. Johannes Fürnkranz

Dezember 2006

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, Dezember 2006

Zusammenfassung

Regel-Lern Algorithmen des Separate-and-Conquer oder Covering Typus lernen Regeln, indem sie gefundene Regeln schrittweise verfeinern. Zur Bewertung aller möglichen Verfeinerungen der Regel werden in der Praxis verschiedenste Masse herangezogen (z.B. precision, information gain, weighted relative accuracy, etc). Diese bewerten jedoch ausschliesslich die Fähigkeit der Regel, zwischen den Beispielen der Klasse zu unterscheiden.
Ziel dieser Diplomarbeit ist es, einen alternativen Ansatz zu testen, der versucht, eine optimale Such-Heuristik für das Regel-Lernproblem zu lernen. Zur Lösung dieses Problems muss ein einfacher Reinforcement Learning Algorithmus in einen bestehenden Regel-Lern-Algorithmus in der Weka Data Mining Library in Java integriert und getestet werden, sowie seine Performanz mit konventionellen Regel-Lern-Systemen verglichen werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Kapitel 1

Einleitung

Vorbemerkung Die folgende Einleitung beinhaltet eine informelle Beschreibung des in dieser Diplomarbeit behandelten Meta-Lernproblems, das durch das Lernen einer Suchheuristik gegeben ist, welche in Regel-Lern-Algorithmen eingesetzt werden kann. Den Anfang dieser Beschreibung bildet zunächst eine knappe Beschreibung des sogenannten "Konzept-Lernen", das eine Gattung spezieller Lernprobleme bezeichnet, zu deren Lösung Regel-Lern-Algorithmen eingesetzt werden. Basierend darauf werden in den anschliessenden Absätzen die Merkmale von Regel-Lern-Algorithmen vorgestellt, wobei insbesondere die Bedeutung des "Suchheuristik"-Begriffs erläutert wird, was für die abschliessende Beschreibung des Meta-Lernproblems notwendig ist.

Konzept-Lernen: Regel-Lern-Algorithmen werden häufig zum Ableiten von Konzeptbeschreibungen aus vorgegebenen Datenmengen verwendet - dem sogenannten "Konzept-Lernen". Im einfachsten Fall besteht eine solche Datenmenge aus mehreren relationalen Datensätzen, die Objekte anhand von Attributen beschreiben. Eines dieser Attribute stellt dabei das sogenannte Klassenattribut dar. Bei diesem Attribut handelt es sich um ein bool'sches Attribut mit den zulässigen Attributwerten *true* und *false*. Die beim "Konzept-Lernen" zu bewältigende Aufgabe besteht nun im Ableiten einer Funktion aus den vorliegenden Datensätzen, die für ein beliebiges Objekt - dessen Beschreibung nicht unbedingt in der Datenmenge enthalten sein muss - den Wert des Klassenattributs in Abhängigkeit seiner übrigen Attributwerte vorhersagt. Diese gesuchte Funktion wird in der Fachsprache als "Zielkonzept" bezeichnet.

Beispiel: Eine für das "Konzept-Lernen"exemplarische Datenmenge wird in der relationalen Datenbanktabelle 1.1 auf Seite 9 dargestellt[1]. Diese Datenmenge beschreibt anhand der fünf Attribute *outlook*, *temparature*, *humidity*, *windy* und *play* 14 verschiedene Objekte, indem jedes dieser Objekte genau einem in der Tabelle enthaltenen Datensatz zugeordnet wird.

Ein einzelnes Attribut definiert eine Menge von Attributwerten, die bei der Beschreibung eines Objekts in der Tabelle verwendet werden dürfen. Die Attributwerte der fünf genannten Attribute werden in der folgender Auflistung angeführt, wobei das *play*-Attribut das Klassenattribut repräsentiert:

outlook: {overcast, rainy, sunny};

temperature: {cool, hot, mild};

humidity: {high, normal};

windy: {no, yes};

play: {false, true};

[1] Beispiel wurde der Arbeit [Quinlan, 1986] entnommen.

Tabelle 1.1: Beispielhafte Datenmenge für das Konzept-Lernen

#	**outlook**	**temperature**	**humidity**	**windy**	***play***
1	sunny	hot	high	no	*false*
2	sunny	hot	high	yes	*false*
3	overcast	hot	high	no	*true*
4	rainy	mild	high	no	*true*
5	rainy	cool	normal	no	*true*
6	rainy	cool	normal	yes	*false*
7	overcast	cool	normal	yes	*true*
8	sunny	mild	high	no	*false*
9	sunny	cool	normal	no	*true*
10	rainy	mild	normal	no	*true*
11	sunny	mild	normal	yes	*true*
12	overcast	mild	high	yes	*true*
13	overcast	hot	normal	no	*true*
14	rainy	mild	high	yes	*false*

Semantisch betrachtet, stellt jeder dieser 14 Datensätze ein Erfahrungsbeispiel dafür dar, wie sich das Wetter in Abhängigkeit der am Tag zuvor vorgefundenen Wetterbedingungen verhalten hat. Die am Vortag gemessenen Wetterdaten werden dabei duch die vier ersten Attribute beschrieben. Das *play*-Attribut hingegen zeigt an, ob das am Folgetag vorgefundene Wetter schön genug für ein Picknick in freier Natur gewesen ist, was genau dann der Fall ist, wenn der entsprechende Datensatz für das *play*- Attribut den *true*-Wert annimmt.

Das aus dieser Beispielmenge zu lernende Zielkonzept stellt nun eine Funktion h (Gleichung 1.1) dar,

$$h : outlook \times temperature \times humidity \times windy \rightarrow play \tag{1.1}$$

welche für einen beliebigen Tag - beschrieben anhand der Attribute *outlook*, *temperature*, *humidity* und *windy* - den korrekten *play*-Wert für den kommenden Tag vorhersagt.

Lern-Algorithmen: Das Erlernen eines Zielkonzepts aus einer vorgegebenen Datenmenge ist die Aufgabe von "Lern-Algorithmen". Diese Algorithmen führen ihre Aufgabe aus, indem sie aus einer Menge ihnen zur Auswahl stehenden Funktionen - dem sogenannten "Hypothesenraum" - eine möglichst "korrekte" Hypothese suchen, d.h. eine Hypothese, die im Idealfall identisch mit gesuchten Zielkonzept ist.

Die Grösse und die Struktur des Hypothesenraums wird durch die zum Lern-Algorithmus gehörende "Konzeptbeschreibungssprache" definiert. Jedes Element dieser Konzeptbeschreibungssprache repräsentiert eine mögliche Hypothese und wird in der Fachsprache als "Konzeptbeschreibung" bezeichnet.

Eine besondere Gattung von Lern-Algorithmen stellen die sogenannten Regel-Lern-Algorithmen dar, bei denen die jeweiligen Konzeptbeschreibungssprachen durch Regel-Mengen definiert sind.

Regeln: Syntaktisch betrachtet stellt eine einzelne Regel eine Datenstruktur dar, welche aus einer Menge von Selektoren und einem Attributwert des Klassenattributs besteht. Hierbei wird die Menge der Selektoren als "Regelkörper" und der Attributwert als "Regelkopf" bezeichnet. Ein einzelner Selektor wiederum wird durch einen Zweier-Tupel $\langle att, val \rangle$ repräsentiert, der aus Attribut att und einem für dieses Attribut zulässigen Attributwert val besteht.

Die durch eine Regel r definierte Hypothese h_r wird in Gleichung 1.2 angeführt, wobei die Regel r aus $n \geq 0$ Selektoren $\langle att_1, val_1 \rangle$,..., $\langle att_n, val_n \rangle$ und dem Regelkopf + besteht.

$$\texttt{if } (att_1 = val_1) \texttt{ and } \ldots \texttt{ and } (att_n = val_n) \texttt{ then } + \tag{1.2}$$

Die Gleichung 1.2 drückt aus, dass die Hypothese h_r genau dann den +-Wert für einen beliebigen Datensatz berechnet, wenn der Datensatz alle $n \geq 0$ im `if`-Konstrukt enthaltenen Bedingungen erfüllt, wobei die Bedingungen durch die n Selektoren der Regel r definiert werden. Falls ein Datensatz nicht alle Bedingungen erfüllt, so ist das Ergebnis von h_r als logischer Negationswert von + zu interpretieren. Der Datensatz erfüllt hierbei eine einzelne Bedingung $(att_i = val_i)$, $0 \leq i \leq n$, genau dann, wenn sein Attributwert für das att_i-Attribut dem Wert val_i gleicht. Falls ein Datensatz o alle Bedingungen einer Regel r erfüllt, so wird dieser Umstand mit den Worten "Regel r deckt Datensatz o ab" beschrieben.

Beispiel: Ein Regel-Beispiel für die in Tabelle 1.1 enthaltenen Datensätze ist folgende Regel:

$$\texttt{if } (temperature = cool) \texttt{ and } (humidity = normal) \texttt{ then } true$$

Diese Regel deckt in der genannten Tabelle vier Datensätze ab: nämlich die Datensätze 5,6,7 und 9, wobei die Regel für Datensatz 6 fälschlicherweise den *true*-Wert berechnet. Für insgesamt 8 der in der Tabelle enthaltenen 14 Datensätze - 3,4,5,6,10,11,12,13 - gibt die Regel den falschen Wert des entsprechenden *play*-Attributs an.

Eine besondere Regel stellt die $\top$-Regel dar, die durch einen leeren Regelkörper ohne Selektoren gekennzeichnet ist. Ihr Hypothese $h_\top$ deckt alle Datensätze ab und berechnet für jeden Datensatz den im Regelkopf enthaltenen Attributwert.

Regel-Lern-Algorithmen: Ein simpler Regel-Lern-Algorithmus wird in Abbildung 1.1 dargestellt.

Dieser Regel-Lern-Algorithmus sucht nach einer Konzeptbeschreibung, indem er, ausgehend von einer Regel mit leerem Regelkörper (Zeile 3), so lange

```
1  procedure FindBestRule ( examples , concept)
2  {
3    initRule := ⟨ ∅,concept ⟩
4    initVal := EvaluateRule(initRule, examples)
5    bestRule := ⟨ initVal,initRule ⟩
6    candidate :=  bestRule
7
8    while (RefineRule(candidate, examples) ≠ ∅)
9    {
10     refinements := RefineRule( candidate, examples)
11     rule := ⟨ ε,-∞ ⟩
12     for refinement ∈ refinements
13     {
14       evaluation := EvaluateRule(refinement, examples)
15       newRule := ⟨ evaluation,refinement ⟩
16       if ( newRule > candidate )
17       {
18         candidate := newRule
19       }
20     }
21     if( candidate > bestRule )
22     {
23       bestRule := candidate
24     }
25   }
26   return (bestRule)
27 }
```

Quelltext 1.1: Einfacher Regel-Lern-Algorithmus

Bedingungen an diesen Regelkörper hinzufügt, bis keine Bedingungen mehr zur Auswahl stehen. Diejenige unter den hierbei erzeugten Regeln, welche vom Algorithmus als korrekteste Konzeptbeschreibung betrachtet wird, stellt das vom Regel-Lern-Algorithmus berechnete Ergebnis dar.

Das Auswählen und Hinzufügen von Bedingungen erfolgt dabei in einer Folge sogenannter "Verfeinerungsschritte", wobei jeder Verfeinerungsschritt genau einen vollständigen Durchlauf der *while*-Schleife umfasst. Die erste in einem Zeitschritt auszuführende Tätigkeit besteht im Bestimmen aller zulässigen Bedingungen, die an den Regelkörper derjenigen Regel hinzugefügt werden dürfen, auf welche die Variable `candidate` verweist. Diese Aufgabe erledigt in Pseudo-Code 1.1 die Methode `RefineRule`, die zudem noch jede der Bedingungen an den Regelkörper einer Kopie der `candidate`-Regel hinzufügt. Die hieraus entstandenen Regeln ("Regelverfeinerungen") bilden anstatt der Bedingungen den Rückgabewert von `RefineRule`.

Nach dem Bestimmen aller zulässigen Regelverfeinerungen der `candidate`-Regel wird vom Algorithmus die beste unter diesen Regeln zu ermittelt. Für die hierbei stattfindende Regelbewertung wird eine sogenannte "Suchheuristik"

verwendet, welche jeder Regel einen Evaluierungswert zuweist. Dieser Evaluierungswert ist abhängig von den Eigenschaften der Regel auf der dem Lernproblem zugrunde liegenden Beispielmenge, sowie von der Suchheuristik selbst. Die Regelverfeinerung, welche hierbei den höchsten Evaluierungswert erzielt, wird am Ende des Verfeinerungsschritts der `candidate`-Variablen zugewiesen. Sie bildet somit diejenige Regel, für die im nächstfolgenden Verfeinerungsschritt - wiederum nach dem selben Muster - eine zulässige Bedingung ausgewählt wird. Die Suchheuristik wird in Pseudo-Code 1.1 durch die Methode `EvaluateRule` symbolisiert.

Wie bereits oben erwähnt, terminiert der in Abbildung 1.1 dargestellte Regel-Lern-Algorithmus genau dann, wenn keine weitere Bedingung gefunden werden kann, die an den Regelkörper der `candidate`-Regel hinzugefügt werden darf. Dies ist genau dann der Fall, wenn die Methode `RefineRule` in Zeile 8 die leere Menge berechnet.

Der Rückgabewert des in Abbildung 1.1 dargestellten Regel-Lern-Algorithmus ist diejenige Regel, welche den höchsten Evaluierungswert unter all denjenigen Regeln erzielt hat, die zur Laufzeit anhand der Suchheuristik `EvaluateRule` bewertet worden sind.

Veranschaulichung als Graphenproblem: Aus mathematischem Blickwinkel betrachtet, kann der in Abbildung 1.1 dargestellte Regel-Lern-Algorithmus als heuristisches Verfahren zur Lösung eines Graphenproblems veranschaulicht werden.

Der Graph $G = (V, E)$, den der Algorithmus zur Laufzeit untersucht, wird aus seiner Konzeptbeschreibungssprache abgeleitet. Dabei repräsentieren die in ihr enthaltenen Konzeptbeschreibungen die Knotenmenge V des Graphen. Die Knotenmenge V wird formal - unter Zuhilfenahme der im Quelltext 1.1 verwendeten Hilfsmethode `RefineRule` - durch folgende beiden Gleichungen definiert:

$$\begin{array}{ll} \text{(a)} & \top \in V; \\ \text{(b)} & r \in V \rightarrow \texttt{RefineRule}(r) \subseteq V; \end{array} \tag{1.3}$$

Die in der Menge $E \subseteq V \times V$ enthaltenen Elemente stellen die Kanten des Graphen G dar. Eine einzelne Kante $\langle a, b \rangle$ symbolisiert hierbei den Umstand, dass die Regel b eine Regelverfeinerung von Regel a ist. Die Kantenmenge E wird formal durch folgende Gleichung definiert:

$$\langle r_1, r_2 \rangle \in E \Longleftrightarrow r_1 \in V \wedge r_2 \in \texttt{RefineRule}(r_1); \tag{1.4}$$

Die typische Struktur eines Graphen, der anhand der beiden Gleichungen 1.3 und 1.4 definiert worden ist, wird auf Seite 13 in Abbildung 1.1 grafisch angedeutet.

In dieser Grafik werden die Regeln durch Kreise und Quadrate repräsentiert. Dabei wird eine Regel r, für die es keine Regelverfeinerung gibt (d.h., falls gilt: $\texttt{RefineRule}(r) = \emptyset$), durch ein Quadrat symbolisiert (z.B. Knoten o und m). Der oberste Knoten $\top$ kennzeichnet die $\top$-Regel. Da sie die einzige Regel ist,

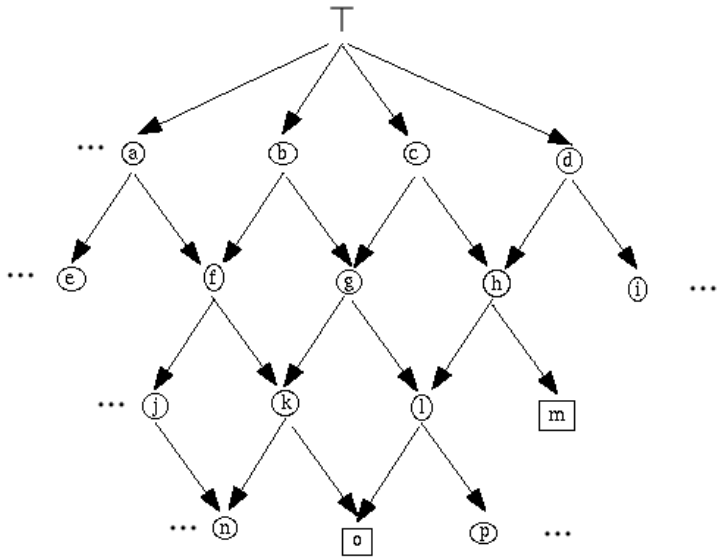

Abbildung 1.1: Graphendarstellung der Konzeptbeschreibungssprache des Regel-Lern-Algorithmus

die niemals eine Regelverfeinerung von einer beliebigen Regel sein kann, besitzt der entsprechende $\top$-Knoten keine Vorgängerknoten.

Wenn man nun unter Berücksichtigung des in Abbildung 1.1 skizzierten Graphen das Laufzeitverhalten des in Quelltext 1.1 abgebildeten Regel-Lern-Algorithmus näher betrachtet, so ergibt sich, dass der Algorithmus während der Laufzeit einen Pfad durch den Graphen beschreitet, und der Rückgabewert diejenige auf dem Pfad liegende Regel ist, welche den höchsten Evaluierungswert durch die Suchheuristik `EvaluateRule` erzielt hat.

Der beschrittene Pfad $r_1, r_2, r_3, \ldots, r_n$ ("Verfeinerungspfad") enthält dabei genau diejenigen Regeln, auf welche die Variable `candidate` jeweils bei Überprüfung der `while`-Bedingung (Zeile 8) zu Beginn der Zeitschritte $t = 1, 2, 3, \ldots, n$ zeigt. Die Regel r_1 stellt hierbei die $\top$-Regel dar, welche alle Beispiele aus der Beispielmenge *examples* abdeckt und die Regel r_n wird im Graphen durch einen quadratischen Knoten symbolisiert. Die übrigen Regeln $r_2, r_3, \ldots, r_{n-1}$ werden im Graphen durch Kreise visualisiert.

Ein exemplarischer vom Regel-Lern-Algorithmus (Quelltext 1.1) beschrittener Verfeinerungspfad wird in Abbildung 1.2 dargestellt, wobei zusätzlich für jede Regel der entsprechende von der Suchheuristik `EvaluateRule` berechnete Evaluierungswert rechts neben den jeweiligen Knoten im Graphen angegeben wird.

Der in Abbildung 1.2 dargestellte Verfeinerungspfad umfasst die Regeln $\top$, c, h, l und o. Die Regel l ist dabei diejenige Konzeptbeschreibung, welche der

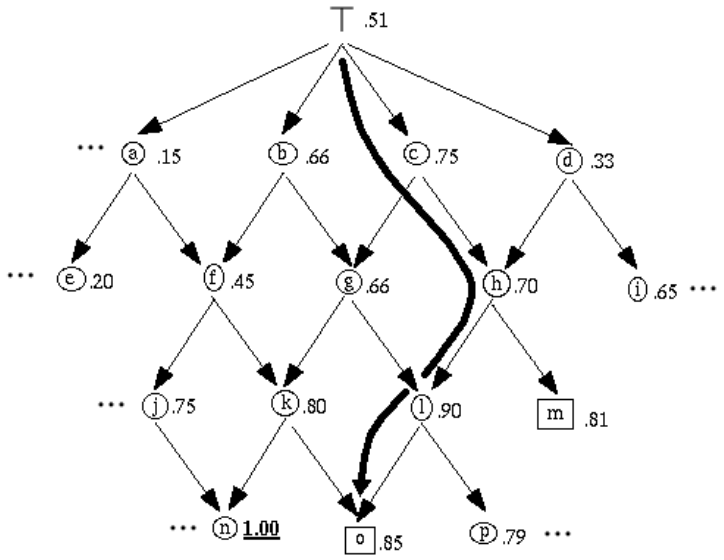

Abbildung 1.2: Von Regel-Lern-Algorithmus berechneter Verfeinerungspfad

Regel-Lern-Algorithmus am Laufzeitende als Ergebnis seiner Berechnungen zurückgibt, da sie den höchsten Evaluierungswert unter den im Pfad enthaltenen Regeln besitzt.

Meta-Lernproblem: Aus dem in Abbildung 1.2 skizzierten Graphen wird ersichtlich, dass die Suchheuristik `EvaluateRule` unter anderem dafür zuständig ist, den Verfeinerungspfad durch den Graphen zu bestimmen ("Steuerungsfunktion"). Denn sie entscheidet in einem beliebigen Verfeinerungsschritt durch die Höhe ihrer Funktionswerte über die beste Regelverfeineurng, und damit über die Richtung, welche der Verfeinerungspfad einschlägt.

Die hierbei in [Fürnkranz, 2003b] (Kap 6, 3.Absatz) aufgeworfene Frage ist, inwieweit die gängigen in Regel-Lern-Algorithmen verwendeten Suchheuristiken zur Festlegung optimaler Verfeinerungspfade geeignet sind. Der Ursprung dieser Fragestellung basiert auf der Erkenntnis, dass eine konventionelle Suchheuristik konzipiert worden ist, um die Eignung einer Konzeptbeschreibung für die Darstellung des gesuchten Konzepts zu bewerten ("Korrektheit"). Als Steuerungsfunktion betrachtet, sollte die Aufgabe einer Suchheuristik jedoch darin bestehen, in den Verfeinerungsschritten diejenigen Regelverfeinerungen auszuwählen, so dass auf dem dadurch beschrittenen Verfeinerungspfad die korrekteste Konzeptbeschreibung liegt. Gemäss [Fürnkranz, 2003b] herrscht noch Unklarheit darüber, ob auf einem durch die Regel-Korrektheit bestimmten Verfeinerungspfad auch tatsächlich die korrekteste Regel gefunden wird.

