Genetische Programmierung und Schach

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Universität Dortmund
am Fachbereich Informatik
von

Wolfgang Kantschik

Dortmund

2006

Genetische Programmierung und Schach

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Universität Dortmund
am Fachbereich Informatik
von

Wolfgang Kantschik
September 2006

Dortmund

2006

Tag der mündlichen Prüfung:

Dekan / Dekanin:	Prof. Dr. Peter Buchholz
Gutachter:	Prof. Dr. Wolfgang Banzhaf
	Prof. Dr. Heinrich Müller

Bewertungsskala: ausgezeichnet, sehr gut, gut, genügend

Inhaltsverzeichnis

Danksagung

Diese Arbeit wäre ohne die Hilfe vieler Menschen nicht möglich gewesen. Mein Dank gebührt zuerst Prof. Dr. Wolfgang Banzhaf für seine vertrauensvolle Unterstützung bei der Erstellung dieser Arbeit.

Als sehr wichtig hat sich die freundliche und kreative Arbeitsatmosphäre am Lehrstuhl für Systemanalyse des Fachbereichs Informatik an der Universität Dortmund erwiesen. Diese war besonders wichtig und inspirierend für diese Arbeit. So möchte ich allen Mitarbeitern dieses Lehrstuhls für die Unterstützung während meiner Zeit dort danken. Besonders hervorheben möchte ich dabei Dr. Christoph Richter, Dr. Jens Niehaus, Dr. Andre Leier und Dr. Markus Brameier. Ohne ihre Hilfe in Form von Diskussionen, programmiertechnischen Tipps, der Bereitstellung von Daten und des Korrekturlesens hätte diese Arbeit nicht die vorliegende Form erreicht. Weiterhin muss ich auch Roderich Groß, Keno Albrecht und Martin Villwock für ihre Hilfe und Tipps danken.

Ein besonderer Dank gebührt Heike Rest, ohne deren Hilfe diese Arbeit nicht so fehlerfrei wäre. Ein weiterer Dank gebührt Christian von Strachwitz und Ursula Neumann, ohne ihre Aufmunterung hätte die Arbeit länger gedauert.

Mein Dank gilt ferner Prof. Dr. Heinrich Müller, Prof. Dr. Günter Rudolph und Dr. Stefan Droste, für Ihre Bereitschaft, sich mit dieser Arbeit auseinanderzusetzen.

Einleitung

„Der nächste Weltmeister im Schachspiel könnte ein Computer sein, der übernächste einer, der von Computern programmiert wurde.“ [111] Mit diesen Worten gab Strouhal im Jahre 1996 eine eindeutige Richtung für die weitere Entwicklung von intelligenten Programmen vor. Der erste Teil des Zitats wurde schon im gleichen Jahr Geschichte. Die Schaffung eines Computerprogramms, das den Schachweltmeister schlagen konnte, war eines der großen Ziele der Künstlichen Intelligenz. Diese Entwicklung nahm in der Nachkriegszeit ihren Anfang mit der Nutzung der Computer für das abstrakte Schlussfolgern. Während der 60er Jahre wurden Computer hergestellt, die logische und geometrische Theoreme beweisen, Rechenprobleme lösen und gute Schachspiele ausführen konnten. Vor einigen Jahren stellte die Carnegie Mellon University zwei tischgroße Computer her. Dieses System, *Deep Blue* genannt, konnte im Jahr 1997 Kasparow in einem Match mit 3.5:2.5 besiegen.

Strouhal erwartet als zweiten Aspekt seiner Vorhersage die Entwicklung eines Weltmeisterprogramms durch einen Computer. Zum damaligen Zeitpunkt bestand jedoch überhaupt keine Klarheit darüber, wie ein Computer ein Schachprogramm entwickeln könnte. Diese Arbeit stellt mit der Nutzung von Evolutionären Verfahren zur Entwicklung von Schachprogrammen eine Lösung dieser Problematik vor.

Trotz der großen Leistungen der KI-Programme können Computer viele Problemaspekte nicht so effizient behandeln wie wir Menschen, so dass sie viele Aufgaben nicht oder nur sehr schlecht lösen können [28, 29]. So werden beispielsweise den KI-Programmen die Züge mit einer Tastatur eingegeben, denn kein computergesteuerter Roboter kann die Figuren auf einem Schachbrett so wie ein durchschnittlicher Mensch sehen und ergreifen. Warum ist es aber leichter das abstrakte Denken als Wahrnehmen und Handeln zu imitieren? Über hunderte Millionen von Jahren überlebten unsere Vorfahren, indem sie besser sahen und sich besser bewegten als ihre Konkurrenz. Die Evolution steigerte die Effizienz unserer Vorfahren bei den überlebenswichtigen Aufgaben. Auf der anderen Seite ist rationales Denken, wie man es für das Schachspiel braucht, eine vergleichsweise junge, recht spät erlernte Fähigkeit, die es vielleicht nicht länger als 100.000 Jahre gibt. Die beim Menschen für das rationale Denken genutzten Hirnbereiche haben noch lange nicht die ausgefeilte Organisation erreicht, wie dies bei den Bereichen für Wahrnehmen und Handeln der Fall ist. Aus diesem Grund ist die Denkfähigkeit des Menschen auch im Vergleich zu anderen Fähigkeiten eher schlecht. Das Schachspiel ist ein sehr gutes Beispiel um den Unterschied zwischen Menschen und Programmen erfahrbar zu machen. Prinzipiell gibt es beim Schach 10 hoch 120 verschiedene Möglichkeiten. *Deep Blue* überprüft immer noch Millionen von Brettstellungen, ehe es einen Zug bestimmt, während die besten Spieler allenfalls einige hundert Konstellationen betrachten. Könnten Menschen den Suchraum aber genauso effizient aufspannen wie Programme, um wievieles besser wären dann die Menschen beim Schachspiel? Diese Frage können wir hier nicht beantworten, da wir wohl eine Million Jahre Evolution abwarten müssten.

Die Evolution hat sich als eine ausgezeichnete Optimiermethode gezeigt, deshalb werden die Mechanismen der natürlichen Evolution auf die Entwicklung von Programmen angewendet um ein Schachprogramm durch einen Computer entwickeln zu lassen. Hierzu werden bekannte Verfahren wie die Genetische Programmmierung und Evolutions Strategien verwendet. Weiter werden auch neue Strukturen für die Genetische Programmierung entwickelt um einen flexibleren Aufbau der Programme zu ermöglichen.

Schach (vom pers.: Schah = König; stehende Metapher: "Das Königliche Spiel") ist ein strategisches Brettspiel für zwei Spieler, bei dem der Zufall keine Rolle spielt (außer beim Losen um die Farbe, d. h. um den ersten Zug), sondern lediglich das Können der Spieler über den Spielausgang entscheidet. Es bietet daher ideale Voraussetzungen, um als

Benchmarkproblem für verschiedene Algorithmen zu dienen, zudem beim Schach auch nur eine eng begrenzte Menge von Objekten, wie die Felder, Figuren und Züge, existieren. Die Regeln sind genauso eindeutig und einfach zu beschreiben, wie auch das große Ziel des Spiels, das Schachmatt. Dennoch besitzt diese Welt eine so hohe Komplexität, dass es bis heute nicht gelungen ist, die spieltheoretischen Gesetzmäßigkeiten des Spiels ganz zu durchschauen, siehe Abschnitt 1.4. Die Struktur des Schachspiels macht es *Brute Force* Verfahren nicht möglich gute Lösungen zu erreichen, so dass *intelligente Techniken* verwendet werden müssen. Gleichzeitig können Techniken, die in Schachprogrammen verwendet werden, auch für die Mustererkennung, Problemlösung, Optimierungsheuristiken und andere *real world* Probleme genutzt werden.

Mit Hilfe der Genetischen Programmierung und Evolutionärer Algorithmen sollen in dieser Arbeit Programme entwickelt werden, die Schach spielen können. Während der Versuche hat sich sehr schnell gezeigt, dass die bekannten Individuenstrukturen zur Lösung der Aufgabe nicht flexibel genug sind. Daher wurden zwei neue Strukturen für die Genetische Programmierung entwickelt. Die Linear-Tree und Linear-Graph Strukturen vereinen durch ihren hybriden Aufbau den Programmfluss und den Datenfluss in einer Struktur. Die Analyse der Struktur hat ihre Leistungsfähigkeit auch für andere Problemstellungen gezeigt. Das Schachproblem ist aber ein Suchproblem und hier wurde auf der Grundlage der Linear-Graph Struktur ein spezialisiertes Individuum entwickelt, das durch seine Struktur und sein Verhalten gut zur Lösung von Suchproblemen genutzt werden kann.

