Knowledge Engineering in Spielen - Klassische Artikel

Kamill Panitzek kamill.panitzek@gmx.de

Immanuel Schweizer immanuel.schweizer@gmx.de

Abstract

Die klassischen Artikel beschäftigen sich mit der effizienten Implementierung von Computergestützten Spielern für das Spiel Draw Poker. Dabei werden Methoden beschrieben wie Heuristiken genutzt werden können, um selbstständig lernende Spieler zu erschaffen. Diese können durch menschliches Einwirken, durch ein anderes Programm oder aber durch Deduktion eigenständig Heuristiken erstellen. Neben einigen weiteren dynamisch lernenden Strategien werden noch statische Strategien beschrieben, sowie eine Spielstrategie, die auf einem genetischen Algorithmus beruht. Außerdem wird beschrieben, wie die unterschiedlichen Spieler in Spielen gegeneinander oder gegen einen menschlichen Kontrahenten abschneiden.

1. Einleitung

Die vier Artikel im Themengebiet "Klassische Artikel" beschäftigen sich mit der Anwendung verschiedener Lerntechniken im Zusammenhang mit dem Spiel "Draw Poker". Dabei dient der Artikel (Waterman, 1970) mit seiner Analyse der Problemstellung als Ausgangspunkt für die Betrachtungen der weiteren Artikel. In den Artikeln werden unterschiedlichste Ansätze besprochen und so ein Überblick über die Möglichkeiten in diesem Bereich der Künstlichen Intelligenz geschaffen.

Das in den Artikeln verwendete Draw Poker ist deshalb besonders interessant, da es im Gegensatz zu Spielen wie Schach oder Dame ein Spiel mit imperfekter Information ist. Spiele mit perfekter Information wie z.B. Schach sind dadurch charakterisiert, dass jeder Spieler zu jeder Zeit alle Informationen über das Spiel besitzt und sich das Problem damit auf das Durchsuchen eines endlichen Suchraumes beschränkt. Bei Poker jedoch sind viele Informationen für den Spieler nicht ersichtlich. Die Karten der Gegenspieler z.B. sind immer unbekannt. Aus diesem Grund muss der Spieler die fehlenden Informationen schätzen oder durch Analyse des Verhaltens der Gegenspieler approximieren. Dieses Ziel wird durch Lernen, adaptive Suche oder statische Heuristiken erreicht. Doch um die verschiedenen Lernmethoden der klassischen Artikel erläutern zu können, müssen zunächst kurz die Regeln des Draw Poker beschrieben werden.

Draw Poker verwendet die gleichen Handkombinationen wie alle übrigen Pokerverianten auch, daher werden diese nicht weiter dokumentiert.Jede Spielrunde des Draw Poker besteht aus den folgenden fünf Phasen:

1. *Deal.* Jeder Spieler erhält fünf Karten (eine Hand) und gibt einen Chip in den Pot. Jeder Spieler sieht nur seine eigene Hand.

2. *Betting.* Jeder Spieler hat die Möglichkeit Chips zu setzen (bet), den geforderten Einsatz mitzugehen (call), oder die Spielrunde zu verlassen (drop oder auch fold). Ein call beendet die Bettingphase.

3. *Replace.* Jeder Spieler hat nun die Möglichkeit bis zu drei Karten seiner Hand gegen eine entsprechende Anzahl an Karten vom Deck zu tauschen.

4. *Betting.* Es folgt eine weitere Bettingphase.

5. *Showdown.* Die Spieler, welche die Spielrunde nicht verlassen haben, zeigen in dieser Phase ihre Karten. Der Spieler mit der höchsten Hand gewinnt die Spielrunde und erhält die Chips des Potts, die ihm zustehen.

Sidepots oder ähnliche Zusatzregeln werden hier nicht angeführt, da sie als bekannt vorausgesetzt werden. Genauere Informationen zu Draw Poker können dem Anhang des Artikels (Waterman, 1970) oder den Texten (Jacoby, 1947) und (Yardley, 1961) entnommen werden.

Im folgenden soll nun zuerst der (Waterman, 1970) Artikel vorgestellt werden, da er einen fundierten Einstieg in die Problemstellung und verschiedene Lösungsansätze bietet und teilweise den anderen Artikeln zu Grunde liegt. Anschließend werden die Artikel (Findler, 1977) und (Findler, 1978) besprochen, da hier viele verschiedene Strategien, sowohl statischer als auch lernfähiger Natur miteinander verglichen werden. Zum Schluss soll mit dem Artikel (Smith, 1983) noch eine weitere Herangehensweise an die Problemstellung beleuchtet werden.

2. Generalization Learning Techniques for Automating the Learning of Heuristics

2.1 Problembeschreibung

Der Artikel (Waterman, 1970) stammt aus dem Jahre 1970. Zu dieser Zeit hatten viele heuristische Programme ihre Heuristiken fest im Programm implementiert. Dieses Problem spricht dieser Artikel an. Ein Programm, dass Heuristiken nutzt um Probleme zu lösen, sollte in der Lage sein, die eigenen Heuristiken zu überwachen, ihre Qualität zu messen, sie zu ändern oder sich selbst mit neuen Heuristiken zu erweitern. Dieser Prozess wird als maschinelles Lernen von Heuristiken bezeichnet.