Hinsichtlich dieser Problematik wird nun in [Fürnkranz, 2003b] dazu ange-

regt, mittels "Reinforcement Learning" eine Suchheuristik `EvaluateDirection` zu lernen, die ausschliesslich darauf spezialisiert ist, als Steuerungsfunktion zu wirken. Der Wert `EvaluateDirection`(r), den diese Suchheuristik zu einer gegebenen Regel r berechnet, wird als das "Verfeinerungspotential" der Regel r bezeichnet. Dieser Wert repräsentiert einen empirisch gemessenen Schätzwert für die maximal zu erwartende Korrektheit unter denjenigen Regeln, die den mit r beginnenden Suchpfad konstituieren, wenn die Suchheuristik `EvaluateDirection` in den folgenden Verfeinerungsschritten diejenigen Regelverfeinerungen auswählt, unter denen sich statistisch betrachtet die korrekteste von r erreichbare Regelverfeinerung befindet.

Unter der Voraussetzung, dass die Suchheuristik `EvaluateDirection` erlernbar ist, wird der in Abbildung 1.1 beschriebene Regel-Lern-Algorithmus modifiziert, indem er nun zwei verschiedene Suchheuristiken `EvaluateDirection` und `EvaluateRule` verwendet, wobei `EvaluateDirection` ausschliesslich zur Bestimmung des Verfeinerungspfads zuständig ist, und `EvaluateRule` nur die Aufgabe zukommt, die Korrektheit von Regeln zu berechnen[2]. Dieser modifizierte Regel-Lern-Algorithmus wird in Abbildung 1.2 dargestellt.

Von Interesse sind nun die Schlussfolgerungen, die aus den Ergebnissen eines Leistungsvergleich beider Regel-Lern-Algorithmen gewonnen werden können. Wenn zum Beispiel der modifizierte Regel-Lern-Algorithmus signifikant bessere Konzeptbeschreibungen erzeugen würde, so wäre dies als ein Indiz dafür zu betrachten, dass die Korrektheit einer Regelverfeinerung nicht das beste Kriterium darstellt, nach dem Verfeinerungspfade bestimmt werden sollten. Wenn dieser Fall eintreten würde, so wäre es ausserdem von Interesse das Verhalten der Suchheuristik `EvaluateDirection` zu studieren, um weitere Erkenntnisse über die Eigenschaften einer optimalen Suchheuristik zu erhalten.

Die vorliegende Diplomarbeit greift den in [Fürnkranz, 2003b] geäusserten Ansatz zum Erlernen einer Suchheuristik `EvaluateDirection` mittels "Reinforcement Learning" auf, woraus sich folgende Aufgabenstellung ergibt:

Aufgabenstellung:

1. Modellieren der Suche nach Suchheuristik `EvaluateDirection` mittels "Reinforcement Learning";
2. Implementieren des sogenannten $Q(\lambda)$-Lern-Algorithmus in der "Weka Data Mining Library" zur Lösung des Meta-Lernproblems;
3. Durchführen von Experimenten, in denen durch Anwenden des $Q(\lambda)$-Lern-Algorithmus Suchheuristiken gelernt und analysiert werden;

Kapitelübersicht: Das folgende Kapitel 2 beinhaltet eine knappe Beschreibung grundlegender Konzepte und Fachbegriffe aus der Wissenschaft des "Maschinellen Lernen", auf denen sich der Inhalt der anschliessenden Kapitel stützt. So enthält dieser Grundlagen-Abschnitts z.B. eine Definition des "Lern"-Begriffs

[2] Annahme: Suchheuristik `EvaluateRule` bemisst die tatsächlich die Korrektheit.

```
1  procedure FindBestRule ( examples , concept)
2  {
3    initRule := ⟨ ∅,concept ⟩
4    initVal := EvaluateRule(initRule, examples)
5    bestRule := ⟨ initVal,initRule ⟩
6    candidate :=  bestRule
7
8    while (RefineRule(candidate, examples) ≠ ∅)
9    {
10     refinements := RefineRule( candidate, examples)
11     rule := ⟨ ε,-∞ ⟩
12     for refinement ∈ refinements
13     {
14       evaluation := EvaluateDirection(refinement, examples)
15       newRule := ⟨ evaluation,refinement ⟩
16       if ( newRule > candidate )
17       {
18         candidate := newRule
19       }
20     }
21     evaluation := EvaluateRule(candidate, examples)
22     if( candidate > bestRule )
23     {
24       bestRule := candidate
25     }
26   }
27   return (bestRule)
28 }
```

Quelltext 1.2: Einfacher Regel-Lern-Algorithmus (modifiziert)

(Kap. 2.1), sowie Zusammenfassungen über das "Konzept-Lernen" (Kap. 2.2), über die "regelbasierten Lernverfahren" (Kap. 2.3) und über das "Reinforcement Learning" (Kap. 2.4).

Das auf den Grundlagen-Teil folgende Kapitel 3 beinhaltet eine formale Spezifikation des in dieser Diplomarbeit behandelten Meta-Lernproblems. In diesem Kapitel wird versucht, das in der Einleitung informell beschriebene Meta-Lernproblem exakt zu beschreiben.

Basierend auf der in Kapitel 3 beschriebenen formalen Spezifikation, wird in Kapitel 4 das vorliegende Meta-Lernproblem mit *"Reinforcement Learning"* modelliert.

Das anschliessende Kapitel 5 enthält die Beschreibung eines auf dem $Q(\lambda)$-Lernen basierenden Lern-Algorithmus, der zur Lösung des in Kapitel 4 definierten Lernproblem im Rahmen dieser Arbeit entwickelt worden ist.

Im Anschluss an die Beschreibung des Lern-Algorithmus erfolgt in Kapitel 6 die Dokumentation (inklusive Auswertung) einer Reihe von Experimenten, in denen jeweils eine Java-Implementierung des Lern-Algorithmus auf einem

Rechner als Prozess ausgeführt worden ist.

Den Abschluss des vorliegenden Textes bildet eine Zusammenfassung (Kap. 7), in der ein Überblick über die in dieser Arbeit gewonnenen Erkenntnisse gegeben wird.

Kapitel 2

Grundlagen

In diesem Kapitel werden die in **dieser** Diplomarbeit verwendeten Fachbegriffe definiert, deren Kenntnis zum vollständigen Verständnis des vorliegenden Textes notwendig ist. Die einzelnen Definitionen wurden hierbei in einer aufeinander aufbauenden Reihenfolge angeordnet, so dass der Text ohne Unterbrechung des Leseflusses gelesen werden kann.

Den Anfang bildet hierbei die in dieser Arbeit gültige Interpretation des "Lern"-Begriffs (Def. 2.1.1). Auf dieser Definition aufbauend wird das sogenannte "Konzept-Lernen" definiert (Kap. 2.2). Wiederum darauf aufbauend wird eine Menge strukturgleicher Lernverfahren beschrieben, die zum "Konzept-Lernen" eingesetzt werden: die sogenannten "S&C"-regelbasierten Lernverfahren (Kap. 2.3). Die anschliessenden Unterkapitel enthalten eine knappe Einführung in *"Reinforcement Learning"* (Kap. 2.4).

2.1 Allgemeines zum "Lern"-Begriff

Die in dieser Arbeit gültige Interpretation des "Lern"-Begriffs ist durch folgende Definition 2.1.1 gegeben ([Mitchell 1997], Kap 1.1, Seite 2).

Definition 2.1.1 (Lernen) *A computer program is said to* ***learn*** *from* experience *experience E with respect to some class of* tasks *T and* performance measure *P, if its performance at tasks in T, as measured by P, improves with experience E.*

Hervorgehend aus dieser Definition, wird in der nächsten Definition der Begriff "Lernproblem" festgelegt.

Definition 2.1.2 (Lernproblem) *Ein* ***Lernproblem*** *wird durch ein Dreier-Tupel $\langle E, T, P \rangle$ definiert*[1]*, indem den Variablen E, T und P Datenstrukturen mit folgender Bedeutung zugewiesen:*

E: *Eine Datenmenge ("**E**xperience");*

A: *Die Aufgabe, die anhand von Datenmenge **E** gelernt werden soll ("**T**ask");*

P: *Die Evaluierungsfunktion, anhand der das Gelernte bewertet - d.h. der Lernerfolg gemessen wird ("**P**erformance measure");*

Das maschinelle Lernen - d.h. das Lösen eines vorgegebenen Lernproblems durch den Einsatz von Rechnern - wird durch die Ausführung eines entsprechenden Algorithmus bewirkt, der als **Lern-Algorithmus** bezeichnet wird.

2.2 Induktives Konzept-Lernen

Der Begriff "Konzept-Lernen" (engl.:*"Concept Learning"*) bezeichnet das Lernen eines sogenannten Zielkonzepts aus einer Menge repräsentativer Beispiele.

[1] ([Mitchell 1997], Kap 1.1, Seiten 2 bis 3.)

In den folgenden Zeilen werden - in systematischer Reihenfolge - elementare Grundbegriffe (z.B. Attribut, Beispiel, Zielkonzept usw) definiert, auf denen sich die formale Definition des "Konzept-Lernen" stützt. Den Anfang hierbei bildet die Definition des "Attribut"-Begriffs.

2.2.1 Begriffsdefinitionen

Definition 2.2.1 (Attribut) *Ein Attribut A wird durch Festlegen des Namen, den Typ und der Domäne definiert.*

- *Der Name eines Attributs dient als Bezeichner und ist eine beliebige (nicht leere) Zeichenkette;*
- *Die Domäne $\mathcal{D}_A$ ist eine beliebige (nicht leere) Menge von Symbolen, die als **Attributwerte** bezeichnet werden;*
- *Die in dieser Diplomarbeit verwendeten Attribute sind entweder **nominalen** oder **numerischen** Typs. Nominale Attribute unterscheiden sich von numerischen Attributen darin, dass auf der Domäne eines numerischen Attributs eine Ordnungsrelation definiert ist.*

Definition 2.2.2 (Instanz) *Sei $\mathcal{A} = (A_1, ...A_n)$ eine aus den n verschiedenen Attributen $A_1, \ldots, A_n$ bestehende Attributsliste, dann wird als eine "auf $\mathcal{A}$ definierte **Instanz**" ein n-dimensionales Wertetupel $\vec{x}_\mathcal{A} = \langle x_1, ..., x_n \rangle$ bezeichnet, welches folgende Eigenschaften erfüllt:*

$$\forall i \in \{, ..., n\} : x_i \in \mathcal{D}_{A_i} \cup \{\epsilon\},$$

Falls gilt $x_i = \epsilon$, so bedeutet dies, dass der entsprechende Attributwert von Instanz $\vec{x}_\mathcal{A}$ undefiniert ist.

Definition 2.2.3 (Instanzenraum) *Sei $\mathcal{A} = (A_1, ...A_n)$ eine aus den n verschiedenen Attributen $A_1, \ldots, A_n$ bestehende Attributsliste, dann ist ein auf $\mathcal{A}$ definierter **Instanzenraum** $X_\mathcal{A}$ gegeben, wenn folgende Bedingung erfüllt ist:*

$$X_\mathcal{A} \subseteq \mathcal{D}_{A_1} \times \ldots \times \mathcal{D}_{A_n}$$

Definition 2.2.4 (Zielkonzept/Konzept) *Sei ein Instanzenraum $X_\mathcal{A}$ gegeben, der auf der n-elementigen Attributsliste $\mathcal{A} = (A_1, \ldots, A_n)$ definiert ist. Dann wird als **Zielkonzept** (bzw **Konzept**) diejenige boolsche Funktion $c_{X_\mathcal{A}} : X_\mathcal{A} \rightarrow \{true, false\}$ bezeichnet, welche beim "Konzept-Lernen" über den Instanzenraum $X_\mathcal{A}$ gelernt werden soll.*

Definition 2.2.5 (Beispiel) *Sei ein auf der Attributsliste $\mathcal{A}$ definierter Instanzenraum $X_\mathcal{A}$ und ein Zielkonzept $c_{X_\mathcal{A}} : X_\mathcal{A} \rightarrow \{true, false\}$ gegeben. Dann wird als **Beispiel** für das Zielkonzept $c_{X_\mathcal{A}}$ ein Zweier-Tupel $\langle \vec{x}_\mathcal{A}, c_{X_\mathcal{A}}(\vec{x}_\mathcal{A}) \rangle$ bezeichnet.*

*Beispiele, für die gilt $c_{X_\mathcal{A}}(\vec{x}_\mathcal{A}) = true$ (bzw. $c_{X_\mathcal{A}}(\vec{x}_\mathcal{A}) = false$) werden **positive**(bzw. **negative**) Beispiele genannt.*

Definition 2.2.6 (Hypothese/Hypothesenraum) *Sei $X_{\mathcal{A}}$ ein auf der Attributsliste $\mathcal{A}$ definierter Instanzenraum. Dann wird als **Hypothesenraum** $\mathcal{H}_{\mathcal{A}}$ die Menge aller derjenigen boolschen Funktionen $h_{X_{\mathcal{A}}} : X_{\mathcal{A}} \rightarrow \{true, false\}$ bezeichnet, die beim "Konzept-Lernen" für die Approximierung des zu lernenden Zielkonzepts zur Auswahl stehen. Ein Element aus diesem Hypothesenraum $\mathcal{H}_{\mathcal{A}}$ wird als **Hypothese** bezeichnet.*

Definition 2.2.7 (Korrektheit) *Seien gegeben eine Attributsliste $\mathcal{A}$, eine auf $\mathcal{A}$ definierter Instanzenraum $X_{\mathcal{A}}$, ein Zielkonzept $c_{X_{\mathcal{A}}}$, eine beliebige Hypothese $h_{X_{\mathcal{A}}}$ aus dem Hypothesenraum $\mathcal{H}_{\mathcal{A}}$ und eine Menge von Beispielen für das Zielkonzept $c_{X_{\mathcal{A}}}$.*

*Dann wird die Hypothese $h_{X_{\mathcal{A}}}$ als **korrekt** bezeichnet, wenn $h_{X_{\mathcal{A}}}$ sowohl **vollständig**, als auch **konsistent** auf E ist; d.h, dass die Hypothese $h_{X_{\mathcal{A}}}$ für alle in E enthaltenen positiven Trainingsbeispiele den bool'schen true-Wert berechnet (Vollständigkeit), und die Hypothese für kein negatives Beispiel aus E den bool'schen true-Wert berechnet (Konsistenz).*

In der folgenden Definition 2.2.8 wird - basierend auf den Definitionen 2.1.1 bis 2.2.6 - das "Konzept-Lernen" formal definiert. Hierzu wird die in Definition 2.1.2 angeführte $\langle \mathbf{E}, \mathbf{P}, \mathbf{T} \rangle$-Taxonomie verwendet.

Definition 2.2.8 (Konzept-Lernen) *Gegeben ist eine Attributsliste $\mathcal{A} = (A_1, \ldots, A_n)$ und ein auf $\mathcal{A}$ definierter Instanzenraum $X_{\mathcal{A}}$. Ausserdem sei die Existenz eines Zielkonzept eines Zielkonzepts $c_{X_{\mathcal{A}}}$ vorausgesetzt. In Anlehnung an Definition 2.1.2 wird das "**Konzept-Lernen**" auf folgende Weise definiert:*

E: *Die Daten anhand derer gelernt wird ist gegeben durch eine sogenannte **Beispielmenge**, die aus einer Menge E aus Beispielen besteht und welche zusätzlich folgende Bedingungen erfüllt:*

- $\forall \langle i, o \rangle \in E . i \in X_{\mathcal{A}} \wedge o = c_{X_{\mathcal{A}}}(i)$;
- $\{i | \langle i, c_{X_{\mathcal{A}}}(i) \rangle \in E\} \subseteq X_{\mathcal{A}}$;
- *Die Menge der Trainingsbeispiele E muss **repräsentativ** für den Instanzenraum $X_{\mathcal{A}}$ sein ([Mitchell 1997], Kapitel 5.2).*

T: *Die Aufgabe besteht darin, eine Hypothese $h \in \mathcal{H}_{\mathcal{A}}$ zu finden, welche das Zielkonzept $c_{X_{\mathcal{A}}}$ möglichst genau approximiert.*

P: *Der Lernerfolg wird durch die **Korrektheit** (siehe Def.2.2.7) der gelernten Hypothese auf der Menge aller denkbaren Beispiele gemessen werden.*

Der exakte Korrektheitswert einer Hypothese auf der Menge aller möglichen Trainingsbeispiele ist nicht berechenbar, so dass die Korrektheit in der Praxis durch statistische Berechnungsverfahren abgeschätzt wird, wie es zum Beispiel die stratifizierte 10×10-Kreuzvalidierung ([Witten/Frank, 2001], Kapitel 5.3) darstellt.

2.2.2 Bemerkungen

- In den "Konzept-Lernen"-Lernproblemen, die im vorliegenden Text betrachtet werden, wird ausnahmslos die Vorbedingung als erfüllt angenommen, dass die Beispielmengen repräsentativ für die jeweiligen Instanzräume sind.
- Wenn eine Hypothese h aus dem Hypothesenraum für eine Instanz x den boolschen Wert *true* (bzw. *false*) berechnet , so wird dieser Umstand im Text oftmals mit der Formulierung beschrieben, dass "h (die Instanz) x positiv (bzw. negativ) klassifiziert".
- Falls das Zielkonzept $c_{X_\mathcal{A}}$ eine Instanz x positiv (bzw. negativ) klassifiziert, so wird für diesen Umstand im Text die Formulierung verwendet, dass "(die Instanz) x dem Konzept $c_{X_\mathcal{A}}$ angehört" (bzw. nicht angehört).

2.3 Auf der "Separate&Conquer"-Strategie basierende regelbasierte Lernverfahren

Das vorliegende Kapitel 2.3 beinhaltet einen Überblick über regelbasierte Lern-Algorithmen, die auf der sogenannten "Separate&Conquer"-Strategie basieren ("S&C-regelbasierte Lernverfahren"). S&C-regelbasierte Lern-Algorithmen sind Lern-Algorithmen, welche beim "Konzept-Lernen" (Kapitel 2.2) eingesetzt werden und deren Hypothesenraum aus Regelmengen besteht. Die Aufgabe eines S&C-regelbasierten Lern-Algorithmus ist das Bestimmen einer Regelmenge aus dem Hypothesenraum, welche das zu lernende Zielkonzept möglichst korrekt approximiert. Jeder regelbasierter S&C-Lern-Algorithmus ist durch den in Abbildung 2.1 (Seite 37) enthaltenen generischen S&C-Algorithmus darstellbar.

Die "Separate&Conquer"-Eigenschaft beschreibt hierbei die algorithmische Vorgehensweise, nach der ein S&C-regelbasierter Lern-Algorithmus eine Regelmenge aus dem Hypothesenraum auswählt. Ein beliebiger S&C-Lern-Algorithmus lernt eine Regelmenge, indem er in zeitlich aufeinanderfolgenden Schritten Regeln lernt und am Ende des jeweiligen Schritts die in diesem Zeitschritt gelernte Regel der Regelmenge hinzufügt. Dabei werden am Ende eines jeden Zeitschritts, nachdem eine Regel zur Regelmenge hinzugefügt worden ist ("Conquer"), die von der hinzugefügten Regel positiv klassifizierten Beispiele aus der Beispielmenge entfernt ("Separate"), weswegen die Beispielmenge von Schritt zu Schritt schrumpft. Das Lern-Algorithmus fügt dabei so lange gelernte Regeln zur Regelmenge hinzu, bis die Beispielmenge keine positiven Beispiele enthält. Das Lernen einer einzelnen Regel wird dabei als **Regelinduktion** bezeichnet.

Eine exaktere Definition der "Separate&Conquer"-Eigenschaft wird in Quelltext 2.1 abgebildet. In diesem Quelltext erzeugt die Methode `SeparateAndConquer` (Zeilen 1 bis 15) so lange durch wiederholten Aufruf der Methode `FindBestRule` (Zeile 6) Regeln und fügt diese zu der dabei entstehenden Hypothese `theory` hinzu, bis die Beispielmenge `examples` keine positiven Beispiele enthält (`positive(examples)` $\neq \emptyset$).

Der weitere Kapitelverlauf beschränkt sich bei der Beschreibung S&C-regelbasierter Lern-Algorithmen nur auf diejenigen Details, die zum Verständnis des vorliegenden Texts notwendig sind. Eine ausführliche Beschreibung S&C-regelbasierter Lern-Algorithmen enthält Arbeit [Fürnkranz, 1999].

Die Kapitelgliederung orientiert sich dabei an der *"Bias"*-Taxonomie ([Witten/Frank, 2001], Seite 32), die in der Literatur üblicherweise zur Charakterisierung von Lern-Algorithmen verwendet wird. Hieraus ergibt sich, dass das Kapitel in drei Unterkapitel aufgeteilt worden ist, welche den **Sprach-Bias** (2.3.1), den **Such-Bias** (2.3.2) und den **"Bias zur Vermeidung von Überanpassung"** (2.3.3) zum Inhalt haben.

2.3.1 Sprach-Bias

Der Sprach-Bias ist der vom Lern-Algorithmus implementierte Hypothesenraum und wird auch als **Konzeptbeschreibungssprache** bezeichnet [Witten/Frank, 2001]. Die von einem S&C-regelbasierten Lern-Algorithmus verwendete Konzeptbeschreibungssprache wird durch eine Menge sogenannter **Regelmengen** gebildet, wobei jede Regelmenge eine Hypothese des Hypothesenraums darstellt. In [Fürnkranz, 1999] (Kap 3) werden unterschiedliche Datenstrukturen genannt, die zur Darstellung von *Regeln* in einem S&C-regelbasierten Lern-Algorithmus verwendet werden können. In der vorliegenden Arbeit werden hierbei lediglich **konjunktive** Regeln betrachtet, die zur Vorhersage der Konzeptzugehörigkeit einer einzelnen Instanz eingesetzt werden und die dabei auf sogenannte **Selektoren**[2] zurückgreifen. Diese Regel-Datenstruktur wird in den folgenden Definitionen algebraisch spezifiziert, bevor in Definition 2.3.4 die Semantik des in dieser Arbeit verwendeten **Regelmengen**-Begriffs festgelegt wird.

Definition 2.3.1 (Bedingung (Selektor)) *Sei $\Omega = \{'\leq','>','='\}$ eine Menge aus der Mathematik bekannter Symbole zur Darstellung von Gleichungen, bzw Ungleichungen, und sei A ein Attribut mit Domäne $\mathcal{D}_A$, dann wird die Menge der zu Attribut A **gültigen Bedingungen (Selektoren)** $\mathcal{B}_A$ auf folgende Weise definiert:*

$$\mathcal{B}_A = \{\langle x, y\rangle | x \in \mathcal{D}_A \wedge y \in \Omega_A\},$$

wobei gilt $\Omega_A = \Omega$, falls es sich bei A um ein numerisches Attribut handelt. Im anderen Fall, wenn A vom nominalen Typ ist, gilt $\Omega_A = \{'='\}$.

Definition 2.3.2 (Bedingungstest) *Seien gegeben:*

- *ein beliebiges Attribut A,*
- *ein beliebiges Element $x \in \mathcal{D}_A$ aus der Domäne von A und*
- *eine beliebige Bedingung $\langle y, op\rangle \in \mathcal{B}_A$ aus der Menge für Attribut A gültigen Bedingungen.*

[2] [Fürnkranz, 1999] Kap 3.1.1

Dann wird als ***Bedingungstest*** $t_A(x, \langle y, op\rangle)$ *die Auswertung der boolschen Funktion* $t_A : \mathcal{D}_A \times \mathcal{B}_A \rightarrow \{true, false\}$ *bezeichnet; wobei* t_A *auf folgende Weise definiert wird:*

1.Fall($op =' \leq '$)**:**

$$t_A(x, \langle y, op\rangle) = \begin{cases} true & \text{falls } x \leq y \\ false & sonst \end{cases}$$

2.Fall($op =' > '$)**:**

$$t_A(x, \langle y, op\rangle) = \begin{cases} true & \text{falls } x > y \\ false & sonst \end{cases}$$

3.Fall($op =' = '$)**:**

$$t_A(x, \langle y, op\rangle) = \begin{cases} true & \text{falls } x = y \\ false & sonst \end{cases}$$

Wenn der Funktionsaufruf $t_A(x, \langle y, op\rangle)$ *den Wert true(bzw. false) berechnet, dann "****erfüllt*** *x die Bedingung* $\langle y, op\rangle$*" (bzw. "x erfüllt die Bedingung* $\langle y, op\rangle$ *nicht").*

Definition 2.3.3 (Regel,Regelkörper,Regelkopf) *Seien gegeben:*

- *eine Attributsliste* $\mathcal{A} = (A_1, \ldots, A_n)$*,*
- *die für* $\mathcal{A}$ *gültigen Bedingungsmengen* $\mathcal{B}_\mathcal{A} = (\mathcal{B}_{A_1}, \ldots, \mathcal{B}_{A_n})$ *und*
- *ein beliebiger boolscher Wert* $c \in \{true, false\}$*.*

Dann ist eine gültige auf $\mathcal{A}$ *definierte Regel* $r_\mathcal{A} = \langle B, c\rangle$ *ein Zweier-Tupel, das folgende beiden Bedingungen erfüllt:*

$$B \subseteq \bigcup_{A \in \mathcal{A}} \mathcal{B}_A$$

und

$$\forall a, b \in B. a \neq b.\ \nexists A \in \mathcal{A}. a \in \mathcal{B}_A \wedge b \in \mathcal{B}_A;$$

Die Bedingungsmenge B *wird dabei als* ***Regelkörper*** *und* c *als* ***Regelkopf*** *von Regel* $r_\mathcal{A}$ *bezeichnet.*

Definition 2.3.4 (Regelmenge) *Seien* n *verschiedene auf der Attributsliste* $\mathcal{A}$ *definierte Regeln* $r_1, \ldots, r_n$ *gegeben, dann wird die Menge* $R = \{r_1, ..., r_n\}$ *als eine auf* $\mathcal{A}$ *definierte* ***Regelmenge*** *bezeichnet.*

Definition 2.3.5 (Regelabdeckung) *Sei gegeben:*

- *eine Attributsliste* $\mathcal{A} = (A_1, \ldots, A_n)$*,*
- *eine beliebige auf* $\mathcal{A}$ *definierte Regel* $r_\mathcal{A} = \langle B, c\rangle$ *und*

- *eine beliebige auf $\mathcal{A}$ definierte Instanz $\vec{x}_\mathcal{A} = \langle x_1, \ldots, x_n \rangle$.*

Wenn $R_\mathcal{A}$ eine beliebige Menge auf $\mathcal{A}$ definierter Regeln darstellt und die Menge $X_\mathcal{A}$ einen auf $\mathcal{A}$ definierten Instanzenraum repräsentiert, dann deckt die Regel $r_\mathcal{A}$ die Instanz $\vec{x}_\mathcal{A}$ genau dann ab (bzw. nicht ab), wenn die in Gleichung 2.1 definierte Funktion covers : $R_\mathcal{A} \times X_\mathcal{A} \rightarrow \{true, false\}$ den boolschen true-Wert (bzw. false-Wert) berechnet.

$$covers(r_\mathcal{A}, \vec{x}_\mathcal{A}) = \bigwedge \{t_{A_i}(x_i, b_{A_i}) | b_{A_i} \in \mathcal{B}_{A_i} \wedge b_{A_i} \in B\} \tag{2.1}$$

Wenn h eine Regelmenge ist, dann deckt h Instanz $\vec{x}_\mathcal{A}$ genau dann ab (bzw. nicht ab), wenn folgende Bedingung erfüllt (bzw. nicht erfüllt) ist:

$$\exists r \in h.covers(r, \vec{x}_\mathcal{A}) = true \tag{2.2}$$

Analog zu Instanzen können auch Trainingsbeispiele von Regeln und Regelmengen abgedeckt werden. Ein beliebiges auf $\mathcal{A}$ definiertes Trainingsbeispiel $\langle \vec{e}_\mathcal{A}, c \in \{true, false\} \rangle$ wird genau dann von einer Regel r abgedeckt (bzw. nicht abgedeckt), wenn r die zum Trainingsbeispiel gehörenden Instanz $e_\mathcal{A}$ abdeckt (bzw. nicht abdeckt).

Darstellung von Hypothesen durch Regelmengen:

In einem S&C-regelbasiertem Lern-Algorithmus werden Hypothesen durch Regelmengen dargestellt. Seien hierbei mit $\vec{x} \in X_\mathcal{A}$ eine auf der Attributsliste $\mathcal{A} = (A_1, \ldots A_n)$ definierte Instanz und mit $r = \langle B, c \rangle$ eine auf $\mathcal{A}$ definierte Regel gegeben, dann wird die zu r gehörende Hypothese $h_r : X_\mathcal{A} \rightarrow \{true, false\}$ auf folgende Weise definiert:

$$h_r(\vec{x}) = \begin{cases} c & \text{falls } covers(r, \vec{x}) = true \\ \bar{c} & \text{sonst} \end{cases}$$

Ein S&C-regelbasierter Lern-Algorithmus lernt jedoch eine Regelmenge R, die üblicherweise aus mehreren auf $\mathcal{A}$ definierten Regeln besteht. Für alle in R enthaltenen Regeln gilt beim Konzept-Lernen folgende Gleichung:

$$\forall \langle B, c \rangle \ , \langle B', c' \rangle \in R.c = c'.$$

Hieraus ergibt sich folgende Definition für die zu R gehörende Hypothese h_R:

$$h_R(\vec{x}) = \begin{cases} c & \text{falls } \exists \langle B, c \rangle \in R.covers(\langle B, c \rangle, \vec{x}) = true \\ \bar{c} & \text{sonst} \end{cases}$$

2.3.2 Such-Bias

Der Such-Bias eines Lern-Algorithmus beschreibt die Vorgehensweise, auf die der Algorithmus den Hypothesenraum nach einer möglichst korrekten Hypothese

durchsucht. Bei einem regelbasierten S&C-Lern-Algorithmus wird dieser durch den Such-Bias des zur Regelinduktion eingesetzten Algorithmus bestimmt. Die Regelinduktion wird im generischen S&C-Lern-Algorithmus (siehe Abb. 2.1) durch die Methode `FindBestRule` (Zeilen 17 bis 46) dargestellt. Diese generische Methode wird zunächst im folgenden Kapitel 2.3.2.1 erklärt. Der Such-Bias eines S&C-Lern-Algorithmus wird gemäss [Fürnkranz, 1999] durch die Wahl des **Suchalgorithmus**, der **Suchstrategie** und der **Suchheuristik** festgelegt, deren Beschreibung jeweils in den drei an Kapitel 2.3.2.1 anschliessenden Unterkapiteln erfolgt.

2.3.2.1 Regelinduktion

Die Methode `FindBestRule` (siehe Abb. 2.1, Zeilen 17 bis 46) erhält als Eingabeparameter eine Beispielmenge `examples` (Zeile 17), aus der eine Hypothese in Form einer Regel gelernt (bzw. **induziert**) werden soll.

Die Methode `FindBestRule` lernt eine Regel, indem sie ausgehend von initialen Regel (Zeile 19) so lange neue Regeln durch Hinzufügen (bzw. Entfernen) von Bedingungen bildet, bis eine Haltebedingung erfüllt ist, und der Lernvorgang beendet wird. Die Regel, auf welche die Variable `bestRule` am Ende des Lernvorgangs verweist, stellt die von `FindBestRule` induzierte Regel dar und wird durch die `return`-Anweisung (Zeile 45) als Ergebnis zurückgegeben.

Wie in Abbildung 2.1 zu erkennen ist, besteht die Methodendefinition aus der Methodensignatur (Zeile 17), einem kurzen Initialisierungsblock (Zeilen 19 bis 22), drei ineinander geschachtelten Schleifenkonstrukten (Zeilen 23 bis 44) und der oben bereits erwähnten `return`-Anweisung (Zeile 45).

Im Initialisierungsblock (Zeilen 19 bis 22) wird die initiale Regel erzeugt (Zeile 19), evaluiert (Zeile 20) und anschliessend zusammen mit ihrem Evaluierungswert als momentan "beste" Regel notiert (Zeile 21). Weiterhin wird im Initialierungsblock noch eine sortierte Menge erzeugt, die aus Regeln besteht, welche anhand ihrer Evaluierungswerte sortiert worden sind (Zeile 22).

Die in Zeile 23 beginnende `while`-Schleife durchläuft nun so viele **Verfeinerungsschritte** (siehe Def.2.3.6), bis die in Zeile 22 initialisierte Regelmenge leer ist. Falls die Regelmenge nicht leer ist, dann werden aus dieser Menge die sogenannten **Kandidatenregeln** (siehe Def.2.3.7) entnommen (Zeilen 25,26). Zu jeder dieser Kandidatenregeln wird anschliessend eine Menge sogenannter **Regelverfeinerungen** (siehe Def.2.3.8) gebildet (Zeilen 27 bis 29); wobei eine Regelverfeinerung eine Regel darstellt, die aus einer Kandidatenregel durch Hinzufügen oder Entfernen einer Bedingung hervorgeht. Jede der hierbei erzeugten Regelverfeinerungen wird evaluiert (Zeile 32) und in die in Zeile 22 angelegte Regelmenge einsortiert (Zeilen 36,37), falls die in den Zeilen 33 und 34 definierte Bedingung erfüllt ist. Ausserdem werden die Regelverfeinerungen mit der gegenwärtig besten Regel verglichen (Zeile 38). Wenn alle Regeln in der Regelmenge einen besseren Evaluierungswert besitzen, so wird die Regelverfeinerung verworfen und damit nicht in die Regelmenge einsortiert. Falls die Regelverfeinerung aber einen noch besseren Evaluierungswert als die gegenwärtig beste Regel besitzt, so wird diese als neue beste Regel gemerkt (Zeile 39). Nachdem

alle Kandidatenregeln untersucht worden sind, die zu Beginn der `while`-Schleife (Zeile 25) ausgewählt worden waren, werden zum Abschluss der `while`-Schleife wenig erfolgsversprechende Regeln aus der sortierten Regelmenge entfernt (Zeile 43).

Begriffsdefinitionen:

- **Definition 2.3.6 (Verfeinerungsschritt)** *Bei symbolischer Ausführung der generischen Methode* `FindBestRule` *(siehe Abb. 2.1, Zeilen 17 bis 46) wird der n-te Durchlauf der* `while`*-Schleife (Zeilen 23 bis 44) als n-ter* ***Verfeinerungsschritt*** *bezeichnet.*

- **Definition 2.3.7 (Kandidatenregel)** *Als "****Kandidatenregeln*** *des n-ten Verfeinerungsschritts" (siehe Def.2.3.6) werden die im n-ten Verfeinerungsschritt durch den Aufruf der Hilfsmethode* `SelectCandidates` *(siehe Abb. 2.1, Zeile 25) erhaltenen Regeln bezeichnet.*

- **Definition 2.3.8 (Regelverfeinerung)** *Als* ***Regelverfeinerung*** *einer Kandidatenregel r des n-ten Verfeinerungsschritts (siehe Def.2.3.7) wird eine Regel genau dann bezeichnet, wenn sie im n-ten Verfeinerungsschritt (siehe Def.2.3.6) durch die Anweisung* `RefineRule(r)` *(siehe Abb. 2.1, Zeile 29) erzeugt wird. Eine Regelverfeinerung entsteht durch Kopieren der Kandidatenregel und anschliessendem Hinzufügen (bzw Entfernen) einer Bedingung an den (bzw vom) Regelkörper der kopierten Regel.*

- **Definition 2.3.9 (Initiale Regel)** *Als* ***Intiale Regel*** *eines Regelinduktionsvorgangs wird diejenige Regel bezeichnet, die bei symbolischer Ausführung der in Abbildung 2.1 dargestellten generischen Methode* `FindBestRule` *(Zeilen 17 bis 46) durch Aufruf der Hilfsfunktion* `InitializeRule` *(Zeile 19) erzeugt wird.*

- **Definition 2.3.10 (Verfeinerungspfad)** *Sei Regel r eine Regelverfeinerung des n-ten Verfeinerungsschritts (siehe Def.2.3.8) die durch n'-maliges ($n' \leq n$) Anwenden der Hilfsmethode* `RefineRule` *aus der initialen Regel (siehe Def.2.3.9) erzeugt worden ist. Dann wird als "****Verfeinerungspfad*** *von r" die $n'+1$-elementige Liste $RefPath(r) = (r_0, \ldots, r_{n'})$ bestehend aus den Regeln $r_0, \ldots, r_{n'}$ bezeichnet, welche folgende Eigenschaften besitzt:*

  - *r_0 ist die initiale Regel;*
  - *$r_{n'} = r$;*
  - *$\forall i.1 \leq i \leq n'.r_i \in$* `RefineRule`*$(r_{i-1})$;*

Wie aus Abbildung 2.1 ersichtlich wird, verwendet die Methode `FindBestRule` einige Hilfsmethoden. Dies werden in der folgenden Auflistung knapp beschrieben:

- `InitializeRule`$(Zeile19)$ - Erzeugt die initiale Regel;

- `EvaluateRule`($Zeilen 20, 32$) - Evaluiert die Qualität einer Regel;
- `SelectCandidates`($Zeile 25$) - Selektiert Kandidatenregeln aus einer Regelmenge;
- `RefineRule`($Zeile 29$) - Bestimmt zu einer Kandidatenregel eine Menge von Regelverfeinerungen;
- `StoppingCriterion`($Zeile 33$) - Dient zur Vermeidung einer Überanpassung der gelernten Hypothese an die dem Lernproblem zugrunde liegende Beispielmenge (siehe Kap. 2.3.3);
- `InsertSort`($Zeile 37$) - Sortiert eine Regel anhand ihres Evaluierungswert in eine Regelmenge an die richtige Position ein;
- `FilterRules`($Zeile 43$) - Entfernt aus einer Regelmenge wenig erfolgsversprechende Kandidatenregeln;

2.3.2.2 Suchalgorithmus:

Der bei der Regelinduktion verwendete Suchalgorithmus bestimmt die Grösse des Hypothesenraumausschnitts, der beim Lernen vollständig durchsucht wird, um daraus - stellvertretend für den gesamten Hypothesenraum - die beste Hypothese zu ermitteln. Das Spektrum von Suchalgorithmen reicht gemäss [Fürnkranz, 1999] von der vollständigen Überprüfung des Hypothesenraums (engl.: "Exhaustive Search"), bei der alle durch die Konzeptbeschreibungssprache formulierbaren Hypothesen evaluiert und verglichen werden, bis hin zum sogenannten "Hill-Climbing"-Suchalgorithmus, der den untersuchten Hypothesenraumausschnitt auf ein Minimum reduziert.

Für das im weiteren Textverlauf behandelte Meta-Lernproblem ist das Verständnis des "Hill-Climbing"-Suchalgorithmus erforderlich, weswegen dieser Suchalgorithmus im folgenden Abschnitt beschrieben wird.

[***Hill-Climbing***] Der "Hill-Climbing"-Suchalgorithmus durchsucht den Hypothesenraum, indem er ausgehend von einer initialen Regel so lange Bedingungen zu dieser Regel hinzufügt (bzw. entfernt), bis durch weiteres Hinzufügen (bzw. Entfernen) von Bedingungen keine weitere Regelverfeinerung mehr gebildet werden kann.

In der in Abbildung 2.1 dargestellten generischen Methode `FindBestRule` (Zeilen 17 bis 46) wird der "Hill-Climbing"-Suchalgorithmus dadurch implementiert, indem die Hilfsmethode `FilterRules` (Zeile 43) bei Aufruf nur die beste Regel aus der sortierten Regelmenge zurückgibt, auf welche die `rules`-Variable zeigt.

Neben "Hill-Climbing" gibt es noch eine Reihe weiterer Suchalgorithmen, welche versuchen die Schwächen von "Hill-Climbing" - jedoch auf Kosten einer erhöhten Rechenaufwands - zu beseitigen. Diese Suchalgorithmen und ein entsprechender Vergleich sind unter anderem ausführlicher Gegenstand von [Fürnkranz, 1999] (siehe Kap.4.1).

2.3.2.3 Suchstrategie:

Der **Suchstrategie**-Begriff beschreibt die Richtung, in welcher der Hypothesenraum während der Regelinduktion durchsucht wird. In [Fürnkranz, 1999] (Kap.4.2) werden hierbei folgende drei Suchstrategien genannnt, von denen die dritte lediglich eine Mischform der beiden zuerst genannten Suchstrategien darstellt:

1. *"Top-Down"*-Suchstrategie;
2. *"Bottom-Up"*-Suchstrategie;
3. *"Bidirectional"*-Suchstrategie;

Die Suchrichtung wird im generischen S&C-Algorithmus 2.1 (Seite 37) durch die Implementierung der Hilfsmethode `RefineRule` (Zeile 29) festgelegt. Wenn `RefineRule` in jedem Verfeinerungsschritt eine Bedingung zur Kandidatenregel hinzufügt, so spricht man von einer ***"Top-Down"*-Suchstrategie**; wenn aber umgekehrt die Hilfsmethode `RefineRule` in jedem Verfeinerungsschritt aus dem Regelkörper der Kandidatenregel eine Bedingung entfernt, so handelt es sich um eine ***"Bottom-Up"*-Suchstrategie**.

Die Begriffe *"Top-Down"* und *"Bottom-Up"* entspringen der Beobachtung, dass der Hypothesenraum als mathematischer Verband dargestellt werden kann. Die den Hypothesenraum konstituierenden Regeln stellen dabei die Verbandselemente dar. In einer graphischen Darstellung dieses Verbands wird nun genau dann ein Pfeil von einer gültigen Regel a zu einer gültigen Regel b eingezeichnet, wenn b durch Hinzufügen einer Bedingung an den Regelkörper von a erzeugt werden kann. Regel b wird in diesem Fall "**Spezialisierung** von a" genannt, und Regel a wird als "**Generalisierung** von b" bezeichnet. Bei der graphischen Darstellung einer Verbandsstruktur ist es dabei nun üblich, dass die Generalisierungen über den Spezialisierungen angeordnet werden. Somit befindet sich an der oberen Spitze des Verbands eine Regel mit leerem Regelkörper, welche alle Trainingsbeispiele abdeckt.

2.3.2.4 Suchheuristik:

Die bei einem S&C-regelbasierten Lernverfahren verwendete **Suchheuristik** stellt diejenige Funktion dar, anhand der die während der Regelinduktion erzeugten Regeln miteinander verglichen werden. Dabei wird die von der Suchheuristik als am besten bewertete Regel als Ergebnis des Regelinduktionsverfahren zurückgegeben. Im generischen S&C-Lernalgorithmus (Abbildung 2.1, Seite 37) wird die Suchheuristik durch die Hilfsmethode `EvaluateRule` (Zeilen 20,32) symbolisiert.

In [Fürnkranz, 1999] (Kap.4.3) werden eine Reihe Suchheuristiken aufgelistet, die Regeln nach unterschiedlichen Kriterien bewerten. Eine ausführliche Analyse dieser Heuristiken beinhaltet [Flach/Fürnkranz, 2003a].

2.3.3 *"Overfitting Avoidance"*-Bias

Die in der Praxis beim Konzept-Lernen vorgefundenen Beispielmengen, aus denen S&C-regelbasierte Lern-Algorithmen Zielkonzepte lernen, enthalten üblicherweise falsch klassifizierte Instanzen. Daher sollte ein S&C-regelbasierter Lern-Algorithmus nicht darauf abzielen, eine Hypothese zu finden, die vollkommen korrekt auf der Beispielmenge ist. Um eine solche Überangepasstheit (*"Overfitting"*) der Hypothese an die gegebenen Trainingsbeispiele zu vermeiden (*"avoid"*), kann ein S&C-regelbasierter Lern-Algorithmus eine entsprechende Strategie implementieren. Die hierzu in einem einzelnen S&C-regelbasiertem Lern-Algorithmus implementierte Strategie wird als der ***"Overfitting Avoidance"*-Bias** des Lern-Algorithmus bezeichnet. In [Fürnkranz, 1999] (Kap.5) werden die folgenden drei Strategie-Klassen identifiziert:

1. ***"Pre-pruning"*:** *"Pre-Pruning"*-Methoden versuchen bereits während der Ausführung der Regelinduktion "überangepasste" Hypothesen zu erkennen. Im generischen S&C-regelbasierten Lern-Algorithmus (siehe Abb. 2.1, Seite 37) wird die *"Pre-Pruning"*-Strategie durch die beiden Hilfsmethoden `RuleStoppingCriterion` (Zeile 8) und `StoppingCriterion` (Zeile 33) implementiert; wobei die zuerst genannte Methode berechnet, ob noch eine weitere Regel zur bisher vom Lern-Algorithmus gelernten Regelmenge hinzugefügt werden soll, und die andere Methode entscheidet, ob eine Regelverfeinerung überangepasst ist.

2. ***"Post-Pruning"*:** Im Gegensatz zu *"Pre-Pruning"*-Methoden behandeln *"Post-Pruning"*-Methoden das Problem der Überangepasstheit nachträglich, erst nachdem eine Hypothese vom Lern-Algorithmus ausgewählt worden ist. Die Hypothesenwahl erfolgt hierbei unter der Annahme, dass die Beispielmenge fehlerfrei ist, weswegen die Hypothese zunächst überangepasst ist. In der anschliessenden *"Post-Pruning"*-Phase wird die Überangepasstheit der Hypothese reduziert, indem die Hypothese systematisch modifiziert wird. Im generischen S&C-regelbasiertem Lern-Algorithmus (Abb. 2.1, Seite 37) wird das *"Post-Pruning"*-Verhalten durch geeignete Implementierung der Hilfsmethode `PostProcess` (Zeile 13) erreicht.

3. **Mischform:** Eine weitere Strategie zur Vermeidung von Überangepasstheit besteht in einer Mischform, die sowohl *"Pre Pruning"*- als auch *"Post-Pruning"*-Eigenschaften beinhaltet, und durch die versucht wird, die Stärken beider Ansätze zu vereinen (siehe [Fürnkranz, 1999], Kap 5.3).

2.4 "Reinforcement Learning" (RL)

Unter dem Begriff *"Reinforcement Learning"* (RL) wird eine spezielle Art von Lernproblemen und der zur Lösung dieser Lernprobleme eingesetzten Lern-Algorithmen zusammengefasst. Informell betrachtet, erfolgt das Lernen bei dieser Art von Lernproblemen dadurch, indem ein **Agent** durch geschicktes Ausprobieren von Handlungsalternativen versucht herauszufinden, sich in seiner

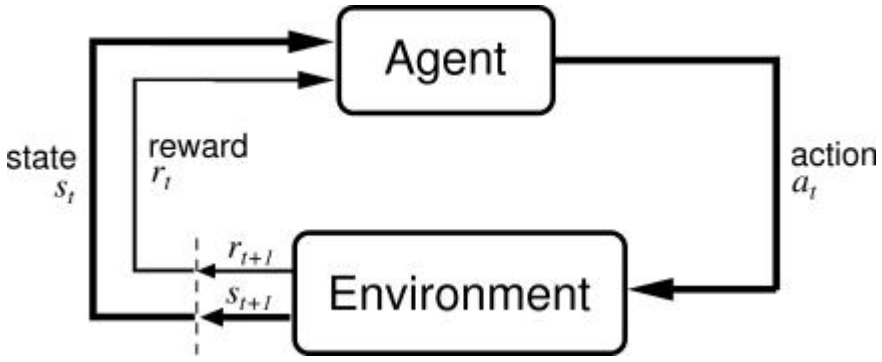

Abbildung 2.1: Interaktion zwischen Agent und Umgebung [Barto/Sutton, 1998] (Kap. 3.1)

Umgebung optimal zu verhalten. Das Lernen erfolgt dabei in einer Reihe von $t = 0, 1, 2, 3, \ldots$ Zeitschritten, wobei zu einem jeweiligem Zeitschrittbeginn der Agent über folgende Informationen verfügt:

- der Zustand s_t, in dem sich die Umgebung in Zeitschritt t befindet und
- eine Aktionsmenge $\mathcal{A}(s_t)$, welche die Menge der Handlungsalternativen darstellt, die dem Agent in Zustand s_t von der Umgebung zur Auswahl gestellt werden.

Nachdem der Agent sich für eine Handlung $a_t \in \mathcal{A}(s_t)$ entschieden hat, teilt er seine Wahl der Umgebung mit, welche daraufhin in einen Folgezustand s_{t+1} übergeht. Neben der Ermittlung des Folgezustands bewertet die Umgebung ausserdem die vom Agenten getroffene Entscheidung a_t und gibt dazu den numerischen Wert r_{t+1} als *Belohnung* an den Agenten zurück. Das Ziel des Agenten besteht nun darin herauszufinden, welche Entscheidungen in den auftretenden Umgebungszuständen getroffen werden müssen, damit die Umgebung, auf Dauer betrachtet, möglichst hohe Belohnungen zurückgibt. Das Interaktionsschema, nach dem Agent und Umgebung während eines Zeitschritts t miteinander kommunizieren, wird in Abbildung 2.1 in einer skizziert.

Eine Schema, das als Vorlage für die Defintionen konkreter RL-Lernprobleme dient, beinhaltet das folgende Unterkapitel 2.4.1.

2.4.1 *"Markov Decision Processes"* (MDP)

In der RL-Literatur erfolgt die mathematische Darstellung eines RL-Lernproblems oftmals durch Anwendung eines besonderen generischen Schemas, welches in der englischen Sprache als *"Markov Decision Process"* (MDP) bezeichnet wird. Der Vorteil einer MDP-Darstellung besteht darin, dass auf eine Reihe in der Mathematik bereits entwickelter Lösungsmethoden zurückgegriffen werden kann (z.B. "Dynamische Programmierung"), aus denen eine Menge

unterschiedlicher RL-Lernverfahren hervorgegangen sind (z.B. TD-Lernen, MC-Lernen,...). Ausserdem dient die MDP-Darstellung als Ausgangspunkt für eine mathematische Analyse, durch die unter anderem weitere Eigenschaften eines konkreten RL-Lernproblems identifiziert werden können (z.B. Komplexität).

Weil später folgende Kapitel häufig RL-Begriffe gebrauchen, wird in den folgenden Absätzen ein endlicher MDP definiert, wobei die Bedeutung wichtiger Begriffe erklärt wird.

Gemäss [Barto/Sutton, 1998] (Kap 3.1) wird als **RL-Task** die Spezifikation einer Agentenumgebung bezeichnet. Ein **endlicher MDP** stellt einen spezialisierten RL-Task dar, der durch folgende Eigenschaften vollständig definiert wird:

- Der RL-Task basiert auf einer endlichen Menge $\mathcal{S}$ möglicher Umgebungszustände.
- Die Elemente der Menge $\{\mathcal{A}(s)|s \in \mathcal{S}\}$, welche die in den Umgebungszuständen dem Agenten zur Auswahl stehenden Handlungsalternativen darstellen, stellen alle endliche Mengen dar.

  Bemerkung: Die Menge $\mathcal{A} = \bigcup_{s \in \mathcal{S}} \mathcal{A}(s)$ repräsentiert im weiteren Kapitelverlauf die Menge aller im RL-Task vorhandenen Handlungsalternativen.
- Für alle Zustandsübergänge $s \rightarrow s'$ zwischen zwei Zeitschritten t und $t+1$ $(t \geq 0)$ gelten folgende Bedingungen:
  - Die Umgebung berechnet s' anhand einer nichtdeterministischen Funktion $\delta : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$, für deren Definitionsbereich gilt:

    $$D_\delta = \{\langle s, a\rangle | s \in \mathcal{S} \, und \, a \in \mathcal{A}(s)\}.$$

    Ausserdem berechnet δ zu jedem $\langle s, a\rangle$-Wertetupel den Folgezustand, indem die Funktion eine auf der Zustandsmenge $\mathcal{S}$ definierte Wahrscheinlichkeitsverteilung $\mathcal{P}_s^a$ verwendet. $\mathcal{P}_s^a$ ordnet jedem Zustand $s' \in \mathcal{S}$ einen Wert aus dem reellwertigen Intervall $[0, 1]$ zu, der die Wahrscheinlichkeit $\mathcal{P}_{ss'}^a$ darstellt, dass die Umgebung in den Zustand s' übergeht, wenn der Agent sich für a bei vorliegenden Umgebungszustand s entscheidet.
  - Die von der Umgebung an den Agenten zurückgegebene Belohnung r' - wenn dieser sich in Umgebungszustand s für Aktion a entscheidet und die Umgebung dabei in Zustand s' übergeht - erfolgt ebenfalls anhand einer Wahrscheinlichkeitsverteilung; wobei diese Wahrscheinlichkeitsverteilung über die Menge aller möglichen Belohnungswerte gebildet wird. Der sich aus einer solchen Wahrscheinlichkeitsverteilung ergebende durchschnittliche Belohnungswert wird im weiteren Verlauf mit $\mathcal{R}_{ss'}^a$ bezeichnet:

    $$\mathcal{R}_{ss'}^a = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}.$$

- Bei einem MDP muss die Umgebung ausserdem die Markov-Eigenschaft erfüllen. Die Markov-Eigenschaft ist genau dann erfüllt, wenn die Berechnung des Folgezustands s_{t+1} und des Belohnungswerts r_{t+1} **nur** vom gegenwärtigen Umgebungszustand s_t und der in diesem Zustand getroffenen Entscheidung a_t abhängig ist.

2.4.2 Agent

Wie bereits zu Kapitelbeginn erwähnt, besteht das Ziel des Agenten im Erlernen einer Entscheidungspolitik, anhand welcher der Agent in jedem Umgebungszustand diejenige Entscheidungsalternative auswählt, so dass die hierdurch von der Umgebung erhaltenen Belohnungswerte optimiert werden.

Im vorliegenden Abschnitt wird das Ziel des Agenten mathematisch erfasst. Hierzu wird zunächst der Begriff "Entscheidungspolitik" oder "Politik" definiert, um auf dieser Definition aufbauend die vom Agenten zu erlernende Entscheidungspolitik mathematisch festzulegen. Dabei sei im gesamten restlichen Abschnitt ein endlicher MDP mit all seinen definierenden Komponenten (Zustandsmenge $\mathcal{S}$, Entscheidungsalternativen $\mathcal{A}$, ...) als gegeben angenommen, welcher die Umgebung des Agenten darstellt.

Entscheidungspolitik: Eine **Entscheidungspolitik** (oder **Politik**) $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ ist eine Funktion, die für jedes definierte Wertepaar $\langle s, a \rangle \in D_\delta$ aus dem Definitionsbereich der Zustandsübergangsfunktion δ die Wahrscheinlichkeit berechnet, mit welcher der Agent die Handlungsalternative a auswählt, wenn die Umgebung sich in Zustand s befindet. Als Politik π_t wird dabei die vom Agenten zu Beginn von Zeitschritt t gelernte Politik bezeichnet.

Im bisherigen Textverlauf wurde das Ziel des Agenten nur informal beschrieben. Eine mathematische Erfassung dieses Ziels wird durch die Einführung einer besonderen Funktion erreicht, welche eine gegebene Politik hinsichtlich der Optimalität ihres Verhaltens bewertet. In [Kaelbling, 1996] (Kap 1.2) wird diese Funktion als *"Model of Optimal Behavior"* genannt, weswegen sie in der vorliegenden Arbeit als **Optimalitätsmodelle** bezeichnet wird. Das Problem des Lernens einer optimalen Entscheidungspolitik wird somit abgebildet auf das Problem des Suchens nach einer Entscheidungspolitik, welche das zum Agenten gehörende Optimalitätsmodell optimiert.

Das Optimalitätsmodell R_t, welches in der vorliegenden Diplomarbeit von Bedeutung ist, wird in Gleichung 2.3 beschrieben.

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{2.3}$$

wobei durch den γ-Parameter, $\leq \gamma \leq 1$, festgelegt wird, welchen Einfluss zukünftige Belohnungen r_{t+2}, r_{t+3}, ..., r_∞ bei der Bewertung einer Politik haben. Je grösser γ, desto stärker werden die in den nachfolgenden Zeitschritten erhaltenen Belohnungen berücksichtigt.

Q-Funktion: Die meisten RL-Lern-Algorithmen lernen eine optimalen Entscheidungspolitik, indem sie eine sogenannte **Q-Funktionen** $Q_\pi : \mathcal{S} \times \mathcal{A}(\mathcal{S}) \rightarrow \mathcal{R}$ bestimmen, aus welcher sie unmittelbar die Politik ableiten können. Dabei repräsentiert $Q_\pi(s,a)$ denjenigen Wert, der vom Optimalitätsmodell zu erwarten ist, wenn der Agent sich im Umgebungszustand $s_t = s$ für Aktion $a_t = a$ entscheidet, und in den nachfolgenden unendlich vielen Zeitschritten $t+1$, $t+2$, ..., ∞ stur die Politik π zur Entscheidungswahl anwendet (siehe Gleichung 2.4).

$$Q_\pi(s,a) = E_\pi\Big\{R_t | s_t = s, a_t = a\Big\} = E_\pi\Big\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\Big\} \quad (2.4)$$

Da der RL-Task ein endlicher MDP ist, kann die Q-Funktion auch als rekursiv definierte Funktion dargestellt werden (siehe Gleichung 2.5).

$$Q_\pi(s,a) = \sum_{s'} \mathcal{P}^a_{ss'} \Big[\mathcal{R}^a_{ss'} + \gamma \sum_{a'} \pi(s',a') Q_\pi(s',a')\Big] \quad (2.5)$$

Im folgenden Abschnitt *"RL-Zielfunktion"* wird die Zielfunktion eines auf einem endlichen MDP (Kap 2.4.1) basierenden RL-Lernproblems definiert.

RL-Zielfunktion π^*: Das Ziel des Agenten besteht im Lernen einer Politik π^*, für die folgende Bedingung erfüllt ist:

$$\forall s \in \mathcal{S}. a \in \mathcal{A}(s). Q_{\pi^*}(s,a) = \max_\pi Q_\pi(s,a)$$

Die Q-Funktion Q_{π^*} erfüllt dabei die in Gleichung 2.6 formulierte Bedingung, welche in der Literatur als **Bellman'sche Optimalitätsbedingung** bekannt ist.

$$Q_{\pi^*}(s,a) = \sum_{s'} \mathcal{P}^a_{ss'} \Big[\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q_{\pi^*}(s',a')\Big] \quad (2.6)$$

Die Politik π^* und die Funktion Q_{π^*} werden jeweils als **optimale Entscheidungspolitik** und **optimale Q-Funktion** bezeichnet.

2.4.3 Episodische & Kontinuierliche RL-Tasks

In der Literatur wird zwischen zwei Klassen von RL-Tasks unterschieden:

- Episodische RL-Tasks und
- Kontinuierliche RL-Tasks;

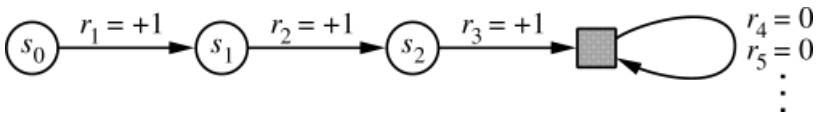

Abbildung 2.2: Zustandsübergänge bei "episodischem" RL-Task [Barto/Sutton, 1998] (Kap. 3.4)

Episodische RL-Tasks: Bei episodischen RL-Tasks werden die in den Zeitschritten $t = 0, 1, 2, \ldots$ auftretenden Zustandsübergänge in einzelne Episoden aufgetrennt. Eine Episode beginnt mit einem von der Umgebung anhand einer vorgegebenen Wahrscheinlichkeitsverteilung bestimmten Startzustand s_0 (siehe [Barto/Sutton, 1998], Kap 3.3), und sie endet, wenn die Umgebung in einen besonderen Zustand übergegangen ist: den sogenannten **Endzustand** s_T. Sobald die Umgebung den Endzustand s_T erreicht hat, bestimmt sie anhand der Wahrscheinlichkeitsverteilung einen neuen Startzustand s_0, von dem ausgehend der Agent durch Interagieren mit der Umgebung eine neue Episode erzeugt.

Die Erzeugung einer Episode wird in Abbildung 2.2 skizziert. Hierbei stellt Zustand s_0 den Beginn der Episode dar und das grau gefärbte Kästchen den Endzustand s_T der Episode. Die Episode umfasst die Zustände s_0, s_1, s_2 und s_T.

Bei episodischen Tasks wird zwischen der Menge aller zulässigen Umgebungszustände $\mathcal{S}^+$ und der Menge $\mathcal{S}$ unterschieden, die aus der Menge $\mathcal{S}^+$ durch Entfernen der Endzustände erzeugt wird.

Kontinuierliche RL-Tasks: Kontinuierliche RL-Tasks unterscheiden sich von episodischen RL-Tasks darin, dass es keine Endzustände gibt.

Auswirkungen auf das Optimalitätsmodell: In Abhängigkeit davon, ob ein RL-Task episodisch oder kontinuierlich ist, wird ein anderes Optimalitätsmodell bevorzugt. Bei episodischen RL-Tasks ist zum Beispiel das in Gleichung 2.7 dargestellte Optimalitätsmodell sinnvoll, weil ein Endzustand garantiert immer erreicht wird.

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_T \tag{2.7}$$

Im Gegensatz dazu würde bei einem kontinuierlichen RL-Tasks ein solches Optimalitätsmodell einen unendlichen Wert liefern, da es bei solchen RL-Tasks keine Endzustände gibt. Deshalb wird hierbei das in Gleichung 2.3 dargestellte Optimalitätsmodell präferiert, wobei zumindest die Ungleichung $\gamma < 1$ erfüllt sein muss.

Das in Gleichung 2.7 dargestellte Optimalitätsmodell kann aber auch mittels Optimalitätsmodell 2.3 modelliert werden. Dies wird dadurch erreicht, indem dem γ-Parameter der Wert 1 fest zugewiesen wird und indem definiert wird, dass bei Erreichen eines Endzustands s_T im Zeitschritt T die Umgebung in den folgenden Zeitschritten $T + 1$, $T + 2$, ..., ∞ den einmal erreichten Endzustand

nicht mehr verlassen kann, und der bei einem Übergang zwischen zwei Endzuständen von der Umgebung berechnete Belohnungswert immer gleich dem 0-Wert ist. In Abbildung 2.2 wird diese Form der Modellierung veranschaulicht. Die rechts im Bild eingezeichnete Schleife am grau gefärbten Kästchens deutet hierbei an, dass die Umgebung den erstmals erreichten Endzustand in den folgenden Zeitschritten nie mehr verlassen wird. Weiterhin weist die Abbildung darauf hin, dass die Belohnungswerte r_4, r_5, ... alle gleich dem 0-Wert sind.

```
1  procedure SeparateAndConquer(examples)
2  {
3    theory := ∅
4    while positive(examples) ≠ ∅
5    {
6      rule := FindBestRule(examples)
7      covered := cover(rule,examples)
8      if RuleStoppingCiterion(theory,rule,examples)
9        exit while
10     examples := examples \ covered
11     theory := theory ∪ rule
12   }
13   theory := PostProcess(theory)
14   return (theory)
15 }
16
17 procedure FindBestRule(examples)
18 {
19   initRule := InitializeRule(examples)
20   initVal := EvaluateRule(initRule,Examples)
21   bestRule := ⟨initVal,initRule⟩
22   rules := {bestRule}
23   while rules ≠ ∅
24   {
25     candidates := SelectCandidates(rules,examples)
26     rules := rules \ candidates
27     for candidate ∈ candidates
28     {
29       refinements := RefineRule(candidate,examples)
30       for refinement ∈ refinements
31       {
32         evaluation := EvaluateRule(refinement,examples)
33         unless StoppingCriterion(refinement,evaluation,
34                                  examples)
35         {
36           newRule := ⟨evaluation,refinement⟩
37           rules := InsertSort(newRule,rules)
38           if ( newRule > bestRule )
39             bestRule := newRule
40         }
41       }
42     }
43     rules := FilterRules(rules,examples)
44   }
45   return (bestRule)
46 }
```

Quelltext 2.1: Generischer S&C-Algorithmus [Fürnkranz, 1999]

Kapitel 3

Formale Spezifikation - Meta-Lernproblem

In diesem Kapitel wird das in der Einleitung informell dargestellte Meta-Lernproblem formal spezifiziert. Diese Spezifikation dient als Grundlage der im anschliessenden Kapitel 4 erfolgenden Modellierung des Meta-Lernproblems durch *"Reinforcement Learning"*.

3.1 Formale Spezifikation

gegeben:

- Eine Signatur der gesuchten Evaluierungsfunktion $h^* : Def(h^*) \rightarrow [0,1]$, wobei jedes Element aus dem Definitionsbereich $Def(h^*)$ die Eigenschaften einer gelernten Regel auf ihrer zugrunde liegenden Beispielmenge anhand reeller Zahlen beschreibt.

  Die Abbildung einer beliebigen Regel r auf das entsprechende Element des Definitionsbereich $Def(h^*)$ wird durch eine Funktion σ ausgeführt, welche die Eigenschaften von r auf der zum Lernproblem gehörenden Beispielmenge bestimmt, so dass gilt:

  $$\sigma(r, E) \in Def(h^*)$$

- Eine Menge *Datasets*, deren Elemente Beispielmengen sind (z.B. UCI-Datenbanken), die zum Konzept-Lernen geeignet sind (siehe Kap. 3.1.1);

- Eine Funktion *init*, die aus der Menge *Datasets* eine Reihe von Konzept-Lernproblemen bestimmt. Der Wert $init(Datasets)$ stellt eine Menge dar, welche aus Dreier-Tupeln $\langle E_{train}, E_{test}, concept \rangle$ besteht, wobei jedes dieser Tupel auf folgende Weise definiert ist:

  E_{train} Trainingsmenge des Lernproblems;

  E_{test} Evaluierungsmenge des Lernproblems, anhand der die Korrektheit einer gelernten Konzeptbeschreibung geschätzt wird;

  concept Identifiziert das anhand der Trainingsmenge zu erlernende Zielkonzept;

  Eine detaillierte Beschreibung der Bedingungen, welche die *init*-Funktion erfüllen muss, beinhaltet das Kapitel 3.1.2.

- Eine über der Menge $init(Datasets)$ definierte Wahrscheinlichkeitsverteilung $P_{init(Datasets)}$

  $$P_{init(Datasets)} : init(Datasets) \rightarrow [0,1],$$

  wobei die von $P_{init(Datasets)}$ zu erfüllenden Eigenschaften in Abschnitt 3.1.3 erläutert werden.

- Der Regel-Lern-Algorithmus $\texttt{FindBestRuleValue}_{h,\sigma}$ (siehe Quelltext 3.1), dessen Rückgabewert die Genauigkeit der gelernten Konzeptbeschreibung in Abhängigkeit von der Evaluierungsmenge E_{test} darstellt 3.1.4);

```
1  procedure FindBestRuleValue_{h,σ} ( E_train, E_test, concept)
2  {
3    bestRule := InitializeRule(concept)
4    bestVal := Precision(bestRule, E_test)
5    candidate := ⟨ bestVal,bestRule ⟩
6
7    while (RefineConditions(candidate, E_train) ≠ ∅)
8    {
9      conditions := RefineConditions(candidate, E_train)
10     refinements := RefineRule( candidate, conditions)
11     candidate := ⟨ ε,-∞ ⟩
12     for refinement ∈ refinements
13     {
14       evaluation := h(σ(refinement, E_train))
15       newRule := ⟨ evaluation,refinement ⟩
16       if ( newRule > candidate )
17       {
18         candidate := newRule
19       }
20     }
21     precision := Precision(candidate, E_test);
22     if( precision > bestVal )
23     {
24       bestVal := precision
25       bestRule := candidate
26     }
27   }
28   return (bestVal)
29 }
```

Quelltext 3.1: Regel-Lern-Algorithmus $\texttt{FindBestRuleValue}_{h,\sigma}$

gesucht:

- Eine Evaluierungsfunktionen h^*, für die gilt:

$$h^* = \arg\max_h \Big(\sum_{t \in init(Datasets)} P_{init(Datasets)}(t) \cdot \texttt{FindBestRuleValue}_{h,\sigma}(t) \Big)$$

Bemerkung: Das Bestimmen von $Def(h^*)$, $Datasets$, $init$ und $P_{init(Datasets)}$ erfolgt im Rahmen der später stattfindenden Experimente. Sie wurden an dieser Stelle lediglich deshalb angeführt, um sich der mathematische Natur des vorliegenden Meta-Lernproblems bewusst zu werden.

3.1.1 Datenmengen *Datasets*

Jede in der Menge *Datasets* enthaltenen Beispielmenge *examples* besitzt folgende Eigenschaften:

- Die Attributmenge $\mathcal{A}_{examples}$, auf der die in *examples* enthaltenen Trainingsbeispiele definiert sind, enthalten ausschliesslich *nominale* und *numerische* Attribute.
- Genau eines der nominalen Attribute aus der Menge $\mathcal{A}_{examples}$ ist als Klassenattribut markiert.

Die Menge *Datasets* sollte dabei so beschaffen sein, dass die *init*-Funktion die Möglichkeit erhält, für den Definitionsbereich $Def(h^*)$ geeignete Konzept-Lernprobleme zu erzeugen (siehe Kap. 3.1.2).

3.1.2 Generieren von Lernproblemen durch *init*-Funktion

Die durch die *init*-Methode erzeugte Menge an Konzept-Lernproblemen sollte im Optimalfall folgende Bedingungen erfüllen:

- Durch die Menge $init(Datasets)$ muss gewährleistet sein, dass für jedes Element des Definitionsbereichs $Def(h^*)$ ausreichend repräsentative Trainingswerte vorhanden sind.
- Für jedes in der Menge $init(Datasets)$ enthaltene Konzept-Lernproblem $\langle E_{train}, E_{test}, concept\rangle$ ist die Grösse der beiden Mengen E_{train} und E_{test} ungefähr gleich. Ausserdem ist das Verhältnis zwischen der Anzahl positiver und negativer Beispiele in den beiden Mengen E_{train} und E_{test} nahezu identisch.

3.1.3 Wahrscheinlichkeitsverteilung $P_{init(Datasets)}$

Die Wahrscheinlichkeitsverteilung $P_{init(Datasets)}$ dient dazu, Lernprobleme in einer Weise aus der Menge $init(Datasets)$ auszuwählen, so dass jedes Element des Definitionsbereichs $Def(h^*)$ ungefähr den selben Einfluss beim Lernen der Evaluierungsfunktion h^* hat.

3.1.4 Regel-Lern-Algorithmus `FindBestRuleValue`$_{h,\sigma}$

Für die im anschliessenden Kapitel 4 erfolgende Modellierung des Meta-Lernproblems durch "Reinforcement Learning" wird folgende anhand von Pseudo-Code 3.1 festgelegte Definition benötigt:

Definition 3.1.1 (Verfeinerungsschritt T_{final}) *Der Verfeinerungsschritt (siehe Def. 2.3.6) in dem der* `bestRule`*-Variable des in Abbildung 3.1 dargestellten Regel-Lern-Algorithmus diejenige Regel zugewiesen wird, auf welche die Variable auch am Ende des Lernvorgangs verweist, wird als Zeitschritt T_{final} bezeichnet.*

Kapitel 4

Modellierung des Meta-Lernproblems mit "Reinforcement Learning"

4.1 Modellierung des Meta-Lernproblems mit *"Reinforcement Learning"*

In diesem Abschnitt wird das im vorangegangenen Kapitel 3 spezifizierte Meta-Lernproblem als *"Reinforcement Learning"* - Lernproblem modelliert.

4.1.1 Umgebung

Zustandsmenge $\mathcal{S}$: Die Zustandsmenge $\mathcal{S}$ ist durch den Definitionsbereich $Def(h^*)$ der gesuchten Suchheuristik h^* gegeben. Daher repräsentiert ein beliebiger Zustand $s \in \mathcal{S}$ eine Menge von Regeln, die jeweils die selben Eigenschaften auf ihren Beispielmengen besitzen.

Aktionsmengen $\mathcal{A}(s)$: Die zu einem (Regel-)Zustand s gehörende Aktionsmenge $\mathcal{A}(s)$ symbolisiert die Menge aller Bedingungen, die an den Regelkörper der s darstellenden Regel angefügt werden dürfen, so dass dadurch zulässige Regelverfeinerungen entstehen. Die Menge $\mathcal{A}(s)$ enthält dabei für jeden möglichen Folgezustand s' von s genau eine Bedingung $a_{s'}$, was durch die Bijektion $\rho : \mathcal{A}(s) \rightarrow Def(h^*)$ ausgedrückt werden soll, die jede in $\mathcal{A}(s)$ enthaltene Regelbedingung dem entsprechenden Folgezustand zuordnet.

Eine konkrete Bedingung $c \in$ `RefineConditions`[1], aus welcher durch Hinzufügen an den Regelkörper von Regel $rule$ die Regelverfeinerung $rule'$ gebildet wird, wird durch die in Gleichung 4.1 definierte Funktion f_{rule} auf ein Element der Bedingungsmenge $\mathcal{A}(s)$ abgebildet.

$$\begin{gathered} f_{rule} : \texttt{RefineConditions}(rule) \rightarrow \mathcal{A}(s) \\ f_{rule}(c) = a \Longleftrightarrow \rho(a) = \sigma(rule') \end{gathered} \tag{4.1}$$

Zustandsübergangsfunktion $\delta : \mathcal{S} \times \mathcal{A}(\mathcal{S}) \rightarrow \mathcal{S}$: Für den Definitionsbereich $Def(\delta)$ der Zustandsübergangsfunktion δ gilt:

$$Def(\delta) = \{\langle s, a\rangle | s \in \mathcal{S} \wedge a \in \mathcal{A}(s)\}$$

Mit Hilfe der oben definierten ρ-Bijektion wird δ auf folgende Wiese definiert:

$$\delta(s, a) = \rho(a)$$

Belohnungsfunktion $r : \mathcal{S} \times \mathcal{A}(\mathcal{S}) \rightarrow [0, 1]$: Sei $task \in init(Datasets)$ das vom Regel-Lerner gegenwärtig bearbeitete Konzept-Lernproblem, für das gilt:

$$task = \langle E_{train}, E_{test}, concept\rangle.$$

Wenn der Regel-Lern-Algorithmus beim Zeitschrittübergang $t \rightarrow t + 1$ der Variablen `candidate` eine neue Regel r' zuweist, dann wird der Rückgabewert $r(\sigma(r), f_r(c))$ auf folgende Weise definiert:

[1] Siehe Quelltext 3.1 auf Seite 40 (Zeile 9).

$$r(\sigma(r), f_r(c)) = \begin{cases} \texttt{Precision}(r', E_{test}) & \text{falls } t+1 = T_{final} \text{ (siehe Def. 3.1.1);} \\ 0 & \text{sonst;} \end{cases}$$

wobei r diejenige Regel repräsentiert, auf welche die Variable `candidate` im Zeitschritt t zeigt, aus welcher durch Hinzufügen von Bedingung c die Regelverfeinerung r' entstanden ist

Episodischer Task: Sobald der Regel-Lern-Algorithmus den Zeitschritt T_{final} (siehe Def. 3.1.1) erreicht hat, ist ein sogenannter Endzustand erreicht. Die Umgebung wird anschliessend neu initialisiert, indem anhand der Wahrscheinlichkeitsverteilung $P_{init(Datasets)}$ ein neues Konzept-Lernproblem ausgewählt wird.

4.1.2 Agent

Optimalitätsmodell: Es liegt ein **episodischer** Task vor, in dem die Belohnungsfunktion r nur dann Werte grösser 0 zurückgeben kann, wenn die Umgebung in den Endzustand übergeht. Daher wird als Optimalitätsmodell die in Gleichung 4.2 definierte Funktion verwendet ($\gamma = 1$).

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_{T_{final}} \tag{4.2}$$

Ziel: Das Ziel des Agenten besteht im Lernen einer Q-Funktion,

$$Q : \mathcal{S} \times \mathcal{A}(\mathcal{S}) \rightarrow [0, 1]$$

die für eine Regel r und eine Bedingung $c \in \texttt{RefineConditions}(\texttt{r})$ abschätzt, wie korrekt die am Ende des Induziervorgangs zurückgegebene Regel ist, wenn der Agent sich in den weiteren Verfeinerungsschritten optimal verhält.

Diese Q-Funktion definiert für jede Regel r eine Politik π_r

$$\pi_r : \texttt{RefineConditions}(r) \rightarrow [0, 1],$$

die derjenigen Bedingung $c \in \texttt{RefineConditions}(r)$ eine 1-Wahrscheinlichkeit zuordnet, für welche die Q-Funktion den höchsten Wert berechnet (Gleichung 4.3).

$$\pi_r(c) = \begin{cases} 1 & \text{falls } c = \arg\max_{c' \in \texttt{RefineConditions}(r)} Q(\sigma(r), f(c')); \\ 0 & \text{sonst;} \end{cases} \tag{4.3}$$

Hierbei ist anzumerken, dass unter Umständen der maximale Q-Wert durch mehrere Bedingungen aus $\texttt{RefineConditions}(r)$ erzielt werden kann. Hierdurch wird eine Modifizierung von π_r notwendig, indem die Wahrscheinlichkeiten gleichmässig auf die entsprechenden Bedingungen, welche den maximalen Q-Wert erreichen, verteilt werden.

Kapitel 5

Beschreibung des Meta-Lernverfahrens

Dieses Kapitel beinhaltet eine Beschreibung des Lern-Algorithmus, der zur Lösung des im vorangegangenen Kapitel 4 definierten RL-Lernproblems implementiert worden ist.

5.1 Beschreibung des Lern-Algorithmus:

Zur Lösung des in Kapitel 4 festgelegten RL-Lernproblems wird im Rahmen der vorliegenden Diplomarbeit $Q(\lambda)$-Lernen [Watkins, 1989] eingesetzt. Die hierbei vom Lern-Algorithmus gelernte Q-Funktion wird durch ein künstliches neuronales Netz dargestellt, dessen Gewichte durch wiederholtes Anwenden des sogenannten *Backpropagation*-Algorithmus modifiziert werden. Eine generische Darstellung des implementierten $Q(\lambda)$-Lern-Algorithmus enthält Quelltext 5.1, welcher in den folgenden Absätzen beschrieben wird.

```
1  Initialize θ⃗ arbitrarily
2  while (stoppingCriterion()!=true)
3      s ← initState(init(Datasets))
4      With probability 1 - ε:
5          a ← arg max_b Q(s,b)
6      else
7          a ← random action ∈ A(s)
8      e⃗ = 0⃗
9      t=0
10     Repeat until t= T_final :
11         Take action a, observe reward, r, and next state, s'
12         δ ← r + γmax_b Q(s',b) - Q(s,a)
13         e⃗ ← γλe⃗ + ∇_θ⃗ Q(s,a)
14         θ⃗ ← θ⃗ + αδe⃗
15         With probability 1 - ε:
16             a ← arg max_b Q(s',b)
17         else
18             a ← random action ∈ A(s')
19             e⃗ ← 0
20         s ← s'
21         t=t+1
```

Quelltext 5.1: Generische Darstellung des Lern-Algorithmus $Q(\lambda)$, der zur Lösung des in Kapitel 4 definierten RL-Lernproblems verwendet wird.

Der in Abbildung 5.1 angeführte Algorithmus repräsentiert eine auf dem Gradientenabstiegsverfahren ([Barto/Sutton, 1998] Kap. 8.2) basierende Version des$Q(\lambda)$-Lern-Algorithmus [Watkins, 1989], in dem anstatt einer Tabelle eine nichtlineare Funktion zur Darstellung der Q-Funktion verwendet wird.

In der ersten Zeile von Quelltext 5.1 erfolgt die Initialisierung des Gewichte-Vektors $\vec{\theta}$ des neuronalen Netzes. Hierbei ist zu bemerken, dass gemäss [Mitchell 1997] (Kap. 4) die durch $\vec{\theta}$ definierte Q-Funktion eine glatte und un-

voreingenommene Oberfläche annimmt, wenn jedes Gewicht einen möglichst kleinen Zufallswert annimmt (z.B. aus dem reellwertigen Intervall $[-.05; .05]$).

Die Ausführung des in Abbildung 5.1 dargestellten $Q(\lambda)$-Algorithmus besteht im Wesentlichen aus dem wiederholten Durchlaufen der `while`-Schleife (Zeilen 3 bis 21), in welcher ein KL-Lernproblem aus der Menge der gegebenen KL-Lernprobleme entnommen wird (Zeile 3), um mit Hilfe der als Suchheuristik verwendeten Q-Funktion eine Regel zu induzieren. Die Ausführung des Meta-Lern-Algorithmus endet, wenn die in der `while`-Bedingung (Zeile 2) enthaltene Methode `stoppingCriterion` den logischen *true*-Wert berechnet. Die Methode `stoppingCriterion` wird in Unterkapitel 5.1.4 beschrieben. Vor dem Beginn des ersten Verfeinerungsschritts (Zeile 11) wird der Regel-Lerner in den Zeilen 3 bis 9 auf folgende Weise initialisiert:

Zeile 3 In Zeile 3 wird der Startzustand `s` der Agentenumgebung bestimmt. Hierzu wird aus der Menge der gegebenen Konzept-Lernprobleme $init(Datasets)$ (siehe Kap. 3.1.2) anhand der Wahrscheinlichkeitsverteilung $P_{init(Datasets)}$ (siehe Kap. 3.1.3) ein KL-Lernproblem ausgewählt, aus dessen Trainingsbeispielen eine Regel mit leerem Regelkörper gebildet wird, welcher der `s`-Variable zugewiesen wird.

Zeilen 4-7 In diesen 4 Zeilen wird die erste an den Regelkörper von `s` anzuhängende Bedingung `a` bestimmt. Dabei wird `a` entweder mit der ϵ-Wahrscheinlichkeit zufällig (Zeile 7), oder anhand der gegenwärtigen Q-Funktion (Zeile 5) ausgewählt.

Zeile 8 Die *"Eligibility Traces"* $\vec{e}$ werden mit dem 0-Vektor initialisiert, da die Trainingswerte bereits ausgeführter Episoden (KL-Lernprobleme) keinen Einfluss auf die Trainingswerte späterer Episoden haben dürfen.[1]

Zeile 9 Die Variable `t` wird initialisiert, indem ihr der 0-Wert zugewiesen wird. Diese Variable identifiziert den Verfeinerungsschritt, in welchem sich der Meta-Lern-Algorithmus im aktuellen Ausführungszeitpunkt gerade befindet. Sie wird am Ende eines jeden Verfeinerungsschritts inkrementiert (Zeile 21).

Jeder einzelne Verfeinerungsschritt t der in $1, 2, \ldots, T_{final}$ Verfeinerungsschritten stattfindenden Regelinduktion wird durch den Durchlauf der `Repeat`-Schleife ausgeführt (Zeilen 11 bis 21). Ein einzelner Verfeinerungsschritt beginnt in Zeile 11, in der folgende Aktivitäten geschehen:

(a) Ausführen der Bedingung a,

(b) Beobachten des daraufhin von der Umgebung zurückgegebenen Belohnungswert r und

(c) des Folgezustand s', in den die Umgebung s übergeht, sobald sie vom Agenten die Bedingung a erhalten hat.

[1] *"Eligibility Traces"* siehe [Barto/Sutton, 1998] Kap.7

In den Zeilen 12 bis 14 werden in Abhängigkeit der vorliegenden Q-Funktion und der bei Programmstart vorliegenden Parameterwerte (siehe Kap. 5.1.2), sowie in Abhängigkeit der von der Umgebung zurückgegebenen Informationen r, s', die Gewichte $\vec{\theta}$ der Q-Funktion durch Anwendung des Gradientenabstiegsverfahren ([Barto/Sutton, 1998], Kap 8.2) modifiziert.

In Zeile 15 wird ermittelt, ob der Suchraum im folgenden Verfeinerungsschritt erforscht wird. Dies erfolgt mit der durch den ϵ-Parameter festgelegten Wahrscheinlichkeit. Wenn der Suchraum erforscht werden soll (Zeilen 18 bis 19), dann wird eine beliebige Regelverfeinerung von Regel s' ausgewählt (Zeile 18) und den *"Eligibility Traces"* $\vec{e}$ der 0-Vektor zugewiesen. Im anderen Fall, falls die gegenwärtigen Erkenntnisse über den Suchraum zur Verbesserung der Q-Funktion verwendet werden sollen, wird anhand der aktuellen Q-Funktion diejenige Regelverfeinerung von Regel s' ausgewählt, welche den höchsten Q-Wert erzielt. Eine tiefergehende Analyse der Suchraumerforschung erfolgt in Unterkapitel 5.1.5.

Am Ende des Verfeinerungsschritts wird die Ausgangsregel s des nächsten Verfeinerungsschritts bestimmt (Zeile 20) und der t-Zähler inkrementiert (Zeile 21).

Die `Repeat`-Schleife endet, sobald die Regel mit dem höchsten Evaluierungswert im Verfeinerungspfad induziert worden ist, d.h. wenn die Gleichung $t = T_{final}$ (siehe Def. 3.1.1) erfüllt ist (Zeile 10). Falls diese Bedingung erfüllt ist, wird der Programmfluss in Zeile 2 fortgesetzt, andernfalls wird der nächste Verfeinerungsschritt in Zeile 11 ausgeführt.

5.1.1 Algebraische Spezifikation der Q-Funktion

Die Q-Funktion $Q : R^4 \rightarrow [0, 1]$ wird algebraisch durch eine vierstellige, reellwertige Funktion dargestellt, die für ein gegebenes Zustands/Aktions-Tupel $\langle s, a\rangle$ den Q-Wert berechnet. Für die Berechnung des Q-Werts $Q(s, a)$ ist es erforderlich, das $\langle s, a\rangle$-Tupel auf die vier Eingabeparameter der Q-Funktion tp,fp,$tp(a)$ und $fp(a)$ abzubilden, so dass der Wert $Q(tp(s), fp(s), tp(a), fp(a))$ dem Q-Wert für das $\langle s, a\rangle$-Tupel entspricht. Das $\langle s, a\rangle$-Tupel wird dabei auf folgende Weise auf die Eingabeparameter abgebildet:

$tp(s)$**:** Die Anzahl der von der Regel s richtigerweise als positiv klassifizierten Trainingsbeispiele (*"True Positives"*).

$fp(s)$**:** Die Anzahl der von der Regel s fälschlicherweise als positiv klassifizierten Trainingsbeispiele (*"False Positives"*).

$tp(a)$**:** Die Anzahl der von der Regelverfeinerung a richtigerweise als positiv klassifizierten Trainingsbeispiele (*"True Positives"*).

$fp(a)$**:** Die Anzahl der von der Regelverfeinerung s fälschlicherweise als positiv klassifizierten Trainingsbeispiele (*"False Positives"*).

Bemerkung: Neben der in diesem Abschnitt beschriebenen Methodensignatur sind noch weitere Signaturen denkbar [Flach/Fürnkranz, 2003a], die anstelle der gewählten vierstelligen Signatur verwendet werden können. Jedoch ist die Stelligkeit - d.h. die Anzahl der Funktionsargumente - dieser Signaturen grösser, so dass deshalb der Suchraum ebenfalls grösser ist. Aus den Experimenten, welche im folgenden Evaluierungskapitel beschrieben werden, wird hervorgehen, ob eine Signaturerweiterung sinnvoll ist.

5.1.2 Konfigurationsparameter:

Der in Abbildung 5.1 dargestellte $Q(\lambda)$-Lern-Algorithmus wird durch die Wahl folgender Parameterwerte konfiguriert:

Gewichtungsfaktor γ: Anhand des γ-Wert kann das Optimalitätsmodell (siehe Kap. 2.4) des RL-Lernproblems verändert werden, d.h. die Semantik des Optimalitätsbegriffs ist durch diesen Parameter veränderbar. Wie bereits im vorangegangen Kapitel 4 erwähnt, verweist diese Variable für die gesamten Lerndauer auf die 1-Konstante.

Lambda-Variable λ: Durch die Wahl der λ-Variablen wird ein Lern-Algorithmus aus dem kontinuierlichen Spektrum von Lern-Algorithmen bestimmt, an dessen einem Ende ($\lambda = 0$) sich das reine *"TD-One-Step"*-Lernen[2] befindet und am anderen Ende ($\lambda = 1$) das MC-Lernen[3] steht.

"TD-One-Step"-Lern-Algorithmen sind dadurch gekennzeichnet, dass ein Q-Wert nur in Abhängigkeit des Folgezustands und des unmittelbar erhaltenen Belohnungswerts aktualisiert wird, wohingegen beim MC-Lernen zur Aktualisierung der aus einer Episode (Regelinduktion) hervorgegangenen Q-Werte **nur** die während der Regelinduktion erhaltenen Belohnungswerte verwendet werden, wobei unmittelbar nach dem Erhalt eines einzelnen Belohnungswerts rückwirkende all die Q-Werte der Vorgängerzustände aktualisiert werden.

Aufschluss darüber, welcher Parameterwert aus dem reellwertigen Intervall $[0, 1]$ die besten Ergebnisse erzielt, sollen eine Serie von Experimenten geben, welche Gegenstand später folgender Kapitel sind.

Lernrate α: Um allen vorkommenden Trainingswerten den selben Anteil an der Bildung von Durchschnittswerten zu gewährleisten - ohne dabei alle Trainingswerte einzeln merken zu müssen - wird die sogenannte Lernrate α verwendet. Ihr Wert wird daher während des gesamten Trainiervorgangs stetig mit jeder Aktualisierung des Gewichtsvektors $\vec{\theta}$ des neuronalen Netzes verkleinert.

[2] TD = **T**emporal **D**ifference
[3] MC = **M**onte **C**arlo

Wahrscheinlichkeit ϵ: Der ϵ-Parameter gibt die Wahrscheinlichkeit an, mit der in einem Verfeinerungsschritt t eine Regelverfeinerung zufällig ausgewählt wird. Durch den ϵ-Wert wird daher gesteuert, wie intensiv der Suchraum erforscht wird. Je höher der ϵ-Wert, desto mehr Betonung legt der Lern-Algorithmus auf die Erforschung des Suchraums, und desto weniger Zeitanteile verbringt er mit der Verarbeitung bereits gewonnener Informationen.

5.1.3 *"Afterstate"* - Bewertungsfunktion

Eine Besonderheit des im vorangegangenen Kapitel spezifizierten RL-Lernproblem besteht in dem Umstand, dass bei vorliegenden Umgebungszustand s und der gewählten Aktion a die daraus gebildete Regelverfeinerung - und damit der Folgezustand s' - eindeutig bestimmt ist, weil s' eben nur durch Hinzufügen der Bedingung a an den Regelkörper von Regel s entstehen kann. Das heisst, dass für die Übergangswahrscheinlichkeit die Gleichung $P^a_{ss'} = 1$ erfüllt ist.

Daher ist es möglich, dass aus zwei beliebigen Zustands/Aktions-Tupeln $\langle s_1, a_1\rangle$ und $\langle s_2, a_2\rangle$ direkt ablesbar ist, ob die jeweiligen Folgezustände s'_1 und s'_2 gleich sind. In [Barto/Sutton, 1998] (Kap.6.8) wird bei Vorliegen dieser speziellen Eigenschaft angeraten, diejenigen $\langle s, a\rangle$-Tupel zusammenzufassen, welche Zustände in gleiche Folgezustände überführen, indem anstatt der Q-Funktion eine Funktion $V : \mathcal{S} \rightarrow [0, 1]$ gelernt wird. Die V-Funktion dient hierbei zur Bewertung von Folgezuständen (*"afterstate value function"*) und wird daher während der Regelinduktion als Suchheuristik zur eingesetzt. In der gegenwärtigen Implementierung des in Abbildung 5.1 wurde $Q(\lambda)$-Lern-Algorithmus wurde die Q-Funktion durch eine solche V-Funktion ersetzt (Implementierungsdetails siehe Java-Quelltext). Gemäss [Barto/Sutton, 1998] ist durch diese Massnahme eine signifikante Laufzeitreduzierung zu erwarten, da der Suchraum beträchtlich verkleinert wird.

Da aber das vorliegende RL-Lernproblem die *Markov*-Eigenschaft nicht erfüllt, ist wiederum noch offen, ob die Integration einer *"Afterstate"*- Bewertungsfunktion die Wahrscheinlichkeit reduziert, dass die Q-Funktion konvergiert. Eine Antwort auf diese Frage sollen die später folgenden Experimente geben.

5.1.4 Beenden der Algorithmus-Ausführung

Wie bereits oben erwähnt, endet die Ausführung des Meta-Lern-Algorithmus, sobald die in Quelltext 5.1 abgebildete `while`-Bedingung (Zeile 2) erfüllt ist, d.h. wenn die Methode `stoppingCriterion` den logischen *true*-Wert berechnet.

Im Idealfall sollte diese Methode den *true*-Wert berechnen, wenn der Agent die optimale Entscheidungspolitik gefunden hat. Leider ist es unmöglich vorherzusagen, ob eine solche optimale Entscheidungspolitik überhaupt existiert und diese auch tatsächlich gefunden wird, da es sich bei dem vorliegenden RL-Lernproblem um keinen MDP handelt. Die Methode `stoppingCriterion` ist

daher so realisiert worden, dass sie genau dann den `true`-Wert berechnet, wenn entweder

(a) eine fest vorgegebene Zahl von Regeln induziert worden ist, oder

(b) die anhand der Entscheidungspolitik gelernten Regeln sich über einen längeren "Zeitraum" nicht verändern.

Die Implementierungsdetails der beiden Punkte (a) und (b) werden im Evaluierungskapitel (siehe Kap. 6) erläutert.

5.1.5 Suchraum-Erforschung

Wie bereits oben erwähnt worden ist, steuert der ϵ-Parameter, wie intensiv der $Q(\lambda)$-Lern-Algorithmus den Suchraum erforscht, indem der ϵ die Wahrscheinlichkeit darstellt, mit der in einem Verfeinerungsschritt eine Regelverfeinerung zufällig als Kandidatenregel des nächstfolgenden Verfeinerungsschritts ausgewählt wird.

Logisch betrachtet wird die Erforschung des Suchraums während der Induktion einer einzelnen Regel auf folgende Weise ausgeführt:

1. Der Regel-Lerner induziert Regel r und beschreitet dabei den durch die Q-Funktion gegebenen Verfeinerungspfad $r_1, r_2, r_3, \ldots, r_n$ ($r_n = r$), wobei in jedem Verfeinerungsschritt die Gewichte $\vec{\theta}$ der Q-Funktion aktualisiert werden.

2. Nach dem Aktualisieren des Gewichtvektors $\vec{\theta}$ wird für jeden der n Verfeinerungsschritte - der Reihe nach mit Index 1 beginnend und mit Index n endend - ein Zufallsexperiment $Random(i)$ ($1 \leq i \leq n$) durchgeführt, in dem ein Zufallswert aus dem reellwertigen Intervall $[0, 1[$ berechnet wird. Sobald hierbei einen Index i^* auftritt, für das die Gleichung $Random(i^*) < \epsilon$ erfüllt ist, wird in Verfeinerungsschritt i^* eine Zufallsverfeinerung durchgeführt wird. Dies bedeutet, dass aus derjenigen Menge an Regelverfeinerungen, aus der im obigen Pfad r_{i^*} anhand der Q-Funktion ausgewählt worden ist, zusätzlich noch eine weitere Regel zufällig r_{random} entnommen wird.

3. Die Umgebung geht nun in den durch r_{random} gegebenen Startzustand über und der Programmfluss setzt im ersten Schritt mit einer neuen Regelinduktion fort.

1.Bemerkung: Die beschriebene Vorgehensweise bei der Erforschung des Suchraums weicht von $Q(\lambda)$-Lernen dadurch ab, dass die Zufallsregelverfeinerung erst nachträglich nach der Induktion des vollständigen Verfeinerungspfads, in dem die Zufallsregelverfeinerung bestimmt worden ist, den neuen Startzustand der Umgebung darstellt. Der Grund für diese Modifikation ist, dass bei Auftreten einer Zufallsverfeinerung der Evaluierungswert der Regel r verloren gehen würde, da der Regel-Lerner bei sturer Anwendung des $Q(\lambda)$-Algorithmus im folgenden Verfeinerungsschritt bei der Zufallsverfeinerung fortfährt ($s = r$).

2.Bemerkung: Die in diesem Unterkapitel 5.1.5 enthaltene Beschreibung beschreibt den Mechanismus zur Gewährleistung der Suchraum-Erforschung **nur** aus einem **logischen** Blickwinkel. So wird in der Implementierung z.B. die Zufallsverfeinerung nicht nach der Regelinduktion, sondern bereits während der Regelinduktion bestimmt (Details siehe Java-Quelltext).

5.2 Motivation für $Q(\lambda)$:

Aus folgenden Gründen stellt der $Q(\lambda)$-Lern-Algorithmus (siehe Abb. 5.1) den vielversprechendsten Ansatz zur Lösung des im vorangegangenen Kapitel beschriebenen RL-Lernproblems dar:

- Reines *"TD-One-Step"*-Lernen in der Form des Q-Lernen (ohne λ-Parameter) eignet sich nicht, weil das RL-Lernproblem nicht die *Markov*-Eigenschaft erfüllt. Gemäss [Barto/Sutton, 1998] (Kap. 7.11) eignet sich in diesem Fall MC-Lernen besser.
- Andererseits ist aus dem Lernproblem nicht ersichtlich, inwiefern die *Markov*-Eigenschaft verletzt ist. Da Q-Lernen jedoch aufgrund der *Bootstrapping*-Eigenschaft schneller korrekte Annahmen über einen eventuell in den Trainingsdaten enthaltenen *MDP* lernt, ist *"TD-One-Step"*-Verhalten wiederum erstrebenswert.

Anhand der beiden genannten Überlegungen ergibt sich die Schlussfolgerung, dass der gewünschte Lern-Algorithmus sowohl Eigenschaften eines MC- als auch eines *"TD-One-Step"*-Lern-Algorithmus besitzen sollte. Beide Eigenschaften vereinigt $Q(\lambda)$-Lernen, wobei durch den λ-Wert konfiguriert wird, welche dieser beiden Eigenschaften den stärkeren Einfluss besitzt.

5.3 Motivation für generalisierte Darstellung der Q-Funktion

Die Q-Funktion wird durch ein neuronales Netz approximiert, um einen Mangel an Trainingswerten auszugleichen. Die hierbei wichtige Frage nach der optimalen Netzstruktur ist noch unbeantwortet. Antworten sollen die im folgenden Kapitel behandelten Experimente geben.

Kapitel 6

Evaluierung

6.1 Experiment 1 - Implementierungstest

Ziel: In diesem einführenden Experiment wird der Agent mit einem **einzelnen** KL[1]-Lernproblem konfrontiert, für welches es nur eine einzige optimale Politik gibt. Der Agent erhält hierbei die Aufgabe diese Politik zu lernen. Der Sinn dieses Experiments ist lediglich in einer ersten Verhaltensbeobachtung des Agenten zu sehen, weswegen zunächst eine weniger anspruchsvolle Umgebung gewählt worden ist. Von Interesse sind hierbei die Antworten auf die beiden Fragen, ob der Meta-Lern-Algorithmus sich tatsächlich auf eine Politik festlegt (1), und wenn ja, wie optimal diese Politik ist (2).

Komponenten: Das Experiment wird aus folgenden Komponenten aufgebaut:

- die Java-Implementierung des in Kapitel 5 erläuterten Meta-Lernalgorithmus `metaLearning.MetaLearning.java`;
- das künstliche angefertigte KL-Lernproblem, das aus der Trainingsmenge `train` (siehe Tab. 6.1), der Evaluierungsmenge `test` (siehe Tab. 6.2) und dem zu erlernenden Konzept `concept` besteht, welches durch das ***attr5***-Attribut der Tabelle 6.1 gegeben ist.
- die Suchheuristik `sh`, welche gegeben ist durch die Java-Klasse `seco.heuristics.Precision`.
- die Java-Implementierung `seco.learners.GreedyTopDown.java` des in Abbildung 1.1 (Seite 11) dargestellten Regel-Lerners;
- die Java-Implementierung des in Abbildung 1.2 (Seite 16) dargestellten modifizierten Regel-Lerners `GreedyTopDownModified.java` aus dem `metalearning.evaluation`-Paket, welcher für die Bewertung der Regelkorrektheit eine andere Heuristik verwendet als für die Bestimmung des Verfeinerungspfads;

Tabelle 6.1: Trainingsmenge `train`

attr1	**attr2**	**attr3**	**attr4**	***attr5***
true	true	false	true	*1*
true	true	true	false	*1*
true	false	false	true	*0*
false	true	true	false	*0*
false	false	true	true	*0*

[1] **K**onzept-**L**ernen

Tabelle 6.2: Evaluierungsmenge `test`

attr1	**attr2**	**attr3**	**attr4**	***attr5***
true	true	false	false	*1*
true	true	false	true	*1*
true	true	true	false	*1*
true	true	true	true	*1*
false	false	false	false	*0*
false	false	false	true	*0*
false	false	true	false	*0*
false	false	true	true	*0*
false	true	false	false	*0*
false	true	false	true	*0*
false	true	true	false	*0*
false	true	true	true	*0*
true	false	false	false	*0*
true	false	false	true	*0*
true	false	true	false	*0*
true	false	true	true	*0*

Beschreibung Der Meta-Lern-Algorithmus wird nun so konfiguriert, dass der Agent die Aufgabe erhält, auf der `train`-Beispielmenge (siehe Tab. 6.1) eine Politik zu lernen, die von der Umgebung als optimal bewertet wird. Die Umgebung verwendet hierbei zur Bewertung der vom Agenten erzeugten Episoden die `sh`-Heuristik und die `test`-Beispiele (siehe Tab. 6.2) als Evaluierungsmenge.

Die beiden Beispielmengen *train* und *test* wurden so gewählt, dass der Agent bei korrekter Implementierung des in Kapitel 5 beschriebenen $Q(\lambda)$-Algorithmus das in Gleichung 6.1 definierte Zielkonzept c lernt, dessen vollständig in Tabelle 6.2 dargestellt wird.

$$c \equiv \texttt{if}\ (att1 = true)\ \texttt{and}\ (att2 = true)\ \texttt{then}\ \textit{1} \tag{6.1}$$

Beobachtungen: Bei folgender Konfiguration des Meta-Lern-Algorithmus wird für die *train*-Beispielmenge (siehe Tab. 6.1) die in Tabelle 6.3 dargestellte *"afterstate value"*-Bewertungsfunktion berechnet:

$\gamma = 1.0$ Begründung siehe Kapitel 4;

$\lambda = 0.8$ festgelegt in Anlehnung an Beispiel 7.3 [Barto/Sutton, 1998];

$\alpha = 0.3$ festgelegt in Anlehnung an [Mitchell 1997] Kap 4.7.2. Weiterhin wurde der Meta-Lerner dahingehend konfiguriert, dass der α-Wert unmittelbar nach jeder Aktualisierung des neuronalen Netzes durch Anwenden der in Gleichung 6.2 definierten Funktion verkleinert wird,

$$\alpha_{k+1} = \alpha \cdot \frac{1}{1 + \frac{k}{const}} \tag{6.2}$$

wobei in diesem Experiment die Konstante *const* mit dem Wert 10000 initialisiert worden ist, und der k-Parameter die Anzahl der Aktualisierungen des Gewichtsvektors des neuronalen Netzes darstellt.

$\epsilon = 0.1$ Begründung: Weil für die meisten in [Barto/Sutton, 1998] enthaltenen Beispiele der Wert 0.1 zugewiesen ist;

Netzstruktur das verwendete neuronale Netz besteht aus **2** Schichten, die auf folgende Weise definiert sind:

- Die erste, mittlere Schicht besteht aus 5 sogenannter *Sigmoid*-Netzknoten (siehe [Mitchell 1997] Kap 4.5);
- Die zweite Schicht wird durch einen einzelnen *lineare* Netzknoten gebildet;

Um für einen gegebenen Folgezustand $\langle p, n \rangle$ den Funktionswert des neuronalen Netzes zu berechnen, werden an die Eingänge der Netzknoten aus der mittleren Schicht **normalisierte** Werte angelegt. Das heisst, dass das Intervall $[0, 3]$ (für die negativen Beispiele in Tab. 6.1) und das Intervall $[0, 2]$ (für die positiven Beispiele in Tab. 6.1) jeweils auf das Intervall $[0, 1]$ abgebildet werden, so dass an den Eingängen der mittleren Netzknoten nur Werte aus dem $[0, 1]$-Intervall anliegen.

Ebenso wird der vom linearen Netzknoten ausgegebene Wert auf das reellwertige Intervall $[0, 1]$ normalisiert. Dieser normalisierte Wert stellt dann den Funktionswert des neuronalen Netzes für den Folgezustand s dar.

Die Struktur des neuronalen Netzes wird in Abbildung 6.1 dargestellt, wobei `result` den Funktionswert symbolisiert und die Variablen `p` und `n` jeweils die Anzahl der positiven und negativen Beispiele repräsentieren.

Haltebedingung Der Meta-Lerner beendet sein Aktivität, nachdem er 10000 Episoden induziert hat, deren jeweiliger Startzustand die alle Beispiele abdeckende $\top$-Regel ist.

Tabelle 6.3: Experiment 1: gelernte Suchheuristik nach 10000 Episoden

p/n	**0**	**1**	**2**	**3**
0	-0.44	-0.10	0.24	0.52
1	0.63	0.79	0.89	0.95
2	0.97	0.99	1.01	1.01

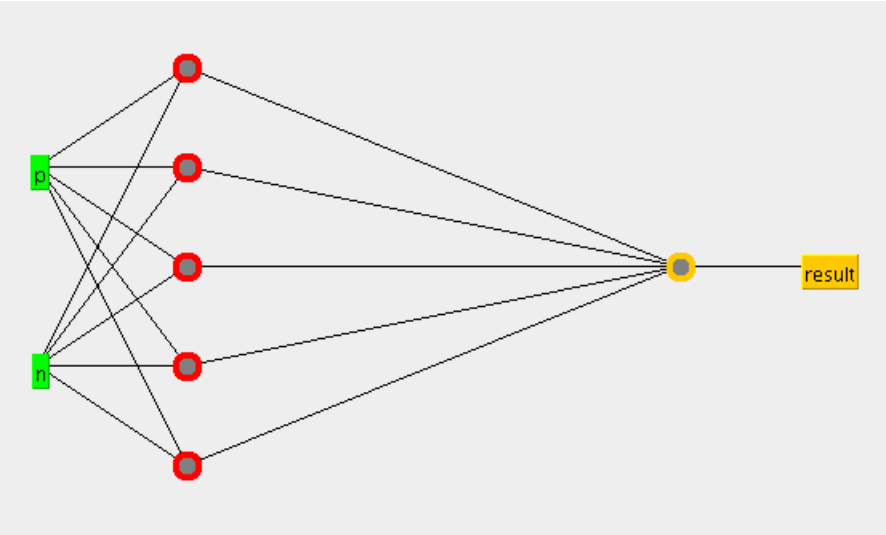

Abbildung 6.1: Experiment 1: Netzwerkstruktur

Auswertung: Wenn nun die in Tabelle 6.3 dargestellte Heuristik für die Bestimmung der Suchpfade, sowie die *sh*-Heuristik zur Bewertung der Regelkorrektheit in den modifizierten Regel-Lerner eingesetzt werden, dann wird, wie erwartet, das Zielkonzept c induziert (siehe Gleichung 6.2). Dieses Ergebnis wurde in bisher jeder Durchführung des Experiments bei unterschiedlicher Initialisierung des zum neuronalen Netz gehörenden Gewichtevektors erzielt. Hierbei ist hervorzuheben, dass der Agent in den meisten Fällen die optimale Politik nach 5 bis 10 ausgeführten Episoden gefunden hat, wobei im beobachteten *"Worst-Case"* der Agent 150 Episoden benötigt hat, bevor er sich endgültig für die optimale Politik entschieden hat.

Aus diesen Ergebnissen geht hervor, dass erstens der Agent sich auf eine Politik festlegt, und dass zweitens diese Politik sogar optimal ist - unabhängig davon mit welcher Politik er startet.

Im folgenden zweiten Experiment werden die Anforderungen an den Agenten erhöht, indem er eine optimale Politik für mehrere KL-Lernprobleme finden soll.

6.2 Experiment 2 - Überprüfen des Konvergenzverhalten und der Lernfähigkeit des Meta-Lern-Algorithmus

Ziel: In diesem Experiment wird untersucht, wie der in Kapitel 5 beschriebene Meta-Lern-Algorithmus sich unter realistischen Bedingungen verhält. Diese realistischen Bedingungen werden erzeugt, indem der Agent mehrere KL-Lernprobleme erhält, deren Evaluierungsmengen unabhängig von den jeweiligen

Trainingsmengen sind. Die Aufgabe des Agenten besteht dabei im Finden einer Entscheidungspolitik, welche für möglichst alle gegebenen KL-Lernprobleme Regeln mit möglichst optimaler Evaluierungsmengengenauigkeit induziert. Um Aussagen über die Qualität der Entscheidungspolitik zu erhalten, wird die Genauigkeit der mit Hilfe der Entscheidungspolitik induzierten Regeln mit der Genauigkeit der anhand einer gewöhnlichen Suchheuristik gelernten Regeln verglichen, wobei die Genauigkeitswerte eines jeden induzierten Regelpaares in Abhängigkeit von der zum Regelpaar gehörenden Evaluierungsmenge berechnet wird.

Aus dem Vergleich soll hervorgehen, ob der in Kapitel 5 beschriebene Meta-Lern-Algorithmus sich eignet, um zum Lernen einer Suchheuristik eingesetzt zu werden. Wenn hierbei die gelernte Entscheidungspolitik schlechter bewertete Regeln zur Folge hätte, so spräche das gegen den Meta-Lern-Algorithmus, da anhand der gelernten Entscheidungspolitik nicht einmal bessere Regeln als mit einer gewöhnlichen Suchheuristik induziert werden würden.

Im anderen Fall, falls anhand der durch den Meta-Lern-Algorithmus gelernten Entscheidungspolitik bessere Regeln induziert werden würden, so wäre es interessant zu erfahren, welche Eigenschaften diese gelernte Suchheuristik aufweist.

Weiterhin ist zu klären, ob der Meta-Lern-Algorithmus sich überhaupt auf eine Entscheidungspolitik festlegt, d.h., ob die Q-Funktion während der Ausführung des Algorithmus konvergiert. Denn aufgrund dessen, da es sich bei dem in Kapitel 4 spezifizierten Lernproblem nicht um einen *"Markov Decision Process"* handelt, kann mathematisch nicht bewiesen werden, ob der Meta-Lern-Algorithmus sich zur Laufzeit überhaupt auf eine Q-Funktion festlegt.

Komponenten: Das Experiment ist aus folgenden Komponenten aufgebaut:

- einer Java-Implementierung des in Kapitel 5 erläuterten Meta-Lernalgorithmus `metaLearning.MetaLearning.java`;
- eine Menge für das Konzept-Lernen geeigneter *UCI*-Datenbanken, vorliegend als *.arff*-Dateien (siehe Tab. 6.5);
- eine Menge von Suchheuristiken (siehe Tab. 6.4);

Tabelle 6.4: Im zweiten Experiment verwendete Heuristiken. p und n symbolisieren die Anzahl abgedeckter positiver und negativer Beispiele bei vorliegender Beispielmenge mit P positiven und N negativen Beispielen.

heuristic	**formula**
Precision	$\frac{p}{p+n}$
Laplace	$\frac{p+1}{p+n+2}$

- einer Java-Implementierung des *"Hill-Climbing/Top-Down"* Regel-Lerners `seco.learners.GreedyTopDown.java`;
- die Java-Implementierung des in Abbildung 1.2 (Seite 16) dargestellten modifizierten Regel-Lerners `GreedyTopDownModified.java` aus dem `metalearning.evaluation`-Paket, welcher für die Bewertung der Regelkorrektheit eine andere Heuristik verwendet als für die Bestimmung des Verfeinerungspfads;

Beschreibung Für das Erreichen der in diesem Experiment verfolgten Ziele werden eine Reihe von Meta-Daten gesammelt, aus denen die entsprechenden Informationen abgeleitet werden sollen. Das Programm, das diese Meta-Daten erzeugt, wird in Quelltextabbildung 6.1 (Seite 66) symbolisch dargestellt und ist wird in den folgenden Zeilen erklärt.

Die Meta-Daten werden durch Aufruf der Methode `generateMetaData` (Zeilen 1 bis 20) erzeugt. Diese Methode erhält folgende Eingabeparameter:

`databases` eine Menge von *UCI*-Datenbanken (siehe Tab. 6.5),

`heuristics` eine Menge von Suchheuristiken (siehe Tab. 6.4),

`options` ein Objekt, welches die Konfigurationsparameter des Meta-Lerners enthält (z.B. Netzstruktur, λ-Wert, Lernrate, usw) und

`seed` eine Zahl, dessen Bedeutung im weiteren Textverlauf erklärt wird.

Im ersten Schritt wird für jede Datenbank `db`, für jedes in dieser Datenbank enthaltene Konzept `concept` und für jede Heuristik `heur` $\in$ `heuristics` eine Beispielmenge `examples` erzeugt (Zeile 3), die nur aus positiven und negativen Beispielen besteht.

Im folgenden Schritt (Zeilen 4 und 5) wird die `examples`-Beispielmenge `seed`-mal durch Aufruf der `split`-Methode in zwei gleich grosse Teilmengen aufgeteilt. Die hierbei erzeugten Mengen `examples(0)` und `examples(1)` werden nach dem Zufallsprinzip gebildet, wobei zu beachten ist, dass in beiden Teilmengen die Werte des Klassenattributs mit ungefähr der selben Häufigkeit vorkommen müssen (Stratifikation). Die Aufgabe des `seed`-Parameter besteht darin, der Gefahr entgegen zu treten, welcher durch die Zufallskomponente der `split`-Methode gegeben ist, da die Wahrscheinlichkeit des Auftretens nichtrepräsentativer Mengenaufteilungen berücksichtigt werden muss. Dabei gilt: je grösser der `seed`-Parameter, desto verlässlicher sind die resultierenden Meta-Daten.

Nach Aufruf der `split`-Methode wird jeweils die Trainings- (`train`) und die Evaluierungsmenge (`test`) bestimmt (Zeilen 6 bis 8), aus denen die KL-Lernprobleme des Meta-Lern-Algorithmus gebildet werden (Zeile 9). Durch Einführung der `fold`-Variable (Zeile 6) wird hierbei erreicht, dass beide aus der `split`-Methode gewonnenen Beispielmengen sowohl einmal als Trainings-, als auch einmal als Evaluierungsmenge verwendet werden.

In Zeile 9 wird aus der Trainingsmenge `train`, der Evaluierungsmenge `test` und der Heuristik `heur` eine Menge von KL-Lernproblemen `lpSet` erzeugt, für die der Meta-Lerner eine Suchheuristik lernt (Zeile 10). Die Erzeugung der KL-Lernprobleme wird durch den Aufruf der Methode `createLearningProblems` ausgeführt, deren Definition in den Zeilen 23 bis 34 abgebildet ist.

Die KL-Lernprobleme werden durch einen Seco-regelbasierten Lern-Algorithmus erzeugt, der zur Regelinduktion einen simplen Regel-Lern-Algorithmus einsetzt (Zeile 28). Hierbei wird für jedes zu Ausführungsbeginn der `while`-Schleife vorliegende (`train`,`test`)-Mengentupel, welches die `while`-Bedingung (Zeile 26) erfüllt, ein KL-Lernproblem erzeugt (Zeile 27). Es wird hierbei in jedem Durchlauf der `while`-Schleife ein KL-Lernproblem erzeugt, bis keine positiven Beispiele in mindestens einer der beiden Menge `train` und `test` enthalten sind (Zeile 26), oder die `if`-Bedingung (Zeilen 29, 30) erfüllt ist. Falls die `if`-Bedingung nicht erfüllt ist, so werden aus den Beispielmengen `train` und `test` die von der `rule`-Regel abdeckten Beispiele entfernt (Zeilen 32, 33) und mit dem Programmfluss in Zeile 27 fortgefahren.

Die Motivation dafür, weswegen eine *Covering*-Strategie für die Generierung der KL-Lernprobleme angewendet wird, geht aus der Überlegung hervor, dass der Meta-Lern-Algorithmus nicht überfordert werden soll und der Agent daher zunächst auf wenige ähnlich strukturierte KL-Lernprobleme angesetzt wird.

Nach dem Erzeugen der KL-Lernprobleme wird der Meta-Lern-Algorithmus ausgeführt, was durch den Aufruf der Methode `metaLearn` (Zeile 10) bewirkt wird. Der Rückgabewert der `metaLearn`-Methode ist die gelernte Suchheuristik, welcher der Variable `sHeur` zugewiesen wird.

Nachdem die Suchheuristik `sHeur` gelernt worden ist, werden für jedes KL-Lernproblem `lp` aus der Menge `lpSet` die zwei Regeln `rule` (Zeile 12) und `newRule` (Zeile 13) gelernt. Die erste Regel wird dabei durch den eingangs bereits erwähnten simplen *"Hill-Climbing/Top-Down"* Regel-Lerner gelernt, der für die Regelinduktion die `heur`-Heuristik und die Trainingsmenge `lp.train` des `lp`-Lernproblems verwendet. Die zweite Regel `newRule` hingegen, wird mit dem modifizierten Regel-Lerner erzeugt, welcher den Verfeinerungspfad anhand der gelernten Heuristik `sHeur` bestimmt und die Korrektheit einer zum Verfeinerungspfad gehörenden Regel mittels der `heur`-Heuristik abschätzt.

In den Zeilen 16 bis 20 erfolgt die Ausgabe der Meta-Daten, deren Semantik in der folgenden Aufzählung beschrieben wird:

`db` Name der `db`-Datenbank;

`heur` Name der `heur`-Suchheuristik;

`concept` Das zu erlernende Konzept;

`seed` Wert der `seed`-Variable;

`fold` Wert der `fold`-Variable;

`lp.id` Identifikations-Nummer des vorliegenden KL-Lernproblems, welches dem Laufindex der $\forall$-Schleife (Zeile 11) entspricht;

`precision(rule,lp.test)` Genauigkeit der `rule`-Regel auf der zum `lp`-Lernproblem gehörenden Evaluierungsmenge `lp.test`;

`rule.length` Die Länge des Regelkörpers der Regel `rule`.

`precision(newRule,lp.test)` Genauigkeit der `newRule`-Regel auf der zum `lp`-Lernproblem gehörenden Evaluierungsmenge `lp.test`;

`newRule.length` Die Länge des Regelkörpers der Regel `newRule`.

`convergationData` Informationen anhand derer Aussagen über das bisher unbekannte Konvergenzverhalten des Meta-Lern-Algorithmus abgeleitet werden sollen. Der Meta-Lerner beendet den Lernvorgang, wenn eine der beiden folgenden Bedingungen erfüllt ist:

(a) Die anhand der Q-Funktion gelernten Regeln haben sich seit x (Konfigurationsparamter!) Episoden nicht verändert;

(b) Der Meta-Lerner hat die maximal mögliche Anzahl x_{max} (Konfigurationsparamter!) an Episoden ausgeführt;

Das `convergationData`-Objekt enthält hierbei die Anzahl der ausgeführten Episoden;

Beobachtungen: Für die im ersten Experiment verwendete Konfiguration des Meta-Lerners (siehe Kap.6.1) werden die durch den Algorithmus 6.1 erzeugten Meta-Daten in den Tabellen 6.5, 6.6, 6.7 und 6.8 zusammengefasst, wobei der `seed`-Parameter mit dem Wert 2 initialisiert worden ist.

Die Tabellen 6.5 und 6.6 beinhalten jeweils eine Zusammenfassung, der für die beiden Heuristiken *Precision* und *Laplace* ausgegebenen Meta-Daten. Beide Tabellen sind nach dem selben Muster aufgebaut, wobei jede Zeile eine Zusammenfassung der für eine *UCI*-Datenbank gewonnenen Meta-Daten repräsentiert. Die Spalten werden auf folgende Weise definiert:

data Name der *UCI*-Datenbank;

#rules Die Anzahl der für die *data*-Datenbank induzierten Regeln;

prec Die durchschnittliche Evaluierungsmengengenauigkeit (*Precision*-Wert) der mittels des gewöhnlichen Regel-Lerners gelernten Regeln, welcher für Berechnung des Verfeinerungspfads und Regelkorrektheit die selbe Heuristik verwendet;

precMod Die durchschnittliche Evaluierungsmengengenauigkeit (*Precision*-Wert) der mittels des modifizierten Regel-Lerners gelernten Regeln, welcher für Berechnung des Verfeinerungspfads die vom Meta-Lerner gelernte Entscheidungspolitik verwendet.

length Die durchschnittliche Länge der vom gewöhnlichen Regel-Lerner induzierten Regeln;

Tabelle 6.5: Tabellarische Zusammenfassung der für die *Precision*-Heuristik erzeugten Meta-Daten

data	*#rules*	*prec*	*precMod*	*length*	*lengthMod*	*w*	*t*	*l*	*#runs*
credit	273	0.34	0.23	2.51	113.52	33	89	151	8
kr-vs-kp	358	0.64	0.43	4.95	28.57	73	177	108	8
monk1	71	0.67	0.39	2.37	4.12	13	25	33	8
monk2	164	0.24	0.02	3.67	5.62	15	71	78	8
titanic	45	0.78	0.41	2.24	0.18	1	1	43	8
audiology	293	0.37	0.39	2.00	22.22	62	149	82	76
breast-cancer	211	0.40	0.00	2.31	8.61	20	77	114	8
contact-lenses	17	0.47	0.19	1.99	3.25	2	6	9	12
glass	201	0.29	0.24	1.97	61.45	55	81	65	24
glass2	120	0.38	0.09	1.78	30.77	23	49	48	8
anneal	159	0.87	0.71	1.88	9.85	11	79	69	20
horse-colic	231	0.36	0.02	2.49	62.82	16	87	128	8
hepatitis	83	0.67	0.21	1.60	21.29	6	25	52	8
heart-c	196	0.25	0.20	2.38	2.72	45	68	83	8
iris	37	0.64	0.29	1.88	25.74	5	15	17	12
monk3	63	0.80	0.25	2.42	4.38	5	38	20	8
tic-tac-toe	169	0.52	0.79	3.29	3.92	49	98	22	8
mushroom	110	1.00	0.52	1.41	10.32	0	99	11	8
labor	35	0.60	0.44	1.23	7.83	4	15	16	8
wine	62	0.51	0.29	1.27	32.66	12	24	26	12
soybean	303	0.54	0.23	2.91	15.09	31	130	142	76
vote	75	0.39	0.43	2.93	11.58	11	33	31	8
vote-1	119	0.36	0.00	3.87	14.04	21	49	49	8
lymph	94	0.48	0.18	2.03	13.22	10	41	43	16
TOTAL	3489	0.48	0.28	2.67	28.35	523	1526	1440	376

lengthMod Die durchschnittliche Länge der vom modifizierten Regel-Lerner induzierten Regeln;

w-t-l Eine *"Wins-Ties-Losses"*-Untertabelle, die anzeigt, wieviele der durch den modifizierten Regel-Lerner induzierten Regeln bessere (w-Spalte), gleich gute (t-Spalte) und schlechtere (l-Spalte) *Precision*-werte auf den entsprechenden Evaluierungsmengen erzielen als diejenigen Regeln, welche für die selben KL-Lernprobleme durch den gewöhnlichen Regel-Lerner berechnet worden sind.

#runs Die Häufigkeit, wie oft der Meta-Lerner ausgeführt worden ist.

Die letzte Zeile (*TOTAL*-Zeile) fasst den Inhalt der Tabelle zusammen und enthält die durchschnittlichen Evaluierungswerte und Regellängen, sowie die

Tabelle 6.6: Tabellarische Zusammenfassung der für die *Laplace*-Heuristik erzeugten Meta-Daten

data	*#rules*	*prec*	*precMod*	*length*	*lengthMod*	*w*	*t*	*l*	*#runs*
anneal	132	0.88	0.28	1.79	51.90	7	42	83	20
audiology	283	0.35	0.37	1.98	26.57	92	128	63	76
breast-cancer	186	0.45	0.06	2.36	7.36	37	66	83	8
contact-lenses	17	0.46	0.27	1.98	2.74	0	11	6	12
credit	219	0.37	0.12	2.59	86.11	11	93	115	8
glass2	93	0.24	0.40	1.83	12.44	14	37	42	8
glass	199	0.28	0.33	1.96	46.11	41	83	75	24
heart-c	168	0.21	0.21	2.40	79.09	25	69	74	8
hepatitis	74	0.56	0.16	1.57	20.02	6	25	43	8
horse-colic	187	0.29	0.02	2.38	54.61	12	74	101	8
iris	35	0.63	0.47	1.89	12.81	3	20	12	12
kr-vs-kp	293	0.57	0.02	4.70	30.53	52	109	132	8
labor	34	0.78	0.16	1.28	9.51	1	16	17	8
lymph	79	0.55	0.19	2.12	9.71	10	29	40	16
monk1	70	0.69	0.04	2.36	5.17	5	22	43	8
monk2	154	0.28	0.00	3.47	5.29	21	62	71	8
monk3	63	0.83	0.08	2.39	5.31	1	13	49	8
vote	70	0.42	0.22	2.96	13.73	3	29	38	8
mushroom	108	1.00	0.84	1.38	4.04	1	95	12	8
soybean	290	0.54	0.31	2.88	14.79	27	139	124	76
tic-tac-toe	176	0.66	0.65	3.32	5.05	32	92	52	8
titanic	40	0.79	0.47	2.19	0.75	0	4	36	8
vote-1	102	0.46	0.09	3.93	13.64	13	40	49	8
wine	54	0.48	0.28	1.32	25.96	9	16	29	12
TOTAL	3126	0.48	0.24	2.62	29.22	423	1314	1389	376

Aufsummierungen der jeweils in den übrigen Spalten enthaltenen numerischen Werte.

Die anderen beiden Tabellen 6.7 und 6.8 fassen jeweils für die beiden in Tabelle 6.4 angeführten Heuristiken die in den Meta-Daten enthaltenen Informationen über das Konvergenzverhalten des Meta-Lerners zusammen. Diese beiden Tabellen sind jeweils aus zwei Zeilen aufgebaut, von denen die erste Zeile die Anzahl der während einer Ausführung **eines** Meta-Lern-Prozess vom Agenten induzierten Episoden angibt, und die zweite Zeile die Summe der Meta-Lern-Prozesse darstellt, welche zu dem in der ersten Zeile in der gleichen Spalte festgelegten Episodenzahl beendet worden sind. Somit repräsentiert zum Beispiel der erste in Tabelle 6.7 enthaltene Datensatz $\langle 100, 284 \rangle$ die Information, dass 284 der ausgeführten Meta-Lern-Prozesse nach 100 induzierten Episoden konvergiert sind, d.h. die Entscheidungspolitik nicht mehr verändert haben.

Tabelle 6.7: Konvergenzverhalten für die *Precision*-Heuristik

convergation time	100	200	300	400	500	600	700	800
#runs terminated	284	45	18	16	4	6	2	1

Tabelle 6.8: Konvergenzverhalten für die *Laplace*-Heuristik

convergation time	100	200	300	400	500	600	1500	5100
#runs terminated	294	44	21	4	10	1	1	1

Auswertung: Aus den beiden Tabellen 6.5 und 6.6 ist direkt ablesbar, dass die Regeln, welche mittels der vom Meta-Lern-Algorithmus gelernten Suchheuristik induziert worden sind, schlechtere Genauigkeitswerte auf den Evaluierungsmengen erzielen, als diejenigen Regeln, die mit dem gewöhnlichen *"Hill-Climbing/Top-Down"* Regel-Lerner erzeugt worden sind. Diese Aussage stützt sich auf der Beobachtung, dass in 2829 von insgesamt 6615 KL-Lernproblemen der gewöhnliche Regel-Induzierer Regeln mit besseren Evaluierungswerten erzeugt, wohingegen die vom modifizierten Regel-Lerner gelernten Regeln nur in 946 Fällen besser sind. Weiterer Beleg für die schlechte Qualität der vom Meta-Lerner erlernten Suchheuristiken sind die eklatant hohen Unterschiede der durchschnittlichen Evaluierungswerte und der durchschnittlichen Regellängen, wie jeweils aus den Spalten 3 bis 6 der Tabellen 6.5 und 6.6 hervorgeht.

Das bei Ausführung des Experiments beobachtete Konvergenzverhalten des Meta-Lerners wird in den Tabellen 6.7 und 6.8 zusammengefasst. Aus beiden Tabellen geht jeweils hervor, dass der Meta-Lerner sich in 578 der insgesamt 752 Ausführungen bereits nach maximal 100 induzierten Episoden für eine Entscheidungspolitik entschieden hat.

Aus diesen beiden Beobachtungen,

(a) dass der Meta-Lerner konvergiert, d.h. sich für eine Entscheidungspolitik entscheidet, und

(b) die anhand der gelernten Entscheidungspolitik induzierten Regeln niedrigerer Evaluierungswerte erzielen als diejenigen Regeln, welche für die selben KL-Lernprobleme durch blosses Anwenden des gewöhnlichen Regel-Lerners erzeugt werden,

ergibt sich die logische Schlussfolgerung, dass das neuronale Netz, welches zur Approximierung der Q-Funktion verwendet wird, im Verlauf des Lernprozesses gegen eine Funktion mit **suboptimalen** Entscheidungsverhalten konvergiert.

Was sind die Konsequenzen dieser Erkenntnisse?

Zweifellos sind diese Ergebnisse enttäuschend, da der im Meta-Lerner verwirklichte Agent sich schon darin überfordert zeigt, eine Entscheidungspolitik zu finden, anhand der Regeln erzeugt werden, die zumindest genauso gut sind

wie die durch gewöhnlichen Regel-Lerner erzeugten Regeln. Bei Vorliegen dieser Sachlage ist es daher nicht sinnvoll, ein weiteres Experiment zum Lernen einer Suchheuristik durchzuführen. Das Durchführen eines solchen Experiments wäre nämlich nur dann von Interesse, wenn der Meta-Lerner eine Suchheuristik finden würde, anhand derer zumindest nicht schlechtere Regeln als mit dem gewöhnlichen Regel-Lerner induziert werden würden.

Trotzdem ist es noch zu früh, das in dieser Diplomarbeit vorgestellte Meta-Lern-Konzept als "gescheitert" zu betrachten. Denn es ist zu bemerken, dass es sich bei dem Meta-Lerner, welcher die in diesem Abschnitt tabellarisch zusammengefassten Meta-Daten erzeugt hat, lediglich um eine prototypische Implementierung handelt. Um das Konzept endgültig als "gescheitert" bezeichnen zu können, müsste daher der in Abbildung 6.1 beschriebene Prozess zur Erzeugung der Meta-Daten mehrere Male ausgeführt werden, wobei bei jeder dieser Ausführungen der Lerner andersartig konfiguriert wird (z.B. andere Netzstruktur, Lernrate, ...). Auf diese Weise liesse sich herausfinden, ob die enttäuschenden Ergebnisse nur aufgrund der Konfiguration des Lerners, oder tatsächlich auf die Struktur des Lern-Konzepts zurückzuführen ist.

Weiterhin ist zu erwähnen, dass es noch eine Reihe vielversprechender Ideen gibt, wie der in Kapitel 5 vorgestellte Lern-Algorithmus weiterentwickelt werden muss (z.B. systematischere Suchraumerforschung), um bessere Ergebnisse zu erzielen. All diese Ansätze zur Weiterentwicklung des Meta-Lern-Algorithmus werden in Kapitel 7.2 erläutert und sollten unbedingt evaluiert werden, bevor über den Erfolg des in dieser Diplomarbeit vorgestellten Lern-Konzepts endgültig entschieden wird.

```
1  procedure generateMetaData(databases,heuristics,options,seed)
2    ∀ db ∈ databases, concept ∈ db, heur ∈ heuristics
3      examples ← generateExamples(db,concept)
4      ∀ i ∈ { 0,1,2,...,seed }
5        ⟨examples(0),examples(1)⟩ ← split(examples, i)
6        ∀ fold ∈ { 0,1 }
7          train ← examples(fold)
8          test ← examples(1-fold)
9          lpSet ← createLearningProblems(train,test,heur)
10         sHeur ← metaLearn(lpSet,options)
11         ∀ lp ∈ lpSet
12           rule ← greedyTopDown(lp.train,heur)
13           newRule ← greedyTopDownModified(lp.train,
14                                           sHeur,heur)
15
16           print:
17             db, heur, concept, seed, fold, lp.id,
18             precison(rule,lp.test), rule.length,
19             precision(newRule,lp.test), newRule.length,
20             convergationData
21
22
23 procedure Set<Examples,Examples> createLearningProblems(
24                                        train,test,heuristic)
25     Set<Examples,Examples> result ← ∅
26     while positive(train) ≠ ∅ AND positive(test) ≠ ∅
27       result.add(<train,test>)
28       rule ← greedyTopDown(train,heuristic)
29       if |covered(rule, positive(train))|
30         ≤ |covered(rule, negative(train))|
31         break
32       train ← train \ covered(rule,train)
33       test ← test \ covered(rule,test)
34     return result
```

Quelltext 6.1: Erzeugen der Meta-Daten

Kapitel 7

Zusammenfassung

7.1 Zusammenfassung

Das in dieser Diplomarbeit verfolgte Ziel bestand im Lernen einer Suchheuristik für regelbasierte Lernverfahren (Meta-Lernen). Dieses Ziel wurde versucht zu erreichen, indem ein einfacher *"Reinforcement Learning"*-Algorithmus implementiert und anschliessend evaluiert wurde. Bei dem implementierten Algorithmus handelt es sich hierbei um den sogenannten $Q(\lambda)$-Lern-Algorithmus, dessen Aufgabe im Lernen einer Q-Funktion besteht, welche die gesuchte Suchheuristik darstellt. Eingesetzt in einem Regel-Lerner wird anhand dieser Funktion in jedem Verfeinerungsschritt entschieden, welche Bedingung dem Regelkörper der bisher induzierten Regel hinzugefügt wird. Wenn s eine Regel ist und a eine Bedingung darstellt, die dem Regelkörper von s hinzugefügt werden darf, so approximiert der Wert $Q(s, a)$ die Testmengen-Genauigkeit der am Ende des Induktionsprozess vom Regel-Lerner zurückgegebenen Regel, wenn - bei anschliessendem optimalen Entscheidungsverhalten - der Regel-Lerner sich entscheidet, die Bedingung a dem Regelkörper von s hinzuzufügen.

Aufgrund des Natur der vorliegenden (Lern-)Problematik wird die Implementierung dahingehend vereinfacht, dass der $Q(\lambda)$-Lern-Algorithmus anstatt der Q-Funktion eine sogenannte *"afterstate"*-Bewertungsfunktion lernt, die nur die Qualität von Folgezuständen - d.h. von Regelverfeinerungen - bewertet (siehe Kap. 5.1.3).

Um der Problematik vorzubeugen, die aus den mangelnden Trainingswerten resultiert, wird die vom $Q(\lambda)$-Lern-Algorithmus gelernte Funktion durch ein neuronales Netz dargestellt. Für die Aktualisierung der Gewichte des neuronalen Netzes wird dabei der *"Backpropagation"*-Algorithmus angewendet.

Vorgehensweise: Um das Diplomarbeitsziel zu erreichen, wurde folgende Vorgehensweise gewählt:

1. Zunächst wurde die vorliegende Lern-Problematik präzise - d.h. mathematisch - erfasst (siehe Kap. 3).

2. Nach dem Erfassen des Meta-Lernproblems erfolgte die Modellierung durch *"Reinforcement Learning"* (siehe Kap. 4).

3. Daran anschliessend wurde der $Q(\lambda)$-Lern-Algorithmus implementiert, um das im vorherigen Schritt formulierte *"Reinforcement Learning"*-Lernproblem zu lösen (siehe Kap. 5).

4. Im vierten Schritt wurde die in Kapitel 5 beschriebene Implementierung des $Q(\lambda)$-Lern-Algorithmus evaluiert. Hierzu wurden zwei Experimente durchgeführt, in denen dem Algorithmus ein oder mehrere KL[1]-Lernprobleme übergeben wurden, für die der Algorithmus optimale Entscheidungsstrategien finden sollte (siehe Kap. 6).

[1] **K**onzept **L**ernen

Schlussfolgerung: Die durch diese Diplomarbeit gewonnenen Erkenntnisse werden in der folgende Tabelle zusammengefasst:

(1) Der Meta-Lern-Algorithmus ist (noch) nicht intelligent genug, um zum Lernen einer Suchheuristik eingesetzt werden zu können.

(2) Jedoch ist es noch **viel** zu früh, um den in dieser Diplomarbeit behandelten *"Reinforcement Learning"*-Ansatz verwerfen zu können.

(3) Es ist unmöglich theoretische Aussagen über das Konvergenzverhalten des Meta-Lern-Algorithmus zu machen.

zu (1): Wie durch das zweite Experiment (siehe Kap. 6.2) empirisch nachgewiesen werden konnte, konvergiert der Meta-Lern-Algorithmus tatsächlich. Durch einen Vergleich der hierbei vom Agenten gelernten Entscheidungsstrategien mit den gewöhnlichen Heuristiken, wurde jedoch deutlich, dass die anhand der gewöhnlichen Heuristik induzierten Regeln höhere Genauigkeitswerte auf den Evaluierungsmengen besitzen. Anhand dieser Überlegung, dass der Algorithmus nur suboptimale Ergebnisse berechnet, ergibt sich die Erkenntnis, dass der Meta-Lern-Algorithmus im gegenwärtigen Entwicklungszeitpunkt (noch) nicht intelligent genug ist.

zu (2): Als Begründung für Aussage (2) ist anzuführen, dass der Meta-Lern-Algorithmus zum gegenwärtigen Entwicklungszeitpunkt lediglich Prototypen-Charakter besitzt. Das heisst, dass die optimale Konfiguration des Meta-Lerners noch ermittelt werden muss (Netz-Struktur, Suchraumerforschung, ...). Ausserdem hält *"Reinforcement Learning"* noch eine Reihe vielversprechender Methoden und Konzepte bereit (siehe Kap. 7.2), die in den Meta-Lern-Algorithmus integriert werden müssten, bevor ein abschliessendes Urteil über die Eignung des *"Reinforcement Learning"*-Ansatz zum Lernen einer Suchheuristik gefällt werden kann.

zu (3): Wegen der folgenden drei Gründe können keine allgemeingültigen Aussagen über das Konvergenzverhalten des Meta-Lern-Algorithmus gemacht werden können:

(a) Die *Markov*-Eigenschaft wird nicht erfüllt (siehe Kap. 2.4.1);

(b) Wenn zwei Regeln r_1 und r_2 den selben Zustand $s = s_1 = s_2 \in \mathcal{S}$ repräsentieren, so ist es häufig der Fall, dass die jeweiligen Mengen von Ent-

scheidungsalternativen $\mathcal{A}(s_1)$ und $\mathcal{A}(s_2)$ verschieden sein können ($\mathcal{A}(s_1) \neq \mathcal{A}(s_2)$).

(c) Die vom Meta-Lern-Algorithmus berechnete Q-Funktion wird in Form eines neuronalen Netzes dargestellt, da nicht für jedes $\langle s, a \rangle$-Tupel ausreichend genug repräsentative Trainingswerte vorliegen.

Da gültige Aussagen über das Konvergenzverhalten nur bei Vorliegen eines *"Markov Decision Process"* und gleichzeitiger tabellarischer Darstellung der Q-Funktion gemacht werden können - was aufgrund der genannten Punkte (a) bis (c) nicht gegeben ist - ist es aus mathematischer Sicht nicht möglich, das Konvergenzverhalten des Meta-Lern-Algorithmus vorherzusagen; d.h. es ist nicht möglich voherzusagen, ob und nach wie vielen Gewichtsaktualisierungen des neuronalen Netzes die Q-Funktion des Meta-Lern-Algorithmus die gesuchte Q-Funktion ausreichend genau approximiert hat.

7.2 Ansätze zur Weiterentwicklung des Meta-Lern-Algorithmus

In diesem Abschnitt 7.2 werden einige vielversprechende Ideen erläutert, wie der in Kapitel 5 definierte Meta-Lern-Algorithmus weiterentwickelt werden kann.

Diese Ideen und Konzepte sind jeweils Inhalt der beiden folgenden Unterkapiteln 7.2.1 und 7.2.2, wobei das erste Unterkapitel diejenigen Ideen enthält, welche in den Augen des Verfassers vorliegender Zeilen **unbedingt** implementiert und getestet werden müssen, bevor über den Erfolg des in dieser Diplomarbeit vorgestellten *"Reinforcement Learning"*-Ansatz zum Lernen einer Suchheuristik entschieden werden kann. Im zweiten Unterkapitel 7.2.2 werden weitere Anregungen zur Weiterentwicklung des Meta-Lern-Algorithmus genannt, die zunächst weniger wichtig hinsichtlich der Performanzverbesserung sind, jedoch dennoch erwähnenswert sind.

7.2.1 Wichtige Weiterentwicklungsansätze

In der folgenden Aufzählung werden die Ansätze zur Weiterentwicklung des Meta-Lern-Algorithmus, der Wichtigkeit nach geordnet, angeführt, wobei der wichtigste Punkt - die Strategie zur Erforschung des Suchraums - den Anfang bildet:

1. Der vielversprechendste Ansatz zur Verbesserung der im zweiten Experiment erhaltenen Evaluierungsergebnisse besteht in der Modifizierung des Agentenverhalten bei der Erforschung des Suchraums.

   In der bisherigen Implementierung wird nämlich in jedem Verfeinerungsschritt, in dem sich der Agent für die Erforschung des Suchraums entschieden hat, aus der Menge aller möglichen Regelverfeinerungen eine Regel nach dem Zufallsprinzip ausgewählt, wobei alle diese Regelverfeinerungen mit der selben Wahrscheinlichkeit ausgewählt werden können.

Dieses Verhalten des Agenten hat jedoch zur Folge, dass es ziemlich lange andauern kann, bis vielversprechende Regelverfeinerungen, welche während der Lernaktivität noch keine Veränderung der vom Agenten gelernten Politik bewirkt haben, ein weiteres mal vom Agenten "ausprobiert" werden, denn sie werden mit der selben Wahrscheinlichkeit gewichtet, wie die sogar offensichtlich schlechtesten Regelverfeinerungen.

Aus dieser Überlegung geht hervor, dass es nicht verwunderlich ist, dass die vom Agenten gelernte Q-Funktion in einem lokalen Optimum verharrt, denn es dauert viel zu lange, bis der Agent wieder die vielversprechende Regelverfeinerungen "ausprobiert", deren Auswirkungen auf den Gewichtsvektor des neuronalen Netzes aufgrund der langen Zeitdauer wieder "verwischen".

Daher besteht der erste Weiterentwicklungsansatz darin, dass der Agent in jedem Verfeinerungsschritt, in welchem er eine Regelverfeinerung zufällig auswählt, diese Regelverfeinerung nach einer Wahrscheinlichkeitsverteilung auswählt, in der vielversprechende Regelverfeinerungen mit höherer Wahrscheinlichkeit selektiert werden als schlechtere, wobei garantiert sein muss, dass jede Regelverfeinerung eine Wahrscheinlichkeit grösser 0 besitzt. Durch die Implementierung einer solchen zusätzlichen dynamisch veränderbaren Wahrscheinlichkeitsverteilung durchsucht der Agent den Suchraum systematischer, indem er mehr Informationen aus den von ihm induzierten Trainingswerten extrahiert.

2. Da der Induktionsvorgang von Regeln ein aufwendiger Prozess ist, weil die Anzahl der in einem Verfeinerungsschritt zu untersuchenden Regelverfeinerungen sehr gross ist, bietet sich der Einsatz von Plan-Methoden an (siehe [Barto/Sutton, 1998], Kap.9), um mehr Trainingswerte pro Zeiteinheit zu erzeugen. Diese Plan-Methoden bewirken, dass der Agent aus den von seiner Umgebung erhaltenen Trainingswerten ein konkretes Modell dieser Umgebung ableitet, anhand dem er zusätzliche - weniger "teure" - Trainingswerte erzeugt (Umgebungs-Simulation), welche er ebenso wie die üblichen Trainingswerte für die Aktualisierung der Gewichte des neuronalen Netzes verwendet.

3. $Q(\lambda)$- durch den $SARSA(\lambda)$-Lern-Algorithmus ersetzen.

   Wenn die Integration der beiden bisher genannten Weiterentwicklungsansätze keine ausreichenden Verbesserungen zur Folge haben, so sollte abschliessend geklärt werden, ob eine Performanzverbesserung dadurch erreicht wird, indem der $Q(\lambda)$-Lern-Algorithmus durch den sogenannten $SARSA(\lambda)$-Lern-Algorithmus ersetzt wird [Barto/Sutton, 1998] (Kap. 8.4, Abb. 8.8.) $SARSA(\lambda)$ unterscheidet sich von $Q(\lambda)$, indem er für die Erforschung des Suchraums die gegenwärtige gelernte Entscheidungspolitik verwendet.

   Begründet ist diese Idee dadurch, da in [Barto/Sutton, 1998] (siehe Kap. 8) $Q(\lambda)$ gezeigt wird, dass beide Lern-Algorithmen in einigen Fällen voneinander abweichende Resultate erzielen.

Bemerkung: Bei der Realisierung der drei genannten Ideen ist implizit zu berücksichtigen, dass jeweils eine geeignete Konfiguration des Meta-Lern-Algorithmus ermittelt werden muss - gegebenenfalls durch eine eigens hierfür entwickelte Reihe von Experimenten, welche genau dann durchgeführt werden sollten, wenn die geeigneten Parameterwerte nicht vorab durch logische Überlegungen ermittelt werden können.

7.2.2 Weitere Weiterentwicklungsansätze

Weitere Konzepte zur Weiterentwicklung des in Kapitel 5 definierten Meta-Lern-Algorithmus beinhaltet die folgende Aufzählung:

1. Suchheuristik mit komplexerer Signatur lernen.

   In der gegenwärtigen Implementierung wird ein Zustand $s \in \mathcal{S}$ lediglich durch ein Zweier-Tupel $\langle p, n \rangle$ definiert, wobei p (bzw n) die Anzahl der von einer Regel r abgedeckten positiven (bzw negativen) Beispiele darstellt. Das bedeutet, dass bei Vorliegen einer solchen Zustandsdefinition eine zweistellige Suchheuristik gelernt wird. Es gibt aber Suchheuristiken, die mehr als zwei Funktionsargumente besitzen [Flach/Fürnkranz, 2003a], so dass in dem Fall, in dem eine solche komplexere Suchheuristik gelernt werden soll, auch die Zustandsdefinition der Heuristiksignatur angepasst werden muss. Das heisst, dass für eine zu lernende n-stellige Suchheuristik eine Zustand durch die n Funktionsargumente definiert wird.

   Zu beachten ist hierbei, dass der Suchraum mit der Zunahme der Funktionsargumente überproportional wächst und dadurch die aufgrund der zu wenigen Trainingswerte hervorgerufene Problematik weiter verschlimmert wird.

2. Anstelle einer *"afterstate value"*-Funktion (siehe Kap. 5.1.3) die Q-Funktion lernen .

   Wie bereits erwähnt, eignet sich eine *"afterstate value"*-Funktion deshalb, weil ein $\langle s, a \rangle$-Tupel den Folgezustand s' eindeutig definiert und zudem weniger Trainingswerte zum Lernen benötigt werden als beim Lernen einer Q-Funktion. Jedoch stellt das vorliegende (Meta-)Lernproblem keinen *MDP* dar, so dass unterschiedliche Evaluierungsergebnisse in beiden Fällen zu denkbar sind.

3. In der bisherigen Implementierung induziert der Agent (bei Vernachlässigung der Suchraumerforschung) ausgehend von einer Startregel mit leerem Regelkörper **nur** einzige Regel r (bzw Episode). Diese Regel r stellt eine Regel aus dem Induktionspfad dar; und zwar diejenige unter den im Pfad enthaltenen Regeln, welche den höchsten Genauigkeitswert auf der zugehörigen Evaluierungsmenge besitzt. Somit werden zum gegenwärtigen Entwicklungszeitpunkt die nach r im Induktionspfad liegenden Regeln bei der Gewichte-Aktualisierung des neuronalen Netzes nicht mehr berücksichtigt.

Um nun die Anzahl der Trainingswerte zu erhöhen, könnte es sinnvoll sein, das "abgeschnittene" Ende des Induktionspfad als eigenständigen Induktionspfad zu betrachten (Rekursivität!). Zu beachten ist hierbei, dass dadurch für die kleineren $\langle s, a \rangle$-Tupel viel mehr Trainingswerte erzeugt werden würden als für die grösseren Tupel.

4. Das neuronale Netz schon vor Beginn der Agentenaktivität mit Wissen anreichern.

   Das neuronale Netz des in Kapitel 5 beschriebenen Lern-Algorithmus wird gegenwärtig mit Zufallswerten aus einem vorgegebenen reellwertigen Intervall initialisiert, damit die Funktion möglichst unvoreingenommen ist und sich möglichst schnell den von der Agentenumgebung erhaltenen Trainingswerten anpasst. Eine interessante Idee ist, den Agenten vorab mit Wissen zu bestücken, indem der Gewichtsvektor des neuronalen Netzes gezielt initialisiert wird. Eventuell könnte diese Massnahme ein effizienteres Konvergenzverhalten bewirken, weil der Agent z.B. von der Aufgabe befreit werden könnte, **alles** Wissen über den Suchraum selbständig herausfinden zu müssen. Die gezielte Vorinitialisierung des neuronalen Netzes dient somit als Starthilfe für den Agenten.

5. Wenn während der Regelinduktion in einem Verfeinerungsschritt die beste der möglichen Regelverfeinerungen anhand der aktuellen Q-Funktion ermittelt wird, so ist es möglich, dass mehrere dieser Regelverfeinerungen gleichzeitig den höchsten Q-Wert erzielen können. Der in Kapitel 5 beschriebene Lern-Algorithmus wählt in einem solchen Fall immer die zuerst auftretende Regelverfeinerung aus.

   Dieses Verhalten ist jedoch nicht wünschenswert, denn hierdurch wird der Lern-Algorithmus anfälliger für fehlerhafte Daten und für stark vom Erwartungswert abweichende Stichprobenwerte. Deswegen sollten in jedem Verfeinerungsschritt aus der Menge der Regelverfeinerungen, welche den höchsten Q-Wert erzielen, eine Regelverfeinerung nach dem Zufallsprinzip ausgewählt werden. Ob der Lern-Algorithmus durch diese Massnahme schneller oder langsamer konvergiert, lässt sich nur experimentell entscheiden. Denn einerseits könnte er schneller konvergieren, weil die Daten repräsentativer sind und das neuronale Netz somit schneller glatte Oberflächen erzeugen könnte, aber andererseits könnte es auch länger andauern, bis die Q-Funktion konvergiert ist, da durch das Mehr an Trainingswerten der Agent gleichzeitig mehr Daten auswerten muss.

Literaturverzeichnis

[Barto/Sutton, 1998] Richard S. Sutton, Andrew G. Barto: *"Reinforcement Learning: An Introduction"*, 1998.

[Fürnkranz, 1999] Johannes Fürnkranz: *"Separate-and-Conquer Rule Learning"*, Februar 1999.

[Flach/Fürnkranz, 2003a] Johannes Fürnkranz, Peter Flach: *"An analysis of rule evaluation metrics"*, 2003.

[Fürnkranz, 2003b] Johannes Fürnkranz: *"Modeling Rule Precision"*, 2003.

[Kaelbling, 1996] Leslie Pack Kaelbling, Michael L. Littman, Andrew W. Moore: *"Reinforcement Learning: A Survey"*, Journal of Artificial Intelligence 4 (1996) 237-285.

[Mitchell 1997] Tom M.Mitchell: *"Machine Learning"*, 1997.

[Quinlan, 1986] Quinlan J.R.: *"Induction of decision trees"*, Machine Learning 1(1):81-106, 1986.

[Watkins, 1989] Watkins C.: *"Learning from Delayed Rewards"*, Thesis, University of Cambridge, England, 1989.

[Witten/Frank, 2001] Ian H. Witten, Eibe Frank: *"Data Mining"*, 2001.