Neben diesen neuen Strukturen werden den Individuen sehr viele Funktionen mit Problemwissen zur Verfügung gestellt. Denn für viele Problemstellungen sind in der Literatur und Praxis schon Verfahren und Funktionen bekannt, die Teilprobleme lösen können. Das System kann dann geeignete Kombinationen dieser Funktionen finden, die es dem Individuum gestatten die Problemstellung gut zu lösen. Durch die Einschränkung des Systems auf spezialisierte Funktionen wird die Evolution zwar auf bestimmte Lösungsansätze festgelegt, aber durch den eingeschränkten Suchraum wird auch die Möglichkeit deutlich verbessert, überhaupt eine Lösung zu finden. Der Ansatz nutzt Wissen über die Problemstellung aus und versucht so schneller zu einem Ergebnis und zu einer Verbesserung bekannter Ansätze zu gelangen. Dies spiegelt sich auch in den verschiedenen GP-Strukturen wieder, die für das System entwickelt wurden. Neben der Lösung des Schachproblems geht es auch darum, durch eine Modularisierung und Nutzung von Vorwissen mit GP-Systemen neue Möglichkeiten zu entwickeln um komplexe und *real world* Probleme zu lösen.

Im ersten Kapitel *Einführung in die Spieltheorie* werden der theoretische Hintergrund von Spielen kurz vorgestellt und auch die Eingliederung des Schachspiels in diesen Zusammenhang beschrieben. Im zweiten Kapitel *Schach* werden die wichtigen Elemente zum Schachspiel vorgestellt sowie die Entwicklung des Computerschachs und die Elemente der Schachprogramme mit ihrer Funktion. Das dritte Kapitel *GP Einführung* gibt eine kurze Einführung in das Gebiet der Genetischen Programmierung und erläutert die grundlegenden Begriffe. Im Kapitel vier werden die drei für diese Arbeit neu entwickelten Strukturen vorgestellt. Daran schließt sich eine empirische Analyse der Linear-Tree und Linear-Graph Struktur mit Hilfe verschiedener Benchmarkprobleme an. Hierzu werden die neuen Strukturen auf Benchmarkproblemen mit anderen Strukturen der Genetischen Programmierung verglichen. Das fünfte Kapitel *Das GeneticChess-System* beschreibt den Aufbau des GeneticChess Systems und der Vorgängersysteme. Daran schließt sich das Kapitel mit der Auswertung der Ergebnisse an. Im siebten Kapitel *Fazit und Ausblick* werden die Ergebnisse noch einmal kurz zusammengefasst und die möglichen Weiterentwicklungen des Systems vorgestellt. Im Anhang findet man eine Auflistung aller Funktionen, die in den verschiedenen Schachsystemen verwendet wurden, mit einer kurzen Beschreibung der Funktion.

Kapitel 1

Einführung in die Spieltheorie

1.1 Einführung

Die Spieltheorie ist eine mathematische Theorie, die sich mit strategischem Denken in Situationen befasst, in denen Entscheidungsträger miteinander interagieren. Das Ergebnis einer Entscheidung hängt nicht nur von dem Verhalten eines *Spielers* ab, sondern wird gleichzeitig durch die Entscheidungen aller beteiligten Spieler determiniert. Die in der Spieltheorie als *Spiele* bezeichneten Entscheidungssituationen stimmen nicht nur mit den Spielen im üblichen Sinn des Wortes überein. Seit John von Neumann und Oskar Morgenstern [125] werden *Spiele* als eine wissenschaftliche Metapher für ein weites Feld von Interaktionen zwischen Entscheidungsträgern gesehen, deren Ausgang von Strategien zweier oder mehr Personen bestimmt werden. Spiele sind somit alle interaktiven Entscheidungsprobleme, an denen interagierende Entscheidungsträger teilnehmen.

Die Spieltheorie basiert dabei auf zwei zentralen Annahmen:

- Jeder Entscheidungsträger verfolgt ein gegebenes Ziel.
- Er berücksichtiget dabei sein Wissen oder seine Erwartung über das Verhalten anderer Entscheidungsträger.

Die Entwicklung der Spieltheorie vollzog sich knapp zusammengefasst in den folgenden Schritten.

- Beginn der mathematische Analyse von Gesellschaftsspielen um die Jahrhundertwende.
- Zu Beginn der Analyse konzentriert sich die Forschung auf Nullsummenspiele.
- John von Neumann und Oskar Morgenstern veröffentlichen 1944 *The Theory of Games and Economic Behavior* [125], sie stellten die Spieltheorie auf ein einheitliches mathematisches Fundament.
- Entwicklung der kooperativen und nicht-kooperativen Spieltheorie. Sie erklärt zahlreicher Phänomene in der Industrieökonomik, Außenhandelstheorie, Makroökonomik, politischen Ökonomie.
- Aus den Erkenntnissen und Methoden der Spieltheorie entwickelt sich die Evolutionäre Spieltheorie, die Lerntheorie und die Vertragstheorie.

Die Methoden der Spieltheorie werden heute nicht hauptsächlich im Poker, Baseball, Fußball und Schach angewendet, sondern im Firmenwettbewerb, dem Arms Race, der Umweltverschmutzung oder dem Verhältnis von Staaten untereinander.

Da der Fokus dieser Arbeit auf dem Schachspiel liegt, werden die Elemente der Spieltheorie, welche auf das Schachspiel anwendbar sind, intensiver beschrieben und die restlichen Elemente der Theorie nur gestreift. Obwohl die Spieltheorie mathematisch und formell definiert ist, wird im folgenden so weit wie möglich darauf verzichtet, um die Kernaussagen der Theorie einfacher darzustellen. Eine einfache Einführung zur Spieltheorie ist in [71] zu finden.

1.2 Grundlegende Elemente der Spieltheorie

Die Spieltheorie beschäftigt sich mit Entscheidungssituationen, die als *Spiele* bezeichnet werden. Der Begriff *Spiel* ist zwar eine Metapher für Interaktionen zwischen Spielern, jedoch sind die Interaktionen von den *Strategien* der Spieler abhängig. Dies bedeutet aber auch, dass die **Spieltheorie sich ausschließlich mit strategischen Spielen beschäftigt**. Sie befasst sich nicht mit Glücksspielen, bei denen die Resultate allein vom Zufall abhängen und der Spieler somit keine persönliche Entscheidung getroffen hat.

In der Spieltheorie werden zwei Typen von *Entscheidungssituationen* unterschieden, diese sind die *nichtkooperativen* und *kooperativen* Situationen.

Eine nichtkooperative Situation im Sinne der Spieltheorie liegt vor, wenn der Vorteil der einen Seite einem Nachteil der anderen Seite entspricht. Dagegen liegt dies nicht bei einer kooperativen Situation vor.

Das Ziel der Spieltheorie ist es, für reale Entscheidungssituationen in ökonomischen, politischen oder militärischen Bereichen bestimmte spieltheoretische Modelle anzugeben, um mit ihrer Hilfe die Situation beschreiben zu können.

Um eine Situation zu beschreiben und ein theoretisches Modell aufzubauen, müssen zuerst die Grundlagen der Theorie gelegt werden. Neben der Definition des Spiels ansich benötigt man auch Instrumente um ein Spiel durchzuführen, zu bewerten und den Prozess des Spiels zu beschreiben.

1.2.1 Nutzenfunktion

In jeder Entscheidungssituation während eines Spiels hat der Spieler mehrere Möglichkeiten, aus dieser Menge muss er nun die *günstigste* Variante auswählen. Es muss also ein Maßstab vorhanden sein, mit dem festgestellt werden kann, ob eine Entscheidung besser ist als eine andere. Das heißt, die einzelnen Entscheidungen müssen bewertet werden können und ihr Nutzen muss bestimmt werden können.

In der Spieltheorie ist eine Entscheidung A nützlicher als eine Entscheidung B, wenn man A B vorzieht.

Wenn man diese Aussage auf das Schachspiel bezieht und die drei möglichen Resultate einer Partie betrachtet, den Gewinn (G) der Partie, den Verlust (V) der Partie und das Unentschieden (U), so kann man die drei Resultate sofort miteinander vergleichen und folgende Bewertung vornehmen:

$$G > U > V.$$

In der Spieltheorie wird nun der Zusammenhang zwischen den Entscheidungen und den Spielresultaten durch eine Funktion ausgedrückt. Die *Nutzenfunktion* f gibt für die verschiedenen Spielausgänge x den Nutzen N für den Spieler in der Entscheidungssituation an, so ist

$$N = f(x)$$

die spieltheoretisch exakte Darstellung für die angegebenen Ordnungsrelation.

Damit ist eine der Voraussetzungen geschaffen, um spieltheoretische Probleme mathematisch zu formulieren. So kann man nun die Suche des Spielers nach der *günstigsten* Variante in einer Entscheidungssituation auch so formulieren: **Der Spieler versucht den maximalen Wert der Nutzenfunktion zu erlangen.**

Eine wichtige Voraussetzung für die Nutzenfunktion in der Spieltheorie ist, dass die an einer Entscheidungssituation beteiligten Spieler die gleiche Nutzenfunktion verfolgen.

1.2.2 Strategische Spiele

In der abstrakten Betrachtungsweise der Spieltheorie kann ein *strategisches Spiel* (im folgenden einfach Spiel genannt) **als die Gesamtheit der es bestimmenden Spielregeln aufgefasst werden**. Jedoch gibt es keine streng mathematische Fassung des Begriffs des strategischen Spiels in der Spieltheorie, für spezielle Typen von Spielen, wie den *Zweipersonen-Nullsummen Spielen* (siehe Abschnitt 1.4), ist dies aber gelungen. Das strategische Spiel oder Spiel ist in der Spieltheorie nicht die Realisierung des Spiels, sondern nur das *System von Spielregeln*, welches das Spiel definiert. Die konkrete Realisierung von Spielen wird in der Spieltheorie als *Partien* bezeichnet. Dadurch kann man nun ein Spiel als die Gesamtheit aller denkbaren Partien auffassen, was später noch einmal wichtig wird.

Die Spielregeln eines Spiels können sich auf vier Typen beziehen:

1. auf die Anzahl der am Spiel beteiligten Personen,
2. auf die Festlegung darüber, was in einem Spiel unter *Zügen* zu verstehen ist und wie diese durchzuführen sind,
3. auf welche Weise die einzelnen Züge der Spieler aufeinanderfolgen
4. und auf das quantitative Resultat des Spiels, nach der Durchführung aller Züge.

Wenn es sich bei einem Spiel um ein Gesellschaftsspiel handelt, ist sofort klar, was unter Spielregeln zu verstehen ist. Beim Schachspiel sind es die bestehenden Vereinbarungen über die möglichen Bewegungen der verschiedenen Figuren und die Vereinbarung, wann eine Partie zu Ende ist und wie der Ausgang der Partie (Sieg, Verlust oder Unentschieden) zu bewerten ist. Größere Probleme ergeben sich, wenn man eine reale Entscheidungssituation durch die Spieltheorie modelliert.

Einen *Zug* kann man etwa beim Schach einfach festlegen, hier ist jedes Verrücken einer Figur ein Zug. Bei einem *realen Spiel* ergeben sich wieder Probleme, hier werden aber objektive Möglichkeiten der handelnden Seiten als Regeln für die Züge und deren Reihenfolge aufgefasst.

Ebenso wie der Begriff des Zuges ist auch der Begriff der *Strategie* in der Spieltheorie sehr abstrakt und im theoretischen und mathematischen Umfeld zu sehen. **Eine Strategie ist eine *beliebige* Folge von Zügen für ein Spiel**. Normalerweise wird unter einer Strategie nicht eine *beliebige Folge* von Zügen bezeichnet verstanden, sondern eine Folge von Zügen, mit der ein bestimmtes Ziel erreicht werden soll. In der abstrakten spieltheoretischen Analyse wird jedoch jede Zugfolge, die eine Partie beendet, als eine Strategie bezeichnet. Die Bewertung dieser Strategie erfolgt dann erst durch die spieltheoretische Analyse. Im intuitiven Sinne wird davon ausgegangen, dass die Spieler ihre

Zugfolge von Zug zu Zug bestimmen. **In der Spieltheorie wird eine Strategie, also eine Zugfolge, zu Beginn des Spiels festgelegt**. Dies erfordert natürlich, dass das Spiel durch die Spielregeln vollständig bestimmt ist und alle möglichen Situationen, die im Spiel auftreten können, von vornherein erfassbar sind.

Die spieltheoretische Sicht einer Strategie, also die Annahme, dass alle strategischen Entscheidungen vor Beginn eines Spiels getroffen worden sind, hat in der Realität erhebliche Probleme. Wenn man das Schachspiel als Beispiel nimmt, so hat man schon bei 50 Halbzügen einen Baum mit $1,3 * 10^{80}$ Stellungen, dies würde dann $3,2 * 10^{78}$ Strategien entsprechen[1]. Wenn beide Spieler von der Schachpartie eine Strategie im Sinne der Spieltheorie festlegen würden, wäre die betreffende Partie völlig festgelegt und schon vor der Partie stünde der Gewinner fest.

Bei dem letzten Typ von Regeln für ein Spiel geht es um das quantitative Resultat eines Spiels. In der Spieltheorie wird dies auch als die *Auszahlung* bezeichnet. Gewinne werden bei einer Auszahlung durch *positive* Zahlenwerte bezeichnet und Verluste durch *negative* Werte. Das Resultat einer bestimmten Partie ist nicht allein von der Strategie abhängig, die ein Spieler gewählt hat, sondern von allen Strategien, die von allen beteiligten Spielern ausgewählt worden sind. Man kann daher den Gewinn, den ein Spieler erzielt, als eine Funktion der von allen Spielern gewählten Strategien auffassen. Man nennt diese Funktion die *Auszahlungsfunktion* des betroffenen Spielers. Bei einem Zweipersonenspiel gibt es zwei Auszahlungsfunktionen, die durch die beiden Strategien der Spieler bestimmt werden.

1.2.3 Spielbäume

Die *extensive Form* zur Darstellung von Spielen ist sehr intuitiv, besonders bei Spielen, bei denen alle Züge offen sind und keine verdeckten Züge erfolgen[2], dies wird in der Spieltheorie auch mit *vollständiger* und *unvollständiger* Information bezeichnet, hierzu später mehr. Im Schachspiel erfolgen alle Züge offen, beide Spieler haben also die *vollständige Information* über das Spiel. Die extensive Darstellung eines Schachspiels wäre nun der vollständige Spielbaum, der mit der Ausgangsstellung beginnend alle Stellungen als Knoten und die Züge zwischen den Stellungen als Kanten darstellt. Da das Schachspiel zu komplex ist um den vollständigen Spielbaum darzustellen, wird der Suchbaum hier am *Nimmspiel* beschrieben. Das Spiel hat folgende Regeln:

- Das Spiel beginnt mit fünf Spielsteinen.
- Es gibt zwei Spieler sie werden A und B genannt.
- Die Spieler ziehen abwechselnd und dürfen einen oder zwei Steine wegnehmen.
- Der Spieler hat gewonnen, der den letzten Stein wegnimmt.

Die Abbildung 1.1 zeigt nun den vollständigen Suchbaum für das Nimmspiel mit fünf Steinen. Die Auszahlung für den Spieler A wird dabei durch $+1$ und -1 an den Blättern des Baumes gekennzeichnet. Der Spielbaum bietet eine dynamische Sicht auf das gesamte Spiel, mit allen Handlungsmöglichkeiten der Spieler. Es wird aber auch deutlich, dass schon ein Spielbaum mit 20 Knoten unübersichtlich wird, denn der Spielbaum enthält eine Gewinnstrategie für den Spieler, der anfängt. Sie ist jedoch nicht direkt zu erkennen, denn die Auszahlungen des Baumes scheinen ausgeglichen zu sein. Wenn aber der Spieler

[1] Jedes Blatt des Spielbaumes repräsentiert hier eine Strategie und somit ist die Zahl der Blätter des Baumes gleich der Zahl der Strategien im Baum.

[2] In Kartenspielen treten verdeckte Züge auf, wie die Verteilung der Karte.

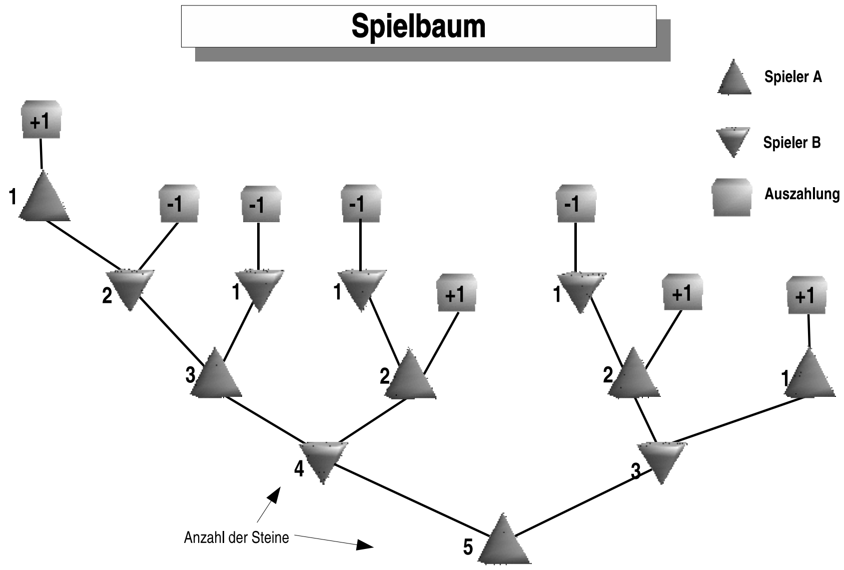

Abbildung 1.1: Spielbaum eines Nimmspiels mit fünf Steinen.

A am Anfang zwei Steine nimmt, kann Spieler B nicht mehr gewinnen, egal, ob er einen oder zwei Steine wegnimmt.

Der Spielbaum muss nicht immer endlich sein, es ist auch möglich zu *unendlichen Bäumen* überzugehen und es können auch *nicht offene Züge* in Bäumen dargestellt werden. Wobei dann nicht die Darstellung des Baumes an Bedeutung gewinnt, sondern seine *abstrakte mathematische Struktur* und deren Möglichkeiten zur Analyse.

1.2.4 Matrizen

Die *Normalform* für Spiele ist eine Darstellung, die zu den jeweiligen möglichen Strategienkombinationen der beteiligten Spieler gehörenden Auszahlungswerte in einer *Matrix* angibt. Bei zwei Spielern ist diese Art der Darstellung sehr übersichtlich, dann werden in der ersten Spalte die Strategien des ersten Spielers aufgeschrieben und in der ersten Zeile werden die Strategien des zweiten Spielers aufgeschrieben. In den Zellen der Matrix wird dann für jedes *Strategiepaar* die entsprechende Auszahlung aufgeschrieben. Diese Art der Darstellung wird auch als *Auszahlungsmatrix* bezeichnet.

A B	**1**	**1,1**	**1,2**	**2**	**2,1**
1,1			-1		-1
1,2		-1		+1	
1,1,1		+1			
2,1		-1		+1	
2,2	+1				

Tabelle 1.1: Die Auszahlungsmatrix für das Nimmspiel, das durch den Spielbaum 1.1 repräsentiert wird.

Die Tabelle 1.1 stellt die Auszahlungsmatrix des Nimmspiel dar, welches in der Abbildung 1.1 durch einen Spielbaum beschrieben wird. Bei der Darstellung eines Spiels in *Normalform* wird die Dynamik des Spiels nicht mehr deutlich. Diese Darstellung zeigt aber sofort, welche Strategiepaare für ein Spiel überhaupt möglich sind. Denn alle freien Felder der Matrix stellen Strategiepaare dar, die nicht möglich sind. Hier wird jetzt auch noch einmal die Bedeutung einer Strategie für die Spieltheorie deutlich. Sie wird hier als eine *beliebige* Folge von Zügen für ein Spiel definiert, die vor dem Spiel von den Spielern festgelegt wird (siehe auch Abschnitt 1.2.2). Das heißt, wählt der Spieler **A** die Strategie **1,1,1**, so kann der Spieler **B** nur die Strategie **1,1** wählen, würde **B** die Strategie **1** wählen, wäre das Spiel nicht beendet.

Bei dem Beispiel des Nimmspiels sind alle Züge offen, es gibt aber auch Spiele, in denen die Züge nicht offen sind und die Spieler so nur eine unvollständige Information haben. Für diese Spiele ist die Normalform zur Analyse der Spiele besser geeignet als die extensive Form. Bei diesen Spielen ist der Verlauf einer Partie weniger entscheidend für die Analyse als die gewählte Strategie der Spieler und das Resultat. In diesem Fall ist die Entscheidung über die Strategie ohnehin nicht von Zügen des anderen Spielers abhängig, da sie ja nicht bekannt sind.

Die Normalform ist die abstraktere Darstellungsform eines Spiels, sie wird jedoch in der Spieltheorie hauptsächlich verwendet. Mathematisch gesehen ist sie mit der *extensiven Form* jedoch identisch.

1.3 Typen von Spielen

Die Spieltheorie befasst sich zwar ausschließlich mit strategischen Spielen, aber auch in dieser Gruppe von Spielen gibt es eine große Anzahl verschiedener Spieltypen. Diese Typen unterscheiden sich nicht nur in der Definition oder der Beschreibung des Spiels, sondern auch in ihrer mathematischen Problemstellung, so dass bei ihrer Analyse unterschiedliche Methoden angewendet werden müssen. Im Zusammenhang mit den verschiedenen Spieltypen ist auch das *Reduktionsprinzip* wichtig, auf welches später noch genauer eingegangen wird.

Zur Unterscheidung von Spielen gibt es verschiedene Haupt- und Untermerkmale und verschiedene Kombinationsmöglichkeiten dieser Merkmale. Die Hauptmerkmale zur Unterscheidung von Spielen sind:

- die Anzahl der beteiligten Spieler,
- die Art der im Spiel zugelassenen Züge und Strategien
- und die Art der Auszahlungfunktion.

Im folgenden soll auf diese Merkmale etwas genauer eingegangen werden.

1.3.1 Anzahl der Spielpartner

Ein Spiel verlangt mindestens zwei Spieler, die an dem Spiel teilnehmen. Die *Zweipersonenspiele* sind die *einfachsten Spieltypen*.

Die mathematische Theorie für diesen Spieltyp ist besonders weit entwickelt. Dies liegt auch daran, dass viele Spiele mit mehr als zwei Spieler auf ein Zweipersonenspiel *reduziert* werden können. Die einfachste Art der Reduktion ist dabei der Zusammenschluss aller am Spiel beteiligten Spieler zu zwei einander entgegengesetzten Parteien. In der Spieltheorie spricht man dann von der Bildung einer *Koalition* und einer *Gegenkoalition*. Eine Koalition als einen Spieler im Sinne der Spieltheorie zu verstehen ist nur dann möglich, wenn von den abweichenden Interessen der einzelnen Spieler in der Koalition abstrahiert werden kann.

Bei vielen Gesellschaftsspielen ist die Bildung von Koalitionen durch die Spielregeln aber verboten, auch reale Situationen kann man nicht immer auf zwei Koalitionen reduzieren. In einem solchen Fall spricht man dann von *Mehrpersonenspielen*. Die starke Trennung zwischen Zweipersonenspielen und Mehrpersonenspielen in der Spieltheorie liegt daran, dass die Probleme bei Mehrpersonenspielen wesentlich schwieriger zu analysieren sind und die mathematische Theorie mit zunehmender Anzahl von Spielern immer komplizierter wird. Dies ist ein Grund, warum die mathematische Theorie über Mehrpersonenspiele nicht so gut ausgebaut ist wie über die Zweipersonenspiele.

1.3.2 Züge und Strategien

Bei der Unterscheidung von Spielen durch die Art der Züge gibt es verschiedene Möglichkeiten, das wichtigste Unterscheidungsmerkmal ist, ob es ausschließlich *persönliche* Züge oder auch *zufällige* Züge gibt. Unter persönlichen Zügen versteht man Züge, die nur von der Entscheidung eines der Spieler abhängen. Zufällige Züge dagegen sind Handlungen, die nicht von der Entscheidung eines der Spieler abhängen, die an dem Spiel teilnehmen, sondern zufallsbedingt sind. Spiele, in denen *nur* Zufallszüge auftreten, sind nicht Gegenstand der Spieltheorie. Denn wenn ein Spiel nicht von Entscheidungen eines Spielers abhängig ist, kann es in solchen Spielen auch keine Strategie geben. **Gegenstand der Spieltheorie sind strategische Spiele mit ausschließlich persönlichen Zügen oder in denen sowohl persönliche als auch zufällige Züge auftreten.**

Das Unterscheidungsmerkmal von Spielen nach dem Charakter der Züge ist eng mit der von den Spielern verfolgten Strategie verbunden. Eine *Strategie* ist bisher immer als eine diskrete Folge von Zügen dargestellt worden, diese Betrachtungsweise ist auch für die meisten Spiele gültig, im Besonderen für das Schachspiel. Diese Spiele werden als *diskontinuierliche Spiele* bezeichnet. Es muss aber darauf hingewiesen werden, dass die Spieltheorie auch Züge betrachtet, in denen Züge als ein Intervall der reellen Zahlen interpretiert wird. Spiele, in denen solche Züge auftreten, heißen *kontinuierliche Spiele*.

Der Strategiebegriff ist einer der fundamentalen Begriffe der Spieltheorie, denn sie soll Empfehlungen für rationales Verhalten herleiten. Dies bedeutet, dass die Spieltheorie für ein bestimmtes Spiel eine möglichst *optimale Strategie* bestimmen soll. Der Begriff der *optimalen Strategie* wird hier im Zusammenhang mit den *Zweipersonen-Nullsummenspielen* entwickelt (siehe hierzu Abschnitt 1.4). Sofern es aber optimale Strategien für ein Spiel gibt, so gibt es auch *nichtoptimale Strategie* oder *suboptimale Strategie*. Bei der Analyse von Strategien sind viele Methoden darauf ausgerichtet festzustellen, ob überhaupt optimale Strategien existieren oder nicht. Die weitere Analyse versucht dann die optimale Strategie zu bestimmen, dies ist jedoch komplizierter. So gibt es viele Typen von Spielen, für die es gelungen ist zu zeigen, dass es eine optimale Strategie gibt, aber die Bestimmung der optimalen Strategie noch nicht gelungen ist. Ein Beispiel für diese Situation ist das Schachspiel.

Wichtig im Zusammenhang mit Strategien ist die Unterscheidung zwischen *reinen Strategien* und *gemischten Strategien*. Unter einer *reinen Strategie* versteht man eine Strategie im intuitiven Sinne. Eine *gemischte Strategie* kann man sich so vorstellen, dass der Spieler die Entscheidung für eine seiner reinen Strategien nicht mehr selbst trifft, sondern von einem Zufallsmechanismus abhängig macht. Der Spieler bestimmt lediglich noch die Wahrscheinlichkeiten, mit der der Zufallsmechanismus die einzelnen Strategien wählt. Damit wird die Bestimmung der Wahrscheinlichkeiten zur eigentlichen Strategie des Spielers. Er muss sich nun überlegen, mit welcher Wahrscheinlichkeitsverteilung er den höchsten erwarteten Nutzen realisiert. Bei reinen Strategien wird an einer Strategie festgehalten und bei einer gemischten Strategie wird diese immer wieder gewechselt. Die Unterscheidung zwischen diesen beiden Strategiearten wird jedoch nicht zur Unterscheidung von

Spielen genutzt.

Ein weiteres Unterscheidungsmerkmal für Spiele ist es, ob die Strategien bzw. die Züge, aus denen sich die Strategien zusammensetzen, *offen* oder *verdeckt* erfolgen. Bei *offenen Zügen* ist jeder Spieler in jeder Phase des Spiels über die aktuelle Situation genau informiert. In einem Spiel mit *verdeckten Zügen* jedoch kennt der Spieler die Züge des Gegners teilweise nicht oder alle nicht. Man spricht in diesem Zusammenhang dann auch von Spielen mit *vollständiger Information* oder mit *unvollständiger Information*. Das Schachspiel ist ein Spiel mit *vollständiger Information*, denn hier kennt jeder Spieler zum Zeitpunkt des Zuges die Resultate aller vergangenen Züge beider Seiten. Das Skatspiel ist ein Beispiel für Spiele mit *unvollständiger Information*, denn hier ist nicht bekannt, welche Karten der andere Spieler in der Hand hat.

Ein weiteres Unterscheidungsmerkmal von Spielen ist die *Anzahl möglicher Strategien*. Danach kann man Spieler mit *endlichen* und *unendlichen Strategien* unterscheiden. Bei *endlichen Spielen* hat jeder Spieler nur eine endliche Anzahl von Strategien, aus denen er wählen kann. Bei *unendlichen Spielen* hat mindestens einer der beteiligten Spieler unendlich viele Strategien zur Verfügung. Alle bisher vorgestellten Spiele waren endliche Spiele, wobei die Anzahl der Strategien teilweise sehr groß sein kann. Diese trifft zum Beispiel auf das Schachspiel zu.

1.3.3 Gewinnverteilung

Die *Gewinnverteilung* ist ein weiteres Merkmal um Spiele zu unterscheiden. Ist die Gewinnverteilung für die beteiligten Spieler gleich und hängt nur von der gewählten Strategie ab, so nennt man die Spiele *gerechte (faire)* Spiele. Bei *ungerechten (unfairen)* Spielen sind bestimmte Spieler durch die Spielregeln bevorzugt, sie haben höhere Gewinnaussichten durch die Spielregeln zu erwarten. Das Nimmspiel, welches weiter oben vorgestellt wurde, ist ein unfaires Spiel, da immer der Spieler gewinnt, der beginnt und die Gewinnstrategie kennt.

Ein wichtiger Spieltyp der *fairen Spiele* ist das *Nullsummenspiel*, denn für diesen Spieltyp ist die Spieltheorie am weitesten ausgebaut. **In Nullsummenspielen sind die Gewinne aller beteiligten Spieler zusammengenommen stets gleich null.** Ein Spezialfall der Nullsummenspiele sind die *Zweipersonen-Nullsummenspiele*, für diese Spiele gibt es eine vollständig abgeschlossene mathematische Theorie, Näheres dazu in Abschnitt 1.4.

Nullsummenspiele sind ein Spezialfall einer umfassenderen Klasse von Spielen, den *Konstantsummenspielen*. Im Fall der *Konstantsummenspielen* ist die Summe der Gewinne aller beteiligten Spieler gleich einem konstanten Wert, der ungleich null ist. Eine weitere Verallgemeinerung der Konstantsummenspiele besteht darin, die Gewinnauszahlung nicht konstant zu halten, sondern variabel zu gestalten, diese *Nicht Konstantsummenspielen* sind noch schwerer zu analysieren als Konstantsummenspiele. Daher werden auch in diesem Bereich Möglichkeiten zur Reduktion, wie bei den Mehrpersonenspielen genutzt. Dabei wird versucht Nicht-Nullsummenspiele auf Nullsummenspiele zurückzuführen. Allerdings nimmt man dabei in Kauf, dass die Anzahl der Spieler um einen erhöht wird, so dass aus einem Zweipersonenspiel ein Dreipersonenspiel wird. Die Aufgabe dieses fiktiven Spielers ist es nur, die Differenz in der Gewinnausschüttung zu kompensieren, so dass man wieder ein Nullsummenspiel erreicht.

1.4 Zweipersonen-Nullsummenspiele

Zweipersonen-Nullsummenspiele sind Spiele, die sich aus den vorangegangenen Abschnitten wie folgt definieren lassen. **Zweipersonen-Nullsummenspiele sind strategische**

Spiele, an denen zwei Personen beteiligt sind und bei denen sich die erzielbaren Gewinne für jedes Strategiepaar ausgleichen.

Die Gewinne und Verluste bei Zweipersonen-Nullsummenspielen ergeben sich in Abhängigkeit von den verschiedenen Kombinationen der Strategien beider Spieler, daher können sie in einem quadratischen Schema erfasst werden. Dieses Schema wird auch als *Spielmatrix* bezeichnet, daher bezeichnet man diese Spiele auch als *Matrixspiele*. Die Spieltheorie versucht nun mit Hilfe der Spielmatrix, die beste Spielweise (die optimale Strategie) zu bestimmen. In der Theorie der Matrixspiele versteht man unter einer *optimalen Strategie* die Strategie, mit der ein Spieler den höchstmöglichen Gewinn in dem Spiel erreichen kann. Spiele, für die eine optimale Strategie existiert, heißen *lösbar*. Für eine optimale Strategie wird auch vorausgesetzt, dass der maximale Gewinn nicht für jede Partie gesichert werden kann. Er kann nur im Durchschnitt über viele Partien erreicht werden, somit wird die Angabe des Gewinns auf der Grundlage einer optimalen Strategie zu einer Wahrscheinlichkeitsaussage über den Gewinn. Ein wichtiges Prinzip für eine optimale Strategie ist, **dass beide Spieler vernünftig handeln und alles tun, um zu verhindern, dass der Gewinn des Gegners maximiert wird.** Dieses Prinzip wird auch die *Minimax-Strategie* genannt, man kann diese sehr leicht an einem Beispiel verdeutlichen. In der Tabelle 1.2 wird die Auszahlungsmatrix eines Spiels dargestellt, anhand dessen man eine optimale Strategie nach dem Minimax-Prinzip herleiten kann.

A B	B_1	B_2	B_3	B_4	**Zeilenminima**
A_1	4	5	9	3	3
A_2	5	4	3	7	3
A_3	7	**6**	8	9	**6**
A_4	6	2	4	6	2
Spaltenmaxima	7	**6**	9	9	

Tabelle 1.2: Die Auszahlungsmatrix für ein Beispiel mit Zeilenminima und Spaltenmaxima. Die optimalen Werte für die Zeilenminima, Spaltenmaxima und der Sattelpunkt sind fett gedruckt.

Die Zeilenminima für das Spiel zeigen, dass die Strategie A_3 den *maximal* möglichen Gewinn unter den möglichen Strategien für Spieler A bedeutet, egal welche Strategie Spieler B wählt. Wenn der Spieler B auch der Minimax-Strategie folgt, muss er unter den Spaltenmaxima den kleinsten Wert suchen, dieser wird durch die Strategie B_2 erreicht. Der Schnittpunkt beider Strategien in der Matrix wird der *Sattelpunkt* genannt, in der Tabelle 1.2 fettgedruckt. Der Sattelpunkt hat die Eigenschaft, dass er sowohl das Minimum eines Parameters ist als auch das Maximum eines anderen. Der Sattelpunkt bei Matrixspielen in der Minimax-Strategie ist ein Sonderfall, er ist aber ein *wichtiger Sonderfall*, da für Spiele mit einem Sattelpunkt die Minimax-Strategie *stabil* ist.

Wenn man nun die Matrix an der Position A_3, B_1 zu 6 verändert, ergibt sich auch für die Spalte B_1 ein Spaltenmaximum von 6. Dadurch wäre der Spieler B, wenn er der Minimax-Strategie folgt, nicht auf die Strategie B_2 festgelegt, sondern kann auch die Strategie B_1 wählen. In diesem Fall wäre die *Minimax-Strategie instabil* und somit auch keine optimale Strategie.

Es lässt sich aber zeigen, dass **bei Spielen mit einem Sattelpunkt die Minimax-Strategien optimale Strategien sind**. Obwohl Matrixspiele mit einem Sattelpunkt ein Sonderfall sind, gilt aber auch, dass **jedes Spiel mit vollständiger Information einen Sattelpunkt hat**. Daraus folgt für das Schachspiel, da es ein Spiel mit vollständiger Information ist, dass es einen Sattelpunkt hat und dass die Minimax-Strategie für das Schachspiel optimal ist, jedoch konnte dieses optimale Paar von Minimax-Strategien für das Schachspiel noch nicht gefunden werden.

In der Spieltheorie wurde weiterhin gezeigt, dass **der Sattelpunkt bei Nullsummenspielen ein Nash-Gleichgewicht ist.** Allgemeiner konnte gezeigt werden, dass jedes N-Personen Nicht-Nullsummenspiel ein Nash-Gleichgewicht hat.

Das *Nash-Gleichgewicht* ist ein wichtiger Satz in der Spieltheorie, der besagt: **Wenn es in einem Spiel eine Strategiekombination gibt, für die gilt, dass es für keinen Spieler profitabel ist seine Strategie zu ändern, bilden die Strategiekombination und die resultierenden Payoffs ein Nash-Gleichgewicht** (siehe [84, 83]).

John von Neumann konnte zeigen, dass jedes **Zweipersonen-Nullsummenspiel eine optimale Lösung in den gemischten Strategien wenn nicht in einer reinen Strategie hat.** Dieser Satz ist deshalb so bedeutungsvoll, da er die Suche nach einer optimalen Lösung des Spiels sinnvoll macht. Dieser Satz gilt auch für alle Spiele, die auf Zweipersonen-Nullsummenspiele zurückgeführt werden können. Kann ein Spiel aber nicht auf ein Zweipersonen-Nullsummenspiel abgebildet werden, so kann auch nicht die Theorie für diese Spieltypen angewendet werden. Dies ist deshalb sehr wichtig, da bei Nichtnullsummen- bzw. Nicht-Konstantsummenspielen nicht das Minimax-Prinzip angewendet werden kann.

Nun muss für diese Arbeit noch darauf hingewiesen werden, dass das Schachspiel im engen Sinne kein Nullsummenspiel ist, da die Auszahlungfunktion die eines Konstantsummenspiels ist, denn der Gewinner erhält zwei Punkte, der Verlierer null und im Falle des Remis erhalten beide Spieler einen Punkt. Es ist aber leicht zu sehen, dass man durch Änderung der Auszahlungsfunktion ein Nullsummenspiel erhalten kann und somit kann für das Schachspiel auch die Theorie der Zweipersonen-Nullsummenspiele mit vollständiger Information angewendet werden. Da nun die spieltheoretischen Grundlagen für das Schachspiel behandelt wurden, wird auf die weiteren Elemente der Spieltheorie nur kurz eingegangen.

1.5 Nicht Konstantsummenspiele

Die *Nicht Konstantsummenspiele* sind Spiele, deren Auszahlungsfunktion eine variable Summe ist. Die *Konstantsummenspiele* gehören nicht zu der gleichen Klasse wie die Nicht Konstantsummenspiele, denn viele Konstantsummenspiele lassen sich auf Nullsummenspiele zurückführen.

Wenn man das Nicht Konstantsummenspiel auf zwei Personen beschränkt, bei dem beide Spieler über eine endliche Anzahl von Strategien verfügen, kann man auch diese Spiele als Matrixspiele darstellen. Da aber bei diesen Spielen der Gewinn des einen Spielers nicht dem Verlust des anderen entspricht, braucht man zwei Matrizen um das Spiel zu beschreiben. Streng genommen braucht man auch für Zweipersonen-Nullsummenspiele zwei Matrizen, um ein Spiel vollständig zu beschreiben, da sie sich aber nur im Vorzeichen unterscheiden, wird darauf immer verzichtet. Es ist üblich diese Matrizen zu einer *Doppelmatrix* zusammenzufassen. Dann enthält jedes Element der Matrix zwei Zahlen, welche die jeweiligen Auszahlungswerte der beiden Spieler für diese Strategie angeben.

A B	B_1	B_2
A_1	-2, -2	+1, +2
A_2	+2, +1	-2, -2

Tabelle 1.3: Die Doppelmatrix für ein Nicht Konstantsummenspiel, die einzelnen Werte für die beiden Spieler stehen in einer Zelle der Matrix.

Für diesen Spieltyp gibt es keine ähnlich einfache Theorie wie für die Zweipersonen-Nullsummenspiele. Zwar gibt es einen *Existenzsatz über Gleichgewichtspunkte*, aber für den Fall wie er in der Tabelle 1.3 beschreiben wird, gibt es keine *eindeutigen* Lösungen. Dieser Existenzsatz über Gleichgewichtspunkte besagt auch, dass ein Sattelpunkt ein Spezialfall eines Gleichgewichtspunktes im Fall der Nullsummenbedingung ist. Daher kann ein Sattelpunkt auch als Gleichgewichtspunkt aufgefasst werden.

Für Zweipersonen-Konstantsummenspiele können zwar auch mehrere Gleichgewichtspunkte auftreten, aber es lässt sich mathematisch zeigen, dass die Strategiepaare dieser Gleichgewichtspunkte stets zum selben Resultat führen. Man spricht in diesem Zusammenhang davon, dass **alle Gleichgewichtspunkte bzw. die entsprechenden Strategiepaare äquivalent sind.** Somit weist die Theorie der Zweipersonen-Konstantsummenspiele eine größere Ähnlichkeit zu den Zweipersonen-Nullsummenspielen auf als die der Nicht-Konstantsummenspiele.

1.6 Mehrpersonenspiele

Mehrpersonenspiele sind Spiele, an denen mehr als zwei Spieler beteiligt sind. Die Besonderheit dieser Spiele zu den Zweipersonenspielen ist, dass sich Spielpartner zu *Koalitionen* zusammenschließen können. Dadurch ist es oft möglich, die Mehrpersonenspiele auf Zweipersonenspiele zu reduzieren. Wenn es sich dann noch um ein Nullsummenspiel handelt, kann man die gesamte Theorie der Zweipersonen-Nullsummenspiele auf diese Situation anwenden und die optimale Strategie bestimmen. Das eigentliche Problem des Mehrpersonenspiels ist aber noch nicht gelöst, dies ist die **Verteilung der Gewinne unter den Mitgliedern der Koalitionen**. Dies ist ein Hauptproblem der Theorie der Mehrpersonenspiele.

1.7 Evolutionäre Spieltheorie

Die *Evolutionäre Spieltheorie* hat in der neueren Zeit große Bedeutung gefunden. Diese Interpretation der Spieltheorie wurde maßgeblich durch Anwendungen in der Evolutionsbiologie vorangetrieben. John Maynard Smith hat dabei das Konzept der *evolutionär stabilen Strategien (ESS)* entwickelt. Aber auch in der Ökonomik gibt es wichtige Fragestellungen, die mit Hilfe des evolutionären Ansatzes und seiner Weiterentwicklung behandelt werden. Die *Evolutionäre Spieltheorie* untersucht, welche Verhaltensweisen sich in einem Spiel entwickeln, wenn die Spieler ausgehend von einer bestimmten Anfangskonstellation ihr Verhalten gemäß einfacher Regeln anpassen können. Ein wichtiger Aspekt der Theorie ist dabei die Frage, ob die Vorhersagen gestützt werden, die sich aus der Analyse des Verhaltens rationaler Spieler ergeben. Weiterführende Bücher zu diesem Gebiet sind [124, 126].

Während in den Modellen der evolutionären Spieltheorie die Abweichungen von der Annahme der Rationalität der Spieler theoretischer Natur sind, verfolgt die experimentelle Spieltheorie einen empirischen Ansatz. Dieser Zweig der Theorie erfuhr wesentliche Anregungen durch die Psychologie. Obwohl auch dieser Ansatz sehr neu ist, begannen die ersten Experimente schon sehr früh in der RAND Corporation (siehe [60, 61, 31, 32, 33]). Neuere Bücher zur experimentellen Ökonomik sind [26, 25].

Kapitel 2

Schach

Das Spiel Schach fasziniert die Menschen schon seit Jahrhunderten, fast jeder kennt das Spiel und kennt sogar die Spielregeln und über die Entstehung des Schachspiels wurde schon viel philosophiert.

Darlegungen aus der indischen Geschichte sowie Experten-Meinungen lassen darauf schließen, dass das Schachspiel seine Reise etwa im zweiten Drittel des 7. Jahrhunderts in Indien begann.

In dieser Zeit entwickelte sich auch der Name des Schachspiels, dieser entstammt dem Persischen für Schah (König), dort wurde es aber noch *Tschatrang* genannt. Dieser Name hat sich aus dem indischen Namen entwickelt, der *Tschaturanga* lautete, (tschatur=vier, anga=Teil,Abteilung). Ein Heer in Indien setzte sich damals aus vier Teilen (Waffengattungen) zusammen: Infanterie (Bauern), Kavallerie (Pferde), Streitwagen und Elefanten - daher der Sanskritausdruck Tschaturanga. Die indische Variante hatte mit unserem heutigen Schach wenig gemeinsam. Gespielt wurde zu viert auf 64 Feldern - wobei zwei Spieler jeweils eine Mannschaft bildeten - jedoch unter Einsatz von Würfeln. Als es über Handelswege nach Persien gelangte, spielte man es bereits zu zweit ohne Würfel. Von den Persern gelangte das Spiel dann zu den Arabern, die es *Schatrandsch* nannten. Hier ist der heutige Name schon enthalten. Es hat sich dann weiter über die arabischen Länder nach Europa verbreitet, dies geschah um 1000. Es wurde letztendlich auf drei Wegen nach Europa gebracht. Einmal über die russischen Ströme in den Ostseeraum, von dort aus nach Norddeutschland und England, zum anderen durch arabische Vermittlung nach Italien und über die Alpen. Außerdem verbreitete sich das Schachspiel noch durch die Mauren nach Spanien und Frankreich. Die orientalischen Spielregeln blieben dabei stets erhalten, welche für die Dame nur einen Schrägschritt und für den Läufer einen Schrägsprung aufs übernächste Feld vorsehen.

Um 1200 wurden die Schachregeln leicht verändert um das Spiel dynamischer zu gestalten, so durften nun die Bauern einen Doppelschritt aus der Ausgangsreihe vornehmen. Aber die größte Änderung der Schachregeln wurde um 1500 durchgeführt. Man entschloss sich zu einer grundlegenden Reform, welche zu den heutigen Gangarten und Zügen führte. So wurde die Rochade und der en passant Schlagzug eingeführt. Das früher als Partiegewinn gewertete Patt wurde nun zum Remis. Die wichtigsten Änderungen waren aber die Gangart der Königin und des Läufers, sie durften jetzt nicht nur ein Feld weit gehen, sondern über die gesamte Länge des Brettes. Dafür wurde das Überspringen für den Läufer verboten. Seit diesem Zeitpunkt haben sich die Regeln nicht weiter verändert und sich letztlich in die gesamte Welt verbreitet.

2.1 Die Geschichte des Computerschachs

Die Historie künstlicher Gedächtniskunst im Kampf gegen das menschliche Hirn begann mit einem „Fake“. Der ungarische Hofrat Wolfgang von Kempelen (*1734, 1804) präsentierte auf Wunsch der österreichischen Kaiserin Maria Theresia einen vermeintlichen Schachautomaten mit einer eindrucksvollen Mechanik. Hinter dem „mechanischen Türken“ verbarg sich im Inneren der Maschine ein kleinwüchsiger Schachmeister, der die Brettschlachten relativ souverän beherrschte.

Die erste Maschine, die wirklich Schach spielte, konstruierte der Spanier Torres y Quevedo im Jahre 1890. Der Spanier war Ingenieur für elektromechanische Steuerungssysteme. Torres wählte als Aufgabe für seine Maschine einen begrenzten Teilbereich des Schachspiels, das Endspiel König und Turm gegen König. Der Automat war kein Computer, auch kein Vorläufer heutiger Rechenmaschinen, er löste die Aufgabe mit Hilfe von mechanischer Übertragung von magnetischen Relais. Die Maschine existiert noch heute und steht in der Polytechnischen Universität von Madrid.

Bis das erste Schachprogramm auf einem Computer lief, wurden zwar schon Computerprogramme entwickelt, aber noch nicht auf Computern ausgeführt. Das erste Computerprogramm wurde wohl von Konrad Zuse zwischen 1942 und 1945 entwickelt, er wollte aber nur die Möglichkeiten seiner Programmiersprache *Plankalkül* zeigen. Das Programm wurde nie implementiert und erst 1972 veröffentlicht. Im gleichen Zeitraum, zwischen 1945 und 1947, entwickelte auch Alan Turing ein Schachprogramm [122], das korrekte Züge berechnete und eine Position bewerten konnte. Auch er ließ sein Programm mangels eines existierenden Computers nicht laufen, jedoch simulierte er dessen Berechnungen selbst. Im Jahr 1952 kam es zur ersten Partie von Turings *Papiermaschine* und einem Kollegen, Alick Glennie, einem reinen Hobbyspieler. Es entstand ein gleichwertiges Spiel, die Maschine hätte im 22. Zug ohne weiteres gewinnen können. Aber zur Frustration des Mathematikers führten die Berechnungen zu einer schlechteren Fortsetzung. Zum Schluss verlor das Programm durch eine einfache Fesselung.

Unabhängig von Turing hatte in den Vereinigten Staaten Claude E. Shannon, Mitarbeiter der Bell Laboratories, über das gleiche Thema nachgedacht. Am 9. 3. 1949 hielt er einen Vortrag, in dem er die Möglichkeiten erörterte, einen Computer für das Schachspiel zu programmieren. Er hielt diese Aufgabe zwar für praktisch ohne Bedeutung, jedoch aus theoretischen Gründen für äußerst wichtig. Denn die Lösung des Problems könnte, so der Autor, den Anstoß geben, ähnlich gelagerte, jedoch weitaus wichtigere Aufgaben auf dieselbe Weise zu lösen. Probleme, wie sie bei der Erstellung von Schachprogrammen auftreten. In seinem Vortrag stellte Shannon viele der wichtigsten Ideen vor, die heute in allen Schachprogrammen enthalten sind:

- die interne Speicherung von Schachstellungen,
- die Baumsuche,
- die Bewertungsfunktion,
- die Zugauswahl mit Hilfe des Minimax-Verfahrens[1].

Shannon veröffentlichte seine Überlegungen in zwei Artikeln im Jahre 1950[107, 106], in diesen unterschied er zwischen einer *Typ A Strategie*, einer *Typ B Strategie* und einer *Typ C Strategie*.

Die *Typ A Strategie* bestand darin, dass alle legalen Fortsetzungen bis zu einer festgelegten Suchtiefe untersucht werden, sie entspricht also einem Brute Force Verfahren. Die

[1] Das Verfahren wird im Kapitel 2.2.4 genauer erklärt.

Typ B Strategie ist eine selektive Suche, die zweierlei beinhaltet, erstens die Aufnahme der *Ruhesuche* und die Vorauswahl sinnvoller Züge, die zur Untersuchung freigegeben werden. Die Ruhesuche ist einfach zu implementieren, es ist sogar so, dass praktisch kein Schachprogramm je geschrieben wurde, das ohne die Ruhesuche oder auch *statische Bewertung* auskam. Fast alle modernen Schachprogramme arbeiten nach diesem Prinzip. Aber auch mit dieser Verbesserung, so fürchtete Shannon, beruht der Erfolg auf reiner Rechengewalt, statt auf einer logischen Analyse der Situation. Die *Typ C Strategie* war nun die Modellierung des Schachwissens, dadurch sollte ein wesentlich effizienteres Schachprogramm entwickelt werden. Bislang gab es keinen durchschlagenden Erfolg bei der Entwicklung von Programmen, die nicht die Rechenleistung der Computer nutzten, sondern auf intelligentes Verhalten setzten.

Der erste Computer, der selbstständig Schach spielen konnte, war der MANIAC I, jedoch handelte es sich hier um eine vereinfachte Variante des Spiels. Es wurde nur auf einem 6x6 Feld gespielt und dazu die beiden Läufer und die entsprechenden Bauern weggelassen. Der Computer wurde 1950 in den Los Alamos National Laboratories gebaut. Es war ein röhrenbestückter Rechner, der über 10.000 Operationen pro Sekunde ausführen konnte. Trotz der Vereinfachung des Spiels und obwohl das Programm nur 4 Halbzüge berechnete, betrug die Rechenzeit des Programms zwölf Minuten pro Zug. Bei einem nicht eingeschränkten Schachbrett hätte die Maschine drei Stunden für jeden Zug benötigt. Das Programm wurde Mitte der Fünfziger Jahre fertig gestellt und spielte insgesamt drei Partien. Die erste gegen sich selbst (Weiß gewann), die zweite gegen einen Schachmeister, der dem Programm eine Dame vorgab, das Programm verlor nach 10 Stunden Rechenzeit. Die dritte Partie wurde 1956 gegen eine junge Dame des Forscherteams ausgetragen, sie hatte eine Woche vorher Schachspielen gelernt und für die Partie gegen den Computer trainiert. Das Programm gewann nach 23 Zügen, diese Partie war die erste in der Geschichte der KI, in der ein Mensch gegen eine Maschine verlor. In diesem Jahre prophezeiten auch Herbert Simon und Allen Newell, dass ein Computerprogramm in den nächsten 10 Jahren Weltmeister wird[2].

Das erste vollständige Schachprogramm wurde von Alex Bernstein im Jahre 1957 entwickelt, es lief auf einem IBM 704 Computer. Auch dieses Programm berechnete pro Zug 4 Halbzüge, es benötigte dafür 8 Minuten. Das Programm arbeitete nach den Prinzipien, die Shannon aufgestellt hatte, nach denen auch die Programme von Zuse und Turing funktionierten und nach denen auch noch die heutigen Programme arbeiten. Das Programm benutzte einen MiniMax-Algorithmus.

Das erste Programm, das an einem menschlichen Turnier teilnahm, war *MacHack VI*, das von Richard Greenblatt am MIT entwickelt wurde. Das Programm war eine Mischung aus Selektion und Brute Force. Im Januar 1967 spielte es in der Amateurmeisterschaft von Massachusetts und erzielte ein Remis aus fünf Partien, die restlichen vier Partien verlor das Programm. Dafür erhielt es ein USCF Rating von 1243, dies entspricht ungefähr 1040 Elopunkten. Im Laufe des Jahres verbesserte sich das Rating auf 1529 Punkte.

Obwohl nun 10 Jahre vergangen waren, hatte sich die Prophezeiung von H. Simon und A. Newell[3] nicht erfüllt, dass Schachprogramme in diesem Jahr Weltmeister würden. Das Niveau von MacHack war das eines Amateurs und nicht eines Weltmeisters. Zu dieser Zeit war auch der Streit über die Fähigkeiten von Computern in einer Hochphase, so kam es zu einer Wette zwischen David Levy und den Professoren John McCarthy und Donald Michie, beide Experten auf den Gebieten des Computerschachs und der Künstlichen Intelligenz. Levy, zu der Zeit schottischer Meister, bot ihnen daraufhin eine Wette

[2] Im gleichen Vortrag behaupteten sie, dass Computer zu diesem Zeitpunkt auch wertvolle Musikstücke komponieren, mathematische Sätze aufstellen und beweisen, sowie psychologische Theorien bestätigen oder widerlegen würden.

[3] H. Simon und A. Newell sind die Erfinder des Alpha-Beta-Verfahrens.

an, dass es innerhalb von zehn Jahren keinem Computer der Welt gelingen würde, ihn in einem Wettkampf zu besiegen. Insgesamt wurden 1250 Pfund oder 3000 Dollar gewettet. Neun Jahre lang hat er alle Rechner meist im Simultanspiel besiegen können. Im August 1978 kam es zum Abschlussmatch der Wette über sechs Partien gegen das beste Computerschachprogramm dieser Zeit, Chess 4.7. Gleich in der ersten Partie verlor Levy gegen *Chess 4.7*. Er merkte, dass er seinen Gegner unterschätzt hatte und setzte fortan eine andere Strategie ein: *Tue nichts, aber tue es gut (do nothing but do it well)*. Er baute geschlossene Stellungen auf, die keine überraschende Taktik gestatteten. Das Programm konnte bei dieser Strategie keinen Plan finden und zog ziellos hin und her. Währenddessen baute Levy die eigene Position aus und optimierte die Stellung seiner Figuren. Im richtigen Augenblick wagte er dann einen Angriff und bezwang seinen Gegner. Mit dieser Strategie besiegte Levy das Programm mühelos zweimal, anschließend versuchte er noch einmal die erste Strategie, was aber nicht funktionierte, so endete das Match mit 3,5:1,5 für Levy, womit er die Wette endgültig gewonnen hatte. Er verlängerte die Wette noch einmal um 5 Jahre, auch diese Wette gewann er, erst 1988 verlor Levy gegen DEEP THOUGHT.

Ende der siebziger Jahre stand die Schachprogrammierung vor einem Dilemma. Die besten Programme hatten das Niveau von Meisterspielern erreicht. Aber es war unklar, wie man eine weitere Steigerung erzielen konnte. Programme, die Schachwissen und intelligente Verfahren benutzten, konnten nicht mit mittelmäßigen Programmen konkurrieren, die eine Brute Force Methode benutzten. Einer der Hoffnungsträger war das Programm *Pionier* von dem Altweltmeister *Mikhail Botwinnik*. Die Entwicklung des Programmes begann im Jahre 1970, allerdings gab es bis auf wenige Ankündigungen keinen Beweis, wie gut das Programm spielte. Aber auch dem traditionellen Brute Force Ansatz stand kaum ein Weg offen die Spielstärke zu steigern, da diese durch die Geschwindigkeit der Rechner bestimmt wurde. Die stärksten Programme konnten einen Suchbaum bis zur Suchtiefe 7 berechnen, sollte der Suchbaum um einen weiteren Halbzug erweitert werden, benötigte man dafür die fünffache Rechenleistung. Jedoch ging man davon aus, dass ein Programm mindestens zehn Halbzüge analysieren müsste, bevor das Niveau von Großmeistern erreicht werden könnte. Dies könnte aber nur durch eine 250-fache Steigerung der Rechenleistung gelingen. Dies war aber in nächster Zeit nicht möglich, selbst wenn man das Moore'sche Gesetz bemüht, hätte man noch 12 Jahre warten müssen, somit begann die Zeit der Spezialmaschinen, die mit dem Rechner *Deep Blue* den Höhepunkt und auch ihr Ende erreichte. Er war es auch, der 1997 Kasparow in einem Match mit 3,5:2,5 besiegte und somit den Weltmeister ablöste, 30 Jahre später als prophezeit.

2.2 Computerschach Theorie

Viele Aktivitäten des Menschen, wie Spiele spielen, Mathematik, Auto fahren, benötigen *Intelligenz*. Wenn Computerprogramme diese Aufgaben lösen, spricht man von *Künstlicher Intelligenz*. *Problem Solving* oder auch *Problemlösen* als solches ist ein sehr vages Gebiet, welches aber ein zentrales Thema in der Künstlichen Intelligenz ist. Aber im weiteren Sinne gehören zum Thema *Problemlösung* alle Aufgaben die von Programmen gelöst werden, denn jede Aufgabe kann als ein Problem verstanden werden, welches gelöst werden soll [88].

Als typisches Beispiel für Probleme werden Spiele verwendet, denn sehr viele System, die Problemlösungsverfahren verwendet haben, wurden auf Spielen wie Dame, Schach, Mühle, Go aber auch Tic-Tac-Toe getestet. Der Grund dafür ist nicht, dass Spiele gegenüber Problemen der wirklichen Welt so einfach sind, sondern wie Minsky[4] sagte :„It is not

[4]in [81] Seite 12.

Halbzug	**Anzahl möglicher Stellungen**	
1	40	
2	1600	
3	64.000	
4	2.6 Millionen	
5	102 Millionen	
10	10 Billiarden	
11	419 Billiarden	← Alter des Universums in Sekunden
14	$2,7 * 10^{22}$	← Anzahl der Sterne im Universum
49	$3,2 * 10^{78}$	
		← Elementarteilchen im Universum
50	$1,3 * 10^{80}$	

Tabelle 2.1: Exponentielle Entwicklung der Stellungen beim Schachspiel. Die Annahme ist, dass nach jedem Zug 40 Züge möglich sind.

that the games and mathematical problems are chosen because they are clear and simple; rather it is that they give us, for the smallest initial structure, the greatest complexity, so that on can engage some really formidable situations after a relatively minