Numerische Lösung einer Klasse dynamischer Zwei-Personen-Nullsummen-Spiele durch Diskretisierung und Synthese mit einem Neurosimulator

André Meyer

Diplomarbeit

Universität Hannover
Studiengang Mathematik mit Studienrichtung Informatik
Mai 2005

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die vorliegende Arbeit beschäftigt sich mit einer Klasse von Optimierungsproblemen, für die es keine einfache allgemeingültige Lösung gibt: die Klasse der dynamischen Zwei-Personen-Nullsummen-Spiele, anschaulich auch Verfolgungsspiele genannt. Praktische Anwendungen sind zum Beispiel Verfolgungs- und Ausweichprobleme wie die autonome Steuerung von Fahrzeugen und Lenkwaffen. Auch Probleme, die ansonsten der Simulation vorbehalten sind, zum Beispiel optimale Kriegsstrategien, wirtschaftliche Probleme wie die Aufteilung von Marketingbudgets und Marktpräsenz bei Konkurrenz mit anderen Wettbewerbern lassen sich ebenso modellieren wie politischer Wahlkampf oder Brettspiele wie Dame oder Mühle.

Diese vielfältigen Problemstellungen machen einen allgemeinen und schnellen analytischen Lösungsweg unmöglich. Die approximative Diskretisierung dieser Probleme läßt jedoch eine Vorgehensweise zu, die auf eine große Teilmenge dieser Probleme angewendet werden kann. In dieser Arbeit wird gezeigt, daß man diese Probleme durch einen endlichen Spielbaum approximieren kann, der dann mit Hilfe eines Programms nach optimalen Strategien durchsucht wird. Zudem ermöglicht dieser Ansatz die Repräsentation der Lösung in einer Form, die, obwohl sie sehr speichereffizient ist, auch sehr schnell automatisch ausgewertet werden kann: die Interpolation der diskreten Lösung durch ein neuronales Netz. Das Verfahren soll an mehreren konkreten Problemstellungen getestet und bewertet werden.

Nachdem in diesem Kapitel kurz neuronale Netze und ihre Bedeutung für diese Arbeit erläutert werden, werden im zweiten Kapitel Differentialspiele behandelt. Nach einem kurzen Verweis auf die Entstehung der Theorie der Differentialspiele wird das der Klasse der Differentialspiele zugrundeliegende Prinzip erklärt. Dieses Prinzip wird dann in ein approximierendes diskretes Modell überführt, mit Verweisen auf spezielle diskrete Differentialspiele aus der Fachliteratur. Im dritten Kapitel wird ein Algorithmus entwickelt, um das im zweiten Kapitel entstandene diskrete Modell zu lösen. Dieser Algorithmus wird auch in Programmform umgesetzt. Die Entwicklung erfolgt nach Methoden des Software-Engineerings und wird in dem Kapitel ausführlich dokumentiert. Das vierte Kapitel beschäftigt sich dann mit einem konkreten Differentialspiel, seinem diskreten Modell und dem Resultat nach Berechnung durch das im dritten Kapitel entwickelte Programm sowie der Interpolation durch den Neurosimulator FAUN. Das Ergebnis wird auf Abweichungen von der theoretischen Lösung untersucht und diskutiert. Um die Leistungsfähigkeit des entwickelten

Programms zu demonstrieren, wird im fünften Kapitel ein weiteres Problem höherer Dimension gelöst. Im sechsten Kapitel wird ein Fazit gezogen und ein Ausblick auf Verbesserungsmöglichkeiten des Algorithmus gegeben.

1.2 Neuronale Netze

Der Begriff neuronaler Netze findet sich seit Ende der Fünfziger Jahre in der Informatik, ausgehend von einem vereinfachenden Modell einer Nervenzelle, dem künstlichen Neuron, sowie der Vernetzung mehrerer künstlicher Neuronen zum Perzeptron, das Frank Rosenblatt 1957 entwickelte (vergleiche [Ros58]). Ein Perzeptron ist eine Menge von auf spezielle Weise vernetzten künstlichen Neuronen (genauer: ein Feed forward net, siehe Abbildung 1), weshalb im Folgenden nur noch allgemein von neuronalen Netzen und Neuronen gesprochen wird.

Frank Rosenblatt entwickelte das Modell der künstlichen Nervenzelle nach dem Vorbild einer natürlichen Nervenzelle, um sich automatische Prozesse des Gehirns wie Selbstorganisation und selbstständiges Lernen nutzbar zu machen. Besonders in den Gebieten der Mustererkennung und künstlicher Intelligenz wird von neuronalen Netzen reger Gebrauch gemacht, um verschiedene Muster automatisch voneinander zu unterscheiden. Interessanterweise wird auch in der gerade momentan sehr populären Hirnforschung das neuronale Netz und seine Fähigkeit, sich verändernden Datenbeständen anzupassen, wiederum als Modell für Prozesse im Gehirn herangezogen (vergleiche zum Beispiel [Spi00]). Die Bedeutung von neuronalen Netzen für diese Arbeit ist jedoch eine etwas andere, da eine viel grundlegendere Eigenschaft dieser Netze ausgenutzt wird: die Erstellung einer Funktion, welche zwischen mehreren gegebenen Punkten einer Stichprobe interpoliert.

Ein neuronales Netz stellt man sich zuallererst als einen Graphen vor (siehe Abbildung 1).

Jeder Knoten in diesem Graphen hat nun mehrere reellwertige Eingänge $(x_1, ..., x_N)$ und nur einen, ebenfalls reellwertigen, Ausgang O. Jeder Knoten ist also eine Funktion der Form $f : \mathbb{R}^N \mapsto \mathbb{R}$.

Jede dieser Funktionen hat nun Parameter, mit deren Hilfe sie variiert werden kann. Siehe dazu auch Abbildung 2.

Diese Parameter w_i werden Gewichte genannt, da sie die Eingangsparameter x_i verstärken oder abschwächen. Die Summe der gewichteten Eingänge wiederum bildet den Eingangsparameter einer einfachen Funktion f, die idealerweise drei Eigenschaften hat: sie ist beschränkt (üblicherweise in $[0, 1]$ oder $[-1, 1]$), monoton steigend und sie hat einen "Schaltbereich", in dem sie annähernd linear steigt. Eine solche Funktion ist zum Beispiel der in dieser

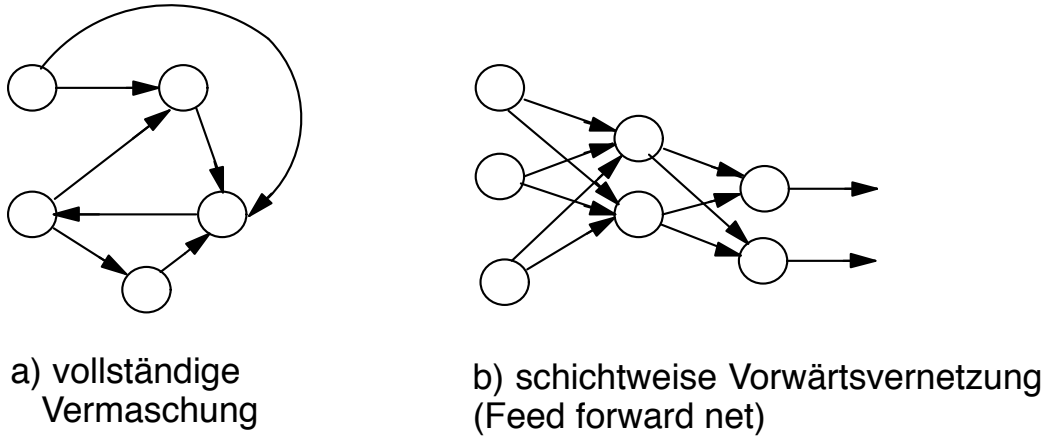

Abbildung 1: Mögliche Topologien neuronaler Netze [Lie03]

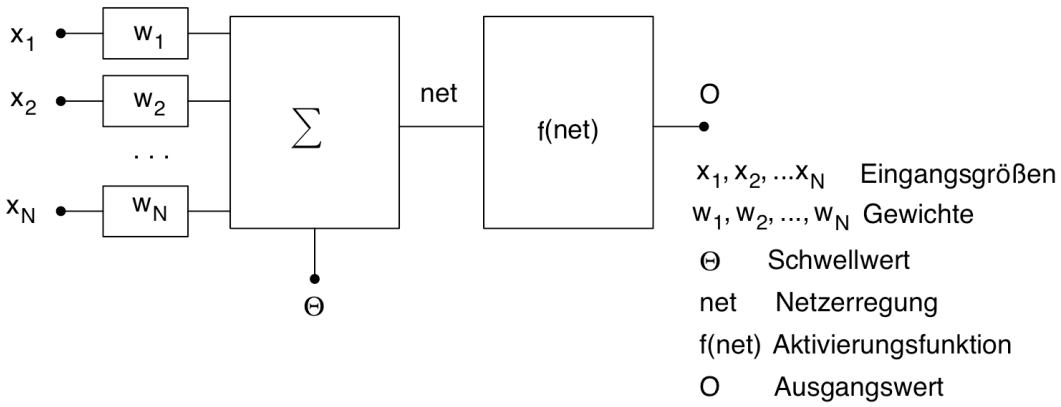

Abbildung 2: Aufbau eines Neurons [Lie03]

Arbeit durch den Neurosimulator FAUN verwendete Tangens-Hyperbolicus. Die $tanh()$- Funktion hat alle drei Eigenschaften: $lim_{x\to-\infty}tanh(x) = -1$ und $lim_{x\to\infty}tanh(x) = 1$, sie ist streng monoton steigend und zudem hat sie in ihrem Wendepunkt im Nullpunkt eine Steigung von 1.

Die Summe der gewichteten Eingänge und des Schwellwertes ist also das Argument der gewählten Aktivierungsfunktion $f(net)$: $O = f(\Theta + \sum_{i=1}^{N} w_i x_i)$. Der in Abbildung 2 genannte Schwellwert Θ wird in der Praxis als weiterer Eingang x_0 mit einem festen Wert gehandhabt. Im Folgenden wird dieser Wert der Einfachheit halber als 1 angenommen, manche Neurosimulatoren (zum Beispiel FAUN) verwenden aber auch $0,5$. Dieser zusätzliche Eingang wird auch 'Bias-Neuron' genannt, da die Schwelle durch ihn in eine Richtung versch-

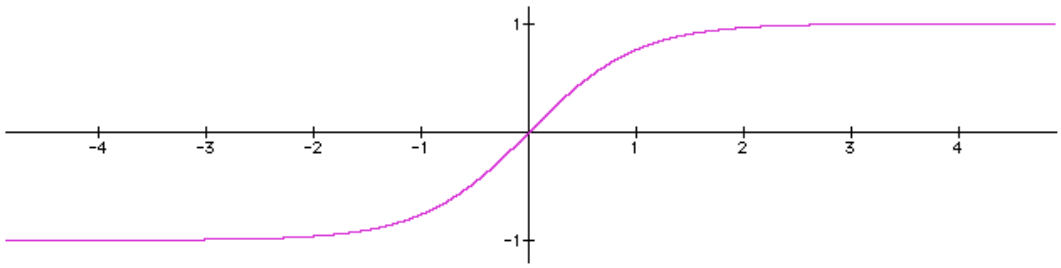

Abbildung 3: Tangens-Hyperbolicus

ben wird, das Neuron also schon ein 'Vor-Urteil' in eine bestimmte Richtung fällt. Die Schwelle wird dann ebenfalls über ein Gewicht w_0 eingestellt, so daß $\Theta = x_0 w_0$. So läßt sich die Funktion weiter vereinheitlichen: $O = f(\sum_{i=0}^{N} w_i x_i)$.

Die eigentliche Aufgabe eines Neurosimulators ist nun, die w_i jedes Knotens so einzustellen, daß sie durch bestimmte vorgegebene Punkte läuft. Diese Punkte werden Muster genannt. Angenommen, es sind M verschiedene Muster gegeben, die mit $1 \leq \mu \leq M$ indiziert werden. Diese Muster bestehen jeweils aus den oben genannten N Eingangswerten, durch $1 \leq i \leq N$ indiziert $x_{\mu i}$ und ihren gewünschten Ausgangswerten t_μ.

Zunächst werden alle Gewichte mit *zufälligen* Werten initialisiert. Jedes Neuron besitzt nun eine vom Gewichtsvektor $\mathfrak{w} = (w_0, w_1, ..., w_N)$ abhängige Fehlerfunktion $E(\mathfrak{w})$:

$$E(\mathfrak{w}) = \frac{1}{2}\sum_{\mu}(t_\mu - O_\mu)^2 = \frac{1}{2}\sum_{\mu}[t_\mu - f(\sum_{i}(w_i x_{\mu i}))]^2$$

Diese wird dann, zum Beispiel durch Anwendung des Gradientenabstiegsverfahrens, auf ein lokales Minimum optimiert. Beim Gradientenabstieg ist zwar bekannt, in welche Richtung und wie steil die Funktion fällt, aber nicht, in welcher Entfernung sich das Minimum befindet. Da die Steigung des Gradienten allerdings direkt am Minimum gegen Null geht, wählt man die Schrittweite $\Delta w_{\mu k}$ meist antiproportional zur Steigung (bei steigendem Gradienten liegt das Minimum links), und zwar mit dem Faktor $\epsilon > 0$. Dabei gilt bei sequenzieller Verarbeitung aller Muster für jedes Muster die Fehlerfunktion:

$$E_\mu(\mathfrak{w}) = \frac{1}{2}(t_\mu - O_\mu)^2 = \frac{1}{2}[t_\mu - f(\sum_{i}(w_i x_{\mu i}))]^2$$

Mit $net_\mu = \sum_i (w_i x_{\mu i})$ ist deren partielle Ableitung nach dem Gewicht w_i dann:

$$\Delta w_{\mu i} = -\epsilon \frac{\partial E_\mu}{\partial w_i} = \epsilon (t_\mu - O_\mu) f'(net_\mu) x_{\mu i}$$

Im gesamten Netzwerk kennt man nun aber nur die geforderten t_μ von Neuronen, deren Ausgabewert direkt Teil des Ergebnisses ist. Bei mehrlagigen Netzen werden deshalb erst einmal die Ausgabeneuronen eingestellt. Danach werden diese als gegeben hingenommen, und so der Fehler der vorhergehenden Schicht ermittelt, mit dem diese Schicht dann optimiert wird. So wird weiter verfahren, bis schließich die erste Schicht eingestellt worden ist.

Auf diese Art und Weise wird jeder Knoten des Netzes in vielen Iterationen an jedes Trainingsmuster angepaßt. Das Problem, das das Gradientenabstiegsverfahren immer mit sich bringt, tritt natürlich auch hier auf: die Vorgehensweise findet nicht notwendigerweise das *globale* Minimum, sondern kann auch in einem *lokalen* Minimum mit inakzeptabel großem Fehler zur Ruhe kommen. Daher verläßt man sich nicht auf das erstbeste Ergebnis, das ja aus einem mit bestimmten Zufallszahlen als Gewichten initialisiertem Netz entstanden ist, sondern trainiert so viele mit verschiedenen Zufallsreihen initialisierte Netze, bis ein zufriedenstellendes Netz gefunden wurde, dessen Fehlersumme für alle vorliegenden Muster annehmbar klein ist.

Moderne Verfahren zum Training neuronaler Netze setzen höher entwickelte Optimierungsverfahren als den Gradientenabstieg ein. Der verwendete Neurosimulator FAUN stellt verschiedene Verfahren zur Auswahl, darunter NPSOL, NLSSOL und NLSCON. Die hier diskutierten Probleme treffen auf diese Verfahren aber in gleicher Weise zu.

Natürlich gibt es noch eine Vielzahl weiterer Dinge zu beachten. So sollten Muster bereitgestellt werden, die zwar nicht zum oben vorgestellten Training verwendet werden, anhand derer jedoch das Netz noch einmal geprüft wird. Diese werden Validierungsmuster genannt. Mit ihrer Hilfe lassen sich hochspezialisierte Netze vermeiden, die ein zufälliges Rauschen zu exakt annähern. Ist der Validierungsfehler sehr viel größer als der Trainingsfehler, so ist dies durch ein starkes Nachbilden des Rauschens zu erklären. Die Validierungsdaten sollten nämlich durch die umliegenden Trainingsmuster angenähert werden, der Rauschanteil der Validierungsdaten ist hingegen stark lokal unterschiedlich, findet sich also auch nicht in den Trainingsmustern wieder.

Bei mehrlagigen Netzen stellt sich die Frage, mit wievielen Mustern nacheinander eine Schicht trainiert wird, bevor die vorhergehende Schicht eingestellt wird, und ob dieselben Muster oder andere für jede Schicht genutzt werden.

Ebenfalls wichtig ist, wie immer beim Gradientenabstiegsverfahren, die

Wahl des richtigen ϵ zur richtigen Zeit. Es existieren eine ganze Reihe von verschiedenen Theorien, mit welchem Faktor die Trainingszeit bzw. die Nummer des Trainingsdurchgangs einbezogen werden soll. So kann man ϵ konstant lassen oder mit fortlaufendem Training immer weiter verringern. Das zur Zeit wohl komplexeste Verfahren in dieser Richtung ist die simulierte Abkühlung, das sogenannte 'Simulated Annealing', das sich momentan bei der Optimierung von bestimmten neuronalen Netzen großer Beliebtheit erfreut (vergleiche zum Beispiel [AdFD00]). Bei diesem Verfahren wird die Wahrscheinlichkeit, daß ϵ einen bestimmten Wert überschreitet, mit der Zeit immer geringer.

Muster mit Messfehlern, ganz besonders große Ausreißer in den Trainingsdaten, bringen das Netz dazu, sich an diesen Ausreißern auszurichten, da die Abweichungen der einzelnen Trainingsmuster wie oben gesehen quadratisch in die Fehlerfunktion einfließen. Somit weist eine einzige starke Abweichung einen größeren Gesamtfehler auf als mehrere nicht so stark ausgeprägte Abweichungen.

Auch die Wahl der Verteilung der Trainings- und Validierungsstichproben kann über Erfolg und Mißerfolg entscheiden. Selbst die Reihenfolge, in der die Trainingsmuster herangezogen werden, wirkt sich stark auf das Ergebnis aus. Es gibt weiterhin die Möglichkeit, verschiedene Netze als sogenannte 'Expertenrunde' zusammenzufassen. Diese Netze sind dann parallel geschaltet, und geben alle sozusagen eine Stimme ab. Das endgültige Ergebnis wird dann aus der Mehrheit (bei Kategorisierung) oder dem Mittelwert bzw. dem Median (bei Funktionsinterpolation) gebildet. Man kann mehrere Netze auch seriell hintereinanderschalten, so daß erst eine grobe Kategorisierung durch ein Netz erfolgt, dessen Ergebnis wiederum Eingabe des nächsten Netzes ist. Im Endeffekt wird zwar so wieder nur ein großes neuronales Netz aus mehreren Einzelnetzen gebildet, aber die Trainingsweise ergibt eventuell ein besser angepaßtes Netz. Es existiert allerdings kein Automatismus, um solche Zwischenschritte zu erstellen. Diese Vorgehensweise bietet sich also nur in Fällen an, in denen der 'gesunde Menschenverstand' schon grobe Kategorien erkennt, mit denen die Netze zur Erstkategorisierung trainiert werden können.

6 Fazit und Ausblick

6.1 Erkenntnisse

Der in dieser Arbeit vorgestellte Algorithmus läßt sich zur annähernden Lösung von Differentialspielen verwenden, deren Auszahlungsfunktion den im dritten Kapitel gestellten Anforderungen entspricht. Aussagen über die Abweichungen von der kontinuierlichen Lösung lassen sich durch genaue Betrachtung von Rundungsfehler und Schrittweite der Züge von beiden Spielern treffen. Sowohl bei den Rundungsfehlern als auch den Schrittweiten spielen jedoch Betrag und Richtung abhängig von der am jeweiligen Spielstand zur Anwendung kommenden Strategie eine Rolle, was die Abschätzung des Approximationsfehlers erschwert. Außerdem ist natürlich der zeitliche Rundungsfehler zu beachten, da die Auszahlung auf ganze Züge gerundet wird. Dieser stört jedoch nicht bei einer Ermittlung der optimalen Strategie, die mit derselben oder einer größeren Zugdauer berechnet wird. Der Rundungsfehler ist durch den Einsatz von genügend Speicher- und Rechenkapazität dadurch zu minimieren, daß die Auflösung des Spielfeldes erhöht bzw. der Maßstab des Modells vergrößert wird. Alternativ läßt sich dadurch natürlich auch der Fehler der zeitlichen Diskretisierung verkleinern, indem die zeitliche Schrittweite heruntergesetzt wird. Das Training der neuronalen Netze funktioniert mit den durch das Programm extrahierten Daten zufriedenstellend.

In der Arbeit ist klar geworden, daß die optimalen Strategien der beiden Spieler einen sehr empfindlichen Gleichgewichtspunkt bilden. Bei einem Abrücken eines Spielers von der optimalen Strategie eröffnen sich dem anderen Spieler schnell Möglichkeiten, seine Auszahlung zu verbessern, wenn er genaue Kenntnis von der gegnerischen Strategie besitzt. Im Beispiel des Homicidal Chauffeur Games mit netzgesteuertem Verfolger reicht dem Verfolgten theoretisch schon die Erkenntnis, daß sein Gegner durch ein trainiertes neuronales Netz gesteuert wird, um ganz allgemeine Schwächen wie die Stetigkeit eines solchen Netzes auszunutzen.

Daher ist beim Einsatz neuronaler Netze in solchen Spielen darauf zu achten, daß sich der Gegenspieler nicht auf die Abweichungen des Netzes von der optimalen Strategie einstellen kann. Für den Fall, daß der Gegner sich selbst mit “optimaler Strategie” bewegt, funktionieren allerdings schon Netze mit geringem Trainingsaufwand sehr gut.

6.2 Verbesserungen am Programm

Das im Rahmen dieser Arbeit entwickelte Programm ist gut dazu geeignet, diskrete Differentialspiele zu lösen. Es ermöglicht eine eindrückliche Visualisierung der berechneten Auszahlungsfunktionen und der Strategien der einzelnen Spieler. Allerdings lassen sich noch einige Erweiterungen und Verbesserungen des Programms denken. Im Folgenden sollen einige dieser Ideen, die bei der Entwicklung des Programms und der Auswertung seiner Ergebnisse aufgekommen sind, vorgestellt und diskutiert werden.

6.2.1 Graphische Oberfläche

Eine graphische Oberfläche zu erzeugen, die die vielen Anwendungsmöglichkeiten des Programms ausnutzen kann, ist nur schwerlich möglich. Spätestens bei der Erstellung eigener Spieler und Spielendezonen kann eine graphische Benutzeroberfläche den Kompiler nicht ersetzen. Es wäre jedoch möglich, für einzelne Anwendungsfälle getrennte Oberflächen zu erstellen, die entweder die entsprechenden Batchdateien generieren oder die Funktionen in `Solver` direkt ansprechen. So würde zum Beispiel die Möglichkeit, kritische Bereiche für den Export der Trainings- und Validierungsdaten für FAUN manuell mit dem Mauszeiger zu markieren diesen Vorgang stark vereinfachen. Reizvoll wäre in diesem Zusammenhang sicherlich auch eine Vorschau auf die noch in der Berechnung befindlichen Auszahlungsmatrizen.

6.2.2 Parallelisierung

Das vorliegende Programm rechnet mit einem einzigen Thread. Das eine Parallelisierung des Programms von Nutzen wäre, zeigt der Zeitaufwand für die beiden in dieser Arbeit berechneten moderaten Probleme. Dabei ist der Algorithmus prinzipiell parallelisierbar, schließlich muß immer dieselbe Aufgabe für jedes Feld im Stack ausgeführt werden. Allerdings müssen alle Threads sowohl auf die vier verschiedenen Stacks als auch auf die beiden Auszahlungsmatrizen zugreifen, was bei der Parallelisierung zu berücksichtigen ist. Durch eine geschickte Aufteilung der Verwaltung der Auszahlungsmatrizen zwischen den verschiedenen Rechnern können dann auch Probleme gerechnet werden, die sich dem Speicher eines Rechners entziehen. Durch die immensen Speicheranforderungen des Algorithmus könnte es bei der Parallelisierung sogar dazu kommen, daß durch Caching-Synergien bei einigen Problemen eine Steigerung der Geschwindigkeit um einen Faktor erreicht wird, der größer ist als die Anzahl der Prozessoren. Bei dem Beispiel des Modified Cornered Rat Game verbrachte der Rechner die meiste Zeit mit dem Zugriff auf den virtuellen

Speicher, und der Prozessor wurde durchschnittlich nur zu einem Achtel ausgenutzt. In solchen Fällen würde die Parallelisierung mit getrennten Speichern ganz erhebliche Geschwindigkeitsvorteile bringen.

6.2.3 Weitere Ausgabefunktionen

Momentan ist keine Funktion vorgesehen, um den Spielverlauf von einem gegebenen Punkt an zu visualisieren. Dazu sollten die Punkte, die ausgehend von einem gegebenen Startpunkt nach jeder Runde erreicht werden, mit Linien verbunden werden. Hier könnte man auch Zwischenschritte berücksichtigen, die beispielsweise nach einem halben Zug von E, gefolgt von einem halben Zug von P erreicht würden. An einer solchen Graphik lassen sich Auswirkungen von verschiedenen Strategien sehr gut verdeutlichen.

Da die Funktion des Vergleiches zweier Auszahlungsmatrizen in dieser Arbeit häufig benötigt wurde, wäre eine Funktion im Programm wünschenswert, die das automatisch erledigt. Darüber hinaus könnte diese Funktion noch statistische Auswertungen vornehmen, so wie den mittleren und maximalen Fehler, sowohl prozentual wie auch absolut, berechnen. Diese Funktion könnte explizit aufgerufen oder automatisch bei der Simulation mit einer vorgegebener Strategie ausgeführt werden.

Ein Training eines Netzes mit den Werten der Auszahlungsfunktion ist im vorliegenden Programm nicht zu realisieren. Bei einem verwendeten Wertebereich von -32760 bis 32760, von dem in den vorgestellten Problemlösungen jedoch weniger als ein halbes Prozent im Außenbereich verwendet wird, läßt sich ein noch in diesem Bereich fein abgestuftes Netz nur schwer trainieren. Bei Spielen nach Art, bei denen es nur auf Gewinn oder Verlust ankommt, wäre das ohne weiteres möglich, allerdings läßt sich daraus dann keine Strategie mehr bestimmen. Das Training mit dem Gradienten der Auszahlungsfunktion, unter anderem verwendet zur automatischen Steuerung eines in die Atmosphäre eintauchenden Space-Shuttles in [Bre00], gestaltet sich durch die stufige, weil auf ganze Züge gerundete Auszahlung schwierig. Dazu wird weiter unten im Abschnitt “Zeitlicher Rundungsfehler” noch mehr gesagt.

Um einen effektiven Trainingsdatensatz für FAUN schnell erzeugen zu können, wäre die automatische Erkennung kritischer Bereiche eine Hilfe. Die Herausforderung liegt hier darin, Unstetigkeitsstellen der Strategiematrix von Rauschen zu unterscheiden.

6.2.4 Rundungsfehler der Statusvariablen

Es wurde gezeigt, daß sich der Rundungsfehler bei bestimmten Problemstellungen ab einer bestimmten Zuganzahl stark auf die berechnete Auszahlungsmatrix auswirkt. Ein Algorithmus, der ohne eine Diskretisierung des Spielstandes auskommt, würde diesen Rundungsfehler nicht aufweisen. Dazu müsste ein Weg gefunden werden, die entstehenden Flächen gleicher Auszahlung anders zu beschreiben. Im vorliegenden Algorithmus werden sie als Menge von Punkten auf einem Gitter vermerkt, es wäre aber auch möglich, ihre Hülle als n-dimensionalen Polyaeder zu beschreiben. Eventuell wären bei komplexen oder unzusammenhängenden Flächen auch mehrere Polyaeder für einen Auszahlungswert notwendig. Die Ziel- bzw. Ursprungsmenge der Fläche könnte durch die Ermittlung der Ziel- bzw. Ursprungspunkte ihrer Eckpunkte erfolgen.

Weiterhin notwendig wäre jedoch die Diskretisierung der Kontrollvariablen, da die Endlichkeit der verschiedenen Strategien für die Baumsuche unerlässlich ist. Die entstehenden Polyaeder wären also weiterhin eine Näherung, allerdings würde die aus ihnen ermittelte optimale Strategie weniger sprunghaft verrauscht sein. Um eine weitere Verfeinerung zu erlangen, könnte zwischen den einzelnen Punkten des Polyaeders auch nicht linear, sondern quadratisch oder komplexer interpoliert werden. Eine solche Interpolation wäre auch zwischen den einzelnen Pixeln im vorliegenden Algorithmus denkbar. Diese Interpolation würde allerdings nur das Rauschen der ermittelten optimalen Strategie vermindern, aber keine exaktere Abschätzung der Auszahlungsfunktion ermöglichen.

6.2.5 Fehler der zeitlichen Diskretisierung

Eine nach obiger Vorgehensweise erwirkte Verringerung des Rundungsfehlers bewirkt auch, daß eine feinere zeitliche Auflösung ermöglicht wird. Dadurch läßt sich wiederum die Abweichung vom kontinuierlichen Fall verringern.

Außerdem ließe sich eine Näherung der Auszahlungsfunktion aus den beiden berechneten Auszahlungsmatrizen ermitteln. Eine Abschätzung durch einfache Mittelwertbildung beider Auszahlungsmatrizen ist zwar nicht möglich, es müßte nach einer geometrischen Analyse des Verlaufs der Niveauflächen in beiden Matrizen jedoch möglich sein, diejenigen Flächen zu bilden, deren Rand immer in der Mitte zwischen den Rändern der beiden korrespondierenden berechneten Niveauflächen verläuft. Inwiefern sich die genaue Auszahlungsmatrix besser dazu eignet, die optimale Strategie eines Spielers zu ermitteln, für die ja die Differenz der Auszahlung für einen bestimmten Zug mit einer bestimmten Dauer ausgewertet wird, ist fraglich.

6.2.6 Zeitlicher Rundungsfehler

Aus den errechneten Auszahlungsmatrizen ließe sich mit denselben Methoden wie bei der Ermittlung der besten Strategie auch ein genauerer Auszahlungswert auf Bruchteile eines Zuges genau berechnen. Da sich aus einer so erzeugten zeitlich feiner aufgelösten Matrix aber keine bessere Strategie bestimmen ließe, ist dieser Schritt in der momentanen Ausführung des Programms überflüssig. Eventuell wäre jedoch die Auszahlungsfunktion oder ihr Gradient von Interesse. Dann würde diese Funktion unentbehrlich werden, da sich eine Bestimmung des Gradienten durch den stufigen Verlauf der Auszahlungsmatrix nicht durchführen läßt.

6.2.7 Erweitertes Anwendungsgebiet

Das Programm hat die Einschränkung, daß der Betrag der Auszahlungsfunktion mit jedem Zug um einen festen, immer gleichen Betrag sinken muss. Es wäre jedoch auch möglich, Spiele zu berechnen, bei denen der Betrag der Auszahlungsfunktion mit jedem benötigten Zug um einen Wert sinkt, der beliebig und vom jeweils gewählten Zug abhängig sein darf. Beispielsweise könnte im Modified Cornered Rat Game nicht die Dauer, sondern der Energieverbrauch der teilnehmenden Spieler im jeweiligen Zug die Auszahlungsfunktion reduzieren. Dann würde eine Bewegung mit geringerer als der maximalen Geschwindigkeit Sinn machen, um den Energieverbrauch zu reduzieren. Außerdem könnte in diesem Beispiel auch eine optimale Strategie für die Katze berechnet werden, wenn die Maus entkommt, wenn die Maus eine niederigere Auszahlung bei geringerem Abstand zur Katze erhält.

Das Problem, das sich hierbei stellt, ist die Notwendigkeit, daß das Maximum der Nachfolger für jeden Knoten bekannt ist. Nur in diesem Fall kann ein Knoten bereits berechnet werden, obwohl seine Nachfolger nicht alle bereits berechnet sind.

Das vorliegende Programm geht von allen Endpunkten des Spiels rückwärts durch den Spielbaum. Alle Knoten des Baumes, die auf einer Höhe liegen, werden auch in dem entsprechenden Durchgang gesetzt. Nur weil ein gemeinsames bekanntes Maximum für die gesamte Höhe vorliegt, können überhaupt erst die Knoten, auf denen der gewinnende Spieler am Zug ist, in der SET-Phase berechnet werden.

Eine andere Möglichkeit der Berechnung wäre es, jedes Feld erst einmal auf den besten gefundenen Wert zu setzen, allerdings im späteren Verlauf des Spieles auftretende bessere Werte des Knotens wieder zu berücksichtigen. Wenn ein besserer Wert zu dem Knoten im späteren Verlauf der Berechnung auftritt, be-

deutet das einen früheren Zeitpunkt des Spieles. Sämtliche Vorgängerknoten dieses Knotens müssen dann ebenfalls neu berechnet werden. Durch diese Art der Berechnung steigt jedoch entweder der Speicherverbrauch oder die Anzahl der möglichen Züge sinkt, da in den Bytes für den Wert jedes Feldes der Matrix zusätzlich die Einträge für die Stacks mit abgelegt werden müssen, und nicht wie bisher entweder der Wert oder der Stackstatus. Bei den verwendeten zwei Byte für jedes Feld abzüglich zwei Bits für den Stackstatus und unter Berücksichtigung des Vorzeichenbits und des Rückgabewertes für den Zugriff auf Felder außerhalb des Spielfeldes wären somit noch 8190 Züge möglich. Da in diesem Fall allerdings auch die Beschränkung der Auszahlung auf ganze Zahlen wegfällt, könnte gleich mit Gleitkommazahlen gerechnet werden, die ohnehin besser dafür geeignet sind, Abstände und andere berechnete Modifikationen der Auszahlung wiederzugeben.

Die Notwendigkeit, daß die Auszahlung mit jedem Zug sinkt, ist dadurch allerdings nicht aufgehoben. Diese Beschränkung muß weiterhin bestehen bleiben, um Bewegungen im Kreis zu verhindern, die eine steigende Auszahlungsfunktion auf dieser Strecke ausnutzen.

Leider lässt sich mit diesem Verfahren nach dem Abschluß der Berechnung nicht mehr mit Sicherheit sagen, welche der errechneten Werte wirklich die endgültigen Werte für das Spiel mit optimalen Strategien sind. Bei dem in dieser Arbeit verwendeten Verfahren werden Felder nur dann gesetzt, wenn der Wert bei optimaler Strategie bekannt ist. Bei der hier vorgestellten Alternative ist das nicht der Fall. Nach Abschluß der Berechnung stellt sich das Problem, alle diejenigen Felder zu finden, deren sämtliche Nachfolgefelder tatsächlich mit optimaler Strategie berechnet sind. Nur dann sind die Felder ebenfalls Teil der Lösung des Spiels mit optimaler Strategie.