Das Problem des maschinellen Lernens kann in zwei Teilprobleme untergliedert werden:

1. Die Repräsentation der Heuristiken, um diese dann dynamisch verändern zu können.
2. Es werden Techniken benötigt, mit denen das Programm seine Heuristiken erzeugen, evaluieren und verändern kann.

Für die Repräsentation der Heuristiken werden sogenannte *production rules* verwendet. Um sie zu verändern bzw. zu verbessern, so dass sie das Problem effizienter lösen (in Bezug auf Draw Poker gewinnen), werden diese Heuristiken trainiert. Durch die Informationen, die durch das Training gewonnen werden, führt das Programm Generalisierungen durch, die entweder neue Heuristiken erstellen oder bestehende Heuristiken verändern. Hierbei gibt es zwei Arten des Trainings: Explizites Training wird von einem Menschen oder andere Programme durchgeführt, und implizites Training bei dem das Programm die Trainingsinformationen durch logische Deduktion im normalen Problemlösungsbetrieb gewinnt.

2.2 Repräsentation von Heuristiken

Eine gute Repräsentation von Heuristiken sollte vom eigentlichen Programm separiert und mit den Generalisierungsschemata kompatibel sein. Dabei sollte jede Heuristik eindeutig identifizierbar sein. Production rules erfüllen diese Anforderungen. Sie werden dennoch untergliedert in:

- *Heuristic Rule*: Eine Heuristik, die direkt eine Aktion beschreibt, die durchgeführt werden soll.
- *Heuristic Definition*: Eine Heuristik, die einen Term beschreibt.

Programme werden in dem Artikel als eine Menge von Berechnungsblöcken beschrieben. Jeder Block verändert dabei bestimmte Variablen des Programms. Also kann jeder Block als Funktion interpretiert werden. Dabei wird die Menge der sich ändernden Variablen als *state vector* ξ bezeichnet. Für die Funktion eines Berechnungsblocks gilt also die Gleichung $\xi' = f(\xi)$, wobei ξ' der resultierende state vector ist.
Für drei Variablen A, B und C mit den Werten a_1, b_1 und c_1 ergibt sich also

$$\xi' = (a'_1, b'_1, c'_1) = f(\xi) = f(A, B, C) = (f_1(\xi), f_2(\xi), f_3(\xi)).$$

Dementsprechend haben alle Regeln die Form $(a_1, b_1, c_1) \rightarrow (f_1(\xi), f_2(\xi), f_3(\xi))$.

Da auf diese Weise jedoch sehr viele Regeln nötig wären, um jede mögliche Kombination von Variablenwerten abzudecken, werden Mengen verwendet, um ganze Regelblöcke zusammenzufassen. Dabei werden die Prädikate $A1$, $B1$ und $C1$ verwendet. Aus der oben genannten Regel ergibt sich dann die Regel

$$(A1, B1, C1) \rightarrow (f_1(\xi), f_2(\xi), f_3(\xi))$$

mit $A1 = a_1$, $B1 = b_1$ und $C1 = c_1$. Dies sind Heuristic Rules und werden auch *action rules* (ac rules) genannt.

Heuristic Definitions werden in zwei Formen unterteilt. Die *backward form* oder bf rule ist ein Term und weißt einer Variablen ein Prädikat zu falls der Wert der Variablen der Bedingung entspricht: $A1 \rightarrow A, A > 20$.
Die *forward form* oder ff rule ist ein Berechnungsterm, der Variablen des state vectors kombiniert: $X \rightarrow K1 * D$.

Um in der aktuellen Situation den besten Spielzug zu treffen, muss eine Entscheidung vom Programm getroffen werden. Dies geschieht in zwei Schritten:

Schritt 1: Jedes Element des aktuellen state vectors wird mit die rechten Seiten aller bf rules verglichen. Wenn ein Prädikat der Bedingung genügt, wird die linke Seite dieser Regel mit die rechten Seiten aller bf rules verglichen usw. bis keine Übereinstimmung mehr gefunden wird. Die resultierende Menge an Symbolen definiert den neuen symbolischen Subvektor.

Schritt 2: Der symbolische Subvektor, der in Schritt 1 abgeleitet wurde, wird mit die linken Seiten aller action rules verglichen. Dabei werden die Regeln von oben nach unten der Reihe nach geprüft. Wenn die erste Übereinstimmung gefunden wird, werden die Variablen entsprechend der rechten Seite der action rule verändert.

2.3 Programmablauf

Bei Draw Poker betrifft die kritische Entscheidung die Bettingphase. Hier muss entschieden werden, wieviel gesetzt werden soll. Hierbei sind die gegnerische Hand, sowie der gegnerische Spielstil entscheidend. Gleichzeitig muss aber ein Spielstil adaptiert werden, der für den Gegner nicht leicht analysierbar ist. Das bedeutet insbesondere, dass das Programm einen Bluff des Gegners ermitteln können sollte, und ebenso selbst in der Lage sein muss zu bluffen.

Für diese Aufgabe wurden fünf verschiedene Programmvarianten erstellt:

- P[built-in]: Hat fest einprogrammierte Heuristiken.
- P[manual]: Besitzt Heuristiken, die durch Training mit einen Menschen erlernt wurden.
- P[automatic]: Hat Heuristiken, die durch Training mit einem anderen Programm erlernt wurden.
- P[implicit]: Besitzt Heuristiken, die durch implizites Training erlernt wurden.
- P[]: Hat nur eine action rule, die eine zufällige Entscheidung fällt.

In einem *Proficiency-Test* wurden die einzelnen Programmvarianten getestet. Dabei trat jede Variante gegen einen menschlichen Spieler an. Heraus kam dabei, dass das Programm P[built-in] um 5% besser, die anderen Varianten jedoch, meist sehr knapp, schlechter als der menschliche Spieler abschnitten. Gleichzeitig wurde auch gemessen wie schnell und effektiv die lernenden Varianten ihre Heuristiken aufbauten und wie redundant die daraus resultierenden Regeln waren. Dabei stellte sich heraus, dass explizites Training besser abschnitt als Implizites.

Um implizites Training zu realisieren, wurden dem Programm grundlegend die Regeln des Draw Pokers sowie einige Grundverhaltensregeln eingegeben. Aus diesen Pokeraxiomen deduzierte das Programm anschließend seine Heuristiken für das Pokerspiel. Insgesamt konnte ein Programm mit implizit trainierter Heuristik gewonnen werden, das in etwa so gut Poker spielt, wie ein erfahrener unprofessioneller menschlicher Spieler. Dabei blufft das Programm hauptsächlich in der ersten Bettingphase einer Spielrunde, also bevor die Karten der Spieler getauscht werden.

2.4 Erzeugung neuer Heuristiken

Um zu lernen, muss das Programm in der Lage sein neue Heuristiken zu erzeugen und diese seinem Satz Heuristiken hinzufügen. Aber gleichzeitig ist es auch wichtig, dass Heuristiken ausgetauscht werden können, die zuvor zu einer schlechten Entscheidung führten. Dafür wird eine System benötigt, welches die Heuristiken bewertet. Dabei werden die Spielzüge einer kompletten Runde betrachtet und jede Entscheidung wird bewertet. Jede production rule wird entweder positiv oder negativ bewertet, um somit zu ermitteln, in wie weit diese Regel zu einer schlechten oder guten Entscheidung beigetragen hat. Schlechte Regeln können dann durch Neue ausgetauscht oder verändert werden.

Um nun neue Heuristiken zu erzeugen oder bestehende zu verändern, wird die *Trainingsinformation* benötigt. Diese besteht aus drei Teilen. Die *Akzeptanzinformation* beschreibt,

wie gut oder akzeptabel eine Entscheidung für eine gegebene Situation ist. Die *Relevanzinformation* gibt an welche Elemente (Subvektor Variablen) für diese Entscheidung relevant sind. Der Grund, aus dem diese Entscheidung ausgeführt wurde, wird durch die Evaluierung der betroffenen Elemente ausgedrückt und wird durch die *Rechtfertigungsinformation* beschrieben. All diese Informationen werden im expliziten Training durch einen Menschen oder durch ein Programm festgelegt. Beim impliziten Training werden diese Informationen durch Beobachtung des eigenen Spielverlaufs ermittelt.

Das Lernen von Heuristic Rules wird erreicht, indem aus der Trainingsinformation eine action rule extrahiert wird, die *training rule* genannt wird. Dabei bedient die Akzeptanzinformation die rechte und die Relevanz- und Rechtfertigungsinformation die linke Seite der training rule. Nun wird eine Regel, die zur schlechten Entscheidung führte modifiziert, so dass sie dem symbolischen Subvektor der training rule entspricht. Diese Methode wird als *Generalisierung* bezeichnet. Falls das nicht möglich ist, wird die training rule direkt nach dieser schwachen Regel eingefügt.

Das Lernen von Heuristic Definitions besteht daraus, dass der Wertebereich der Variablen partioniert wird. Dies geschieht entweder in Form von mutual exclusion, was bedeutet, das der Wertebereich in genau zwei sich nicht überlappende Mengen geteilt wird, oder aber in Form von überlappender Partitionierung.

Beim impliziten Training kann die Rechtfertigungsinformation aus einer sogenannten *Decision Matrix* entnommen werden. Diese wird dem Programm übergeben, wenn das Training beginnt. Jede Zeile der Matrix steht dabei für eine Spielentscheidung bzw. eine Klasse von Spielentscheidungen und jede Spalte steht für eine Subvektor Variable. Jeder Eintrag in der Matrix beschreibt dann wie wichtig eine Variable des Vektors für eine gegebene Spielsituation ist.

Die Relevanzinformation kann aus der Generierung und dem Testen von Hypothesen über die Relevanz der einzelnen Subvektor Variablen gewonnen werden. Relevant für eine Regel sind Subvektor Variablen dann, wenn in der Regel auf der linken Seite ein anderer Wert als * steht. * auf der linken Seite einer Regel besagt, dass der Wert dieser Variablen irrelevant ist für das Zutreffen dieser Regel. Auf der rechten Seite einer Regel besagt *, dass der Wert der Variablen nicht verändert wird durch diese Regel.

Des weiteren muss darauf geachtet werden, dass Redundanzen in dem Satz Heuristiken möglichst gering gehalten werden. Das soll Speicher und auch Rechenzeit sparen. Es gilt also insbesondere, neue Heuristiken zu verwerfen, falls diese bereits existieren.

2.5 Erweiterungen

In dem Artikel wird außerdem beschrieben, wie die Decision Matrix erlernt werden könnte. Um die Matrix zu erlernen wird dem Programm zu Beginn eine leere Matrix übergeben. Erst durch die Erfahrungen, die das Programm während des Spieles macht, wird die Matrix gefüllt. Wenn das Programm dann also während dem regulären Betrieb auf die Decision Matrix zugreifen will und diese an der nötigen Stelle einen leeren Eintrag besitzt, so wird eine zufällige Entscheidung getroffen.

3. Computer Poker & Studies in Machine Cognition Using the Game of Poker

Dieser Abschnitt beschäftigt sich mit der Arbeit von Nicholas V. Findler (Findler, 1977) (Findler, 1978). In diesen Artikeln werden eine ganze Reihe interessanter und sehr unterschiedlicher Strategien vorgestellt. Ingesamt 6 statische und 12 lernfähige Bots wurden im Forschungsteam an der SUNY Buffalo entwickelt und verglichen.

Das Programmumfeld für diese Bots ist dabei unterteilt in drei Teile, das "Executive Program", das für den Informationsfluss und die Ausführung zuständig ist. Dann ein "Untility Program", das alle wichtigen Informationen über das Spiel sammelt oder berechnet. Sei es die Geschichte der bisherigen Spiele, statistische Informationen oder die Berechnung verschiedener Wahrscheinlichkeiten. Und als letztes die verschiedenen Bots, die in dieses Umfeld geladen werden konnten. Dazu kommt ein graphische Schnittstelle, über die ein Mensch mit den Pokerbots interagieren kann. Dies ist aber nicht nur für die Evaluation verschiedener Bots wichtig, sondern auch für einige Algorithmen, die einen menschlichen Mentor benötigen.

Das "Utility Program" enthält dabei auch eine sogenannte Monte Carlo Verteilung, in der festgehalten wird, wie wahrscheinlich es ist, mit einer Hand zu gewinnen. Dabei wird hier unterschieden zwischen der Wahrscheinlichkeit vor dem Draw, der Wahrscheinlichkeit nach dem Draw mit Bezug auf die Hände vor dem Draw und der Wahrscheinlichkeit nach dem Draw mit Bezug auf die Hände nach dem Draw. Diese Wahrscheinlichkeit wird von vielen Bots verwendet, die Stärke der eigenen Hand abzuschätzen, manche Algorithmen verlassen sich fast ausschließlich darauf.

Wenden wir uns nun aber der Beschreibung der verschiedenen Bots zu, dabei werden wir uns den beiden erfolgreichsten statischen Bots widmen und dann, zumindest kurz, alle lernfähigen Bots ansprechen.

3.1 Statische Bots

Da die statischen Bots ihr Verhalten über die Zeit nicht ändern, sind sie im Zusammenhang mit intelligenten Bots eher uninteressant. Zwei von Ihnen zeigen aber zumindest eine einigermaßen gute Leistung und dienen in der späteren Betrachtung als Benchmark für die lernfähigen Spieler.

Der erste dieser Bots verhält sich wie ein mathematisch perfekter Spieler (MFS). Dazu berechnet er einen Einsatz der sich aus dem Erwartungswert für den Gewinn und den Verlust berechnet.

$$p_j * B_0(j,k) = (1-p_j) * [\sum_{m=1}^{k-1}(B_a(j,m) + B_f(j,k))]$$

Dabei ignoriert dieser Bot, alle Informationen des Spiels außer die Stärke der eigenen Hand. Außerdem blufft er nicht und könnte von einen intelligenten Programm oder Mensch ständig ausgeblufft werden. Trotzdem ist er in den Experimenten als Sieger aus der Menge der statischen Bots hervorgegangen.

Der zweite statische Bot der beschrieben werden soll, benutzt zusätzlich Informationen zweiter Ordnung, wie z.B. wieviele Spieler ausgestiegen sind, Erhöhungen, die Sitzanordnung etc. Dazu wird ein Faktor berechnet, der die Motivation beschreibt, zum jetzigen

Zeitpunkt im Spiel zu bleiben. Der Faktor RH wird dabei berechnet, in dem die positive Motivation (die aktuelle Größe des Potts) durch die negative Motivation (Anzahl der aktiven Spieler, Anzahl der Erhöhungen, nötiger Einsatz, und die Spieler, die danach noch eine Entscheidung treffen können) geteilt und berechnet sich damit wie folgt.

$$RH = \frac{TABLEPOT}{LIVE*(RAISECOUNT+1)*FOLLOWERS*RAISE}$$

Auch dieser Bot verhielt sich sehr stark im Vergleich zu den anderen statischen Spielern und zieht zumindest einige Sekundäreffekte in Betracht. Im den nächsten Abschnitten werden mit den lernfähigen Bots nun die interessanten Teile der Artikel beschrieben.

3.2 Adaptive Evaluator of Opponents (AEO)

Dieser Bot versucht die aktuelle Situtation seiner Gegner abzuschätzen. Dazu beginnt er mit einem Wissensstand, der es ihm erlaubt, dies sehr grob zu tun. Mit steigender Erfahrung werden die sehr einfachen Abschätzungen weiterentwickelt zu einer detaillierten Einschätzen der "Persönlichkeit" jedes einzelnen Spielers.

Mit dieser Einschätzung schätzt der Bot ein, wie Stark die aktuelle Hand des Spielers ist, in dem er sie in Kategorien einsortiert. Die grobe Einschätzung, in welchem Bereich die aktuelle Hand ist, wird dann mit Hilfe der Erfahrung in drei Dimensionen verbessert.

Dabei ist die erste Dimension eine Sammlung von statistischen Werten, die im Spielverlauf nach jedem Showdown angepasst werden. Zum Beispiel der Pott oder der letzte Einsatz etc. In der zweiten Dimension wird die Gewichtung der einzelnen Faktoren für die verschiedenen Bereiche angepasst. Also wie stark sich ein statistischer Wert auf die Auswahl dieses Bereichs auswirkt. Die dritte Dimension versucht verschiedenen Persönlichkeiten, die optimale Auswahl statistischer Werte zuzuordnen. Denn obwohl jeder auf lange Sicht sein Geld maximieren will, können die Unterziele komplett unterschiedlich sein. So dass verschiedene statistische Faktoren bessere Ergebnisse liefern.

3.3 Adaptive Aspiration Level (AAP)

Dieser Bot basiert auf der Beobachtung, dass es bei Menschen zwei verschiedene Zustände gibt, die gleichzeitig, meist aber getrennt auftreten können. Diese Zustände sind, zum einen Maximierung des Gewinns und zum anderen Schutz des Eigenkapitals. Der Übergang zwischen diesen beiden Zuständen geschieht meist als Antwort auf ein einschneidendes Ereignis. Zum Beispiel, wenn man ein Spiel mit einer sehr starken Hand verliert oder eine Glückssträhne hat und mehrere Spiele hintereinander gewinnt.

Der AAP Bot benutzt nun diese Zustände, um die Höhe der eigenen Einsätze anzupassen. Er berechnet z.B. den Einsatz des MFS oder eines AEO und modifiziert diesen nach oben oder unten je nach aktuellem Zustand.

3.4 Selling and Buying Players' Images (SBI)

Dieser Spieler setzt einen gewissen Teil seines Geldes dazu ein, den anderen Spielern ein gewisses Bild von der eigenen Spielweise zu verkaufen. Dazu nutzt er unter anderem den Bluff, um sich als Risikofreudiger oder Konservativer darzustellen.

Außerdem setzt er einen Teil seines Geldes ein, um Informationen ber das Verhalten der anderen Spieler zu erkaufen. Um das Verhalten zu kategorisieren verwendet er zwei verschiedene Maßstäbe. Einmal wird der Grad der Konsistenz und zum anderen der Grad der Zurückhaltung eines Spieler in die Berechnung mit einbezogen. Diese Informationen werden dazu benutzt, um herauszufinden, wieviel Falschinformationen gestreut werden müssen, um ein gewisses Bild zu verkaufen.

Danach wird die MFS Strategie so angepasst, dass das gewünschte Selbstbild verkauft werden kann. Im Gegenzug hilft dieses falsche Bild, in kritischen Situation, den Gegner zu falschen Entscheidungen zu veranlassen.

3.5 Bayesean Strategies (BS)

Der "Bayesean" Spieler versucht zu jedem Zeitpunkt, seine Regeln anzupassen, in dem er Vorraussagen über den Spielausgang, mit dem tatsächlichen Spielausgang vergleicht und so Wahrscheinlichkeiten anpasst. Dabei geht der Artikel auf vier leicht unterschiedliche "Bayesean" Spieler ein. Diese wurden jeweils gegen die statischen Spieler getestet und nur der besten wurde zu weiteren Tests mit den dynamischen Spielern herangezogen.

Das Prinzip hinter allen ist aber das gleiche. Die Spieler haben vorkonfigurierte Heuristiken die auf drei Arten verändert werden können. Einmal können vordefinierte Parameter verändert werden, zum zweiten können die Heuristiken neu angeordnet werden, je nachdem wie erfolgreich sie sind und zum dritten können neue Heuristiken generiert, getestet und, falls erfolgreich, hinzugefügt werden.

Dazu wird ein kontinuirlich steigender Erfahrungspool verwendet, aus dem die Algorithmen Rückschlüsse in Bezug auf ihre Heuristiken ziehen. Die verschiedenen Spieler unterscheiden sich nur in der Menge und Granularität der Informationen, die sie mitspeichern. Der erste Spieler speichert dabei nur die Stärke der eigenen Hand und die dazugehörige Aktion. Er berechnet dann bei einer Entscheidung noch die durchschnittlichen Gewinne mit dieser Hand, der nächst Schlechteren und der nächst Besseren, um dann unter den Aktionen *bet*, *call* und *fold*, die Aktion mit dem maximalen Gewinn oder minimalen Verlust auszuwählen.

Der beste der "Bayesean" Spieler (3) dagegen, speichert sich außer diesen Informationen noch in welcher Phase des Spiels, welche Entscheidungen gefallen sind. Er erhält somit ingesamt bessere Strategien und kann damit auch schlechtere Hände erfolgreich spielen.

Die beiden anderen Spieler versuchen entweder, die Informationen noch nach einzelnen Spielern aufzuteilen (2) . Dabei entsteht aber das Problem, dass es dann selbst nach 3000 Spielen noch zu wenig Erfahrungswerte gibt. Die letzte Spieler (4) speichert sich noch mehr Informationen, in dem er nach jedem Spiel eine 20 x 5 Matrix verändert.

3.6 EPAM-like Player (EP)

Die Motivation hinter diesem Bot ist die Tatsache, dass ein menschlicher Spieler nicht linear besser wird, sondern manchmal Quantensprünge in der eigenen Entwickliung erlebt. Um diese Eigenschaft abbilden zu können basiert dieser Spieler auf einem EPAM Model (Feigenbaum, 1963). Dazu speichert er sich für jeden Gegner einen Entscheidungsbaum. Dieser wird für alle Gegner gleich initialisiert und dann dem Spiel entsprechend angepasst. Je

nachdem wie sich der gegnerische Spieler verhält, wird aus dem Baum so eine Abschätzung des Verhaltens dieses Spielers.

Dazu speichert sich der Spieler noch einen Baum über alle Spieler, der die optimalen eigenen Aktionen enthält. Auch dieser wird im Spielverlauf dem Verhalten aller Gegner angepasst. So entsteht am Ende ein normativer Entscheidungsbaum, der das Verhalten einer festen Menge an Gegnern quasi-optimal beschreibt.

Da zur Verfeinerung und zum Wachstum aller Bäume eine große Menge an Informationen nötig sind, wird bei diesem Spieler eine gewisse Menge vordefinierter Spiele gespielt, bis alle Möglichkeiten ausgetestet und mit Informationen gefüllt sind.

3.7 Quasi-Optimizer (QO)

Dieser Spieler überwacht eine große Anzahl von Spielen, zwischen den verschiedenen anderen Bots und beobachtet die Komponenten, die für eine Strategie zuständig sind. Daraus erstellt er einen "super player" (SP). Das bedeutet, dass seine Strategie innerhalb aller Verfügbaren Strategien zu einem Optimum konvergieren und so eine normative Theorie innerhalb des Strategieraums schaffen würde.

Dazu müssten die Entscheidungen, Aktionen sowie alle Komponenten in Kategorien einteilbar sein, um eine Analyse durch den QO zu ermöglichen. Zum Beispiel in welchem Teil des Spielablaufs Entscheidungen getroffen wurden, welcher Art die Entscheidungen waren und welcher Form die Entscheidung war.

Dies wurde bei den anderen Bots aber nicht bedacht und es ist auch fragwürdig, ob sich alle Entscheidungen in Kategorien pressen lassen. Dieser Bot befand sich zum Zeitpunkt der Artikel auch noch in der Entwicklung und es wurde an diesem Problem gearbeitet.

3.8 Pattern Recognizing Strategy (PRS)

Diese Erweiterung macht sich die Eigenschaft des Menschen zu nutzen, nach gewissen Mustern zu handeln. Ein menschlicher Spieler hat es auf Grund seiner Natur schwer zufällig zu handeln. Seinem Handeln liegt immer ein gewisses Muster zu Grunde.

Sie verfeinert so die Abschätzung des gegnerischen Verhaltens, in dem sie die Zusammenhänge zwischen beobachtbaren und nicht-beobachtbaren Spielelementen abstrakt darzustellen versucht. Dabei sind beobachtbare Elemente z.B. Erhöhungen, Größe des Potts etc. und nicht-beobachtbare Elemente z.B. die Hand des Gegners etc.

Die Zusammenhänge zwischen Elementen werden dann in die Kategorien "landmark", "trend", "periodicity" und "randomness" unterteilt. Diese Erweiterung zielt hauptsächlich darauf ab, die Bots bei der Einschätzung menschlicher Gegner zu unterstützen.

3.9 Statistically Fair Player (SFP)

Von allen statischen Bots zeigte der MFS die beste Performanz, obwohl er durch intelligente Bots einfach auszubluffen wäre. Es liegt also Nahe diesen Bot als Grundlage für einen lernfähigen Bot heranzuziehen und ihn selbst um den Bluff zu erweitern. Dann wird im Laufe des Spiels ähnlich wie beim SBI der Grad der Konsistenz und der Grad der Zurückhaltung berechnet.

Auf Grund dieser Einschätzung seines Gegenüber wird die Häufigkeit und die Höhe der Bluffs dynamisch angepasst, da bei einem konsistenten Spieler weniger geblufft und bei einem zurückhaltendem Spielverlauf niedriger geblufft wird.

3.10 Fazit

In den beiden Artikel wird eine Vielzahl verschiedener Ansätze zur Erstellung lernfähiger Bots gegeben. Soweit sie schon fertiggestellt waren liegen ebenfalls Ergebnisse bezüglich ihres Abschneidens gegeneinandern fest. Diese Ergebnisse werden von den Autoren zur Einschätzung der Stärke der Bots und ihrer Lernfähigkeit verwendet. Diese Ergebnisse sind vorallem im Artikel (Findler, 1977) umfassend beschrieben und um Abschätzungen zu den noch nicht implementierten oder getesteten Algorithmen erweitert.

Zusätzlich wird noch eine Umgebung zur Interaktion zwischen Mensch und Bots beschrieben, die aber im Zusammenhang mit dem Praktikum keine Relevanz besitzt. Im nächsten Abschnitt wird hier nun ein weiterer lernfähiger Bot beschrieben, der auf einer komplett anderen Idee beruht und sich durch seine Unabhängigkeit von der Problemstellung auszeichnet.

4. Flexible Learning of Problem Solving Heuristics through Adaptive Search

Wir haben uns im letzten Abschnitt mit einer ganzen Reihe verschiedener Lerntechniken beschäftigt. Eine Gemeinsamkeit all dieser Techniken ist, dass die Algorithmen und die Regeln sehr stark auf das Problemfeld zugeschnitten werden müssen um zu funtkionieren.

Bei dem letzten Ansatz, den wir uns nun genauer ansehen, ist dies nur eingeschränkt nötig. Im Gegenteil wird ein Framework vorgestellt, dass nur in sehr wenigen Teilbereichen überhaupt auf spezifisches Wissen zurückgreifen muss.

Die Idee, die hinter diesem Ansatz steckt, ist es, Lernen im Allgemeinen als Suche im Problemlösungsraum zu betrachten. Zur Suche wird hier ein Genetischer Algorithmus verwendet. Genetische Algortihmen sind mächtige Suchheuristiken, die nach dem Vorbild der Evolution arbeiten. Ihre Anwendunggebiete sind vorallem NP-schwere Optimierungsprobleme, wie z.B. das Dilemma des Handlungsreisenden.

Nun werden sie hier im Kontext von Künstlicher Intelligenz eingesetzt, um Heuristiken zur Problemlösung zu optimieren. Der Algortihmus besitzt dabei eine vorgegebene Menge an Lösungsheuristiken und geht dann in zwei, sich wiederholenden, Phasen vor. In Phase 1 werden die Heuristiken nach Performanz bewertet und ausgesucht und in Phase 2 wird die Menge durch genetische Operationen auf den Heuristiken erweitert. So wird sichergestellt, dass Teilräume durchsucht werden, die eine gute Performanz versprechen und sich diese Teile durch den kompletten Vorgang propangieren.

Jede Heuristik, oder besser Regel, besteht dabei aus einer Sequenz von Komponenten und die gentischen Operatoren arbeiten auf diesen Komponenten. Die Variante des Algorithmus, die in dem Artikel vorgestellt wird, arbeitet dabei auf verschiedenen Granularitätsebenen. Als oberste Ebene dient eine Sequenz von kompletten Aktionen und als unterstes Level einzelne Symbole.

Die genetischen Operationen, die dabei von den Autoren verwendet werden, sollen nun kurz vorgestellt werden.

Zuerst gibt es die *crossover* Operation. Dieser Operator nimmt zwei Regeln, auf der selben Ebene, und wählt jeweils einen Breakpoint aus. Die Komponenten rechts von diesem Breakpunkt werden dann ausgetauscht. Nimmt man also die Regeln c1c2‖c3c4 und c5c6‖c7c8, dann erhält man c1c2c7c8 und c5c6c3c4.

Als zweiten Operator gibt es die *inversion*. Diese Operation arbeitet auf einer Regel und definiert dort zwei Breakpoints. Die Komponenten zwischen diesen Breakpoints werden dann vertauscht. Als Beispiel wird so aus c1‖c2c3‖c4, c1c3c2c4. Dies funktioniert nur, wenn die Komponenten unabhängig voneinander sind, also nur auf der obersten Ebene.

Der letzte Operator ist die *mutation*. Sie tritt mit sehr geringer Wahrscheinlichkeit auf und verändert einzelne Komponenten zufällig. Sie stellt damit sicher, das alle Punkte des Suchraums erreicht werden können.

Das Programm selbst ist relativ einfach aufgebaut. Es gibt eine Menge von Regeln, die "knowledge base", zur Lösung des Problems. Diese Menge wird von der "problem solving component" auf k Probleme angewendet und dann von der "critic" analysiert. Diese bewertet die Menge und gibt sie danach an den genetische Algorithmus. Hier werden die besten ausgewählt und durch die oben besprochenen Operationen verändert. Ausserdem gibt die "critic" die aktuelle Problemlösung heraus. Dies ist in jedem Durchgang die Regel mit der jeweils besten Bewertung. Dabei ist die "problem solving component" problemunabhängig gestaltet. In dieses Framwork muss dann eine Menge an problemspezifischen Variablen und Operationen eingespielt werden. Auf dieser Menge arbeitet dann jede der Regeln und Aktionen und schafft so die Vorraussetzung, problemspezifische Lösungen zu finden.

Der wichtigste Teil ist aber eigentlich die "critic", denn sie misst die Performanz, der einzelnen Regeln und sorgt so für die Auswahl durch den genetische Algorithmus. Dazu vergleicht sie die Ausgabe des "problem solvers" mit problemspezifischen Benchmarks und Bestimmungen, aber vergleicht auch allgemeine Kriterien, wie z.B. die Komplexität oder Effizienz der Regeln, um eine feinere Abstimmung zu erreichen.

Die "critic" ist damit der Teil, der nicht Problemunabhängig gestaltet werden konnte. Um ihr System zu testen, griff die Gruppe nun auf das Pokersystem von (Waterman, 1970) zurück und übernahm auch die Pokeraxiome in die "critic" um die Performanz der Programme messen zu können. Ausserdem wurde die "problem solving component" mit 7 Zustandsvariablen gefüttert, auf denen seine Entscheidungen beruhen können. Diese sind: *Wert der eigenen Hand*, *Geldbetrag im Pott*, *Betrag des letzten Einsatzes*, *Verhältnis des Geldbetrags im Pott zum letzten Einsatz*, *Anzahl der Karten die der Gegner ausgetauscht hat*, *Wahrscheinlichkeit das der Gegner erfolgreich geblufft werden kann*, *Mass für die Zurückhaltung des Gegners*.

Dazu gibt es vier Grundoperationen, namentlich: *call*, *drop*, *bet low*, *bet high*.

Ausser der Übereinstimmung der Entscheidungen mit den Pokeraxiomen, wurde auch die Anzahl der mit Erfolg gespielten Runden in die Performanz mit eingerechnet.

Dieser Bot trat dann in einem Pokerspiel gegen den im Artikel von (Waterman, 1970) vorgestellten P[built-in] an, der in etwa die Stärke eines durchschnittlichen Pokerspielers hat. Bei den anschliessenden Test gegen P[built-in], sowie eine verbesserte Version von P[built-in] zeigte sich einmal, dass der Bot mit der Anzahl der gespielten Spiele immer besser wurde, als auch, dass er P[built-in] überlegen war oder gleiche Performanz zeigte, obwohl viel wenigerproblemspezifisches Wissen nötig war.

Abschliessend lässt sich damit sagen, dass der in diesem Artikel vorgestellte Ansatz, sehr vielversprechend ist. Die problemunabhängige Ausrichtung des Ansatzes und die trotzdem sehr guten Ergebnisse in einem definierten Problemumfeld zeigen hier die Stärken gut auf. Dabei ist zu beachten, dass die Ergebnisse nur so gut sind, wie die "critic". Denn nur wenn die richtigen Anforderungen an die Performanz gestellt werden, wird der Algortihmus dementsprechend konvergieren. So verschiebt sich das Problem aber von der Erstellung problemspezifischer Algorithmen und Lerntechniken, hin zu der Erstellung problemspezifischer Benchmarks und wird somit ungleich weniger komplex.

5. Fazit

In den letzten Abschnitten wurde deutlich, dass sich die Problemstellungen in der Künstlichen Intelligenz wenig verändert haben. Auch wenn die Spielvariante heute Texas Holding heißt, haben sich die Probleme nicht verändert.

Wie trifft man Entscheidungen in einem Umfeld imperfekter Information und Ungewissheit? Eine der spannenden Fragen in vielen verschiedenen Anwendungsgebieten.

Die hier vorgestellten Strategien sollten dabei als Grundlage für weiterführende Strategien verstanden werden und sind auch oft noch Bestandteil heutiger Strategien. Denn keine Strategie deckt für sich gesehen genug Möglichkeiten ab, um tatsächlich intelligent zu spielen. Aber sie zeigen die Richtung, in der verschiedene Ansätze der Künstlichen Intelligenz sich entwickeln können und haben.

References

Feigenbaum, E. A. (1963). The simulation of verbal learning behavior. *In Computers and Thought.*

Findler, N. V. (1977). Studies in machine cognition using the game of poker. *Communications of the ACM*, *20*, 230–245.

Findler, N. V. (1978). Computer poker. *Scientific American*, 112–119.

Jacoby, O. (1947). *Oswald Jacoby and Poker.* Double Day, Garden City N.Y.

Smith, S. F. (1983). Flexible learning of problem solving heuristics through adaptive search. *In Proceedings of the 8th International Joint Conference on Artificial Intelligence (IJCAI-83)*, 421–425.

Waterman, D. A. (1970). Generalization learning techniques for automating the learning of heuristics. *Artificial Intelligence*, *1*, 121–170.

Yardley, H. O. (1961). *The Education of a Poker Player.* Pocket Books, New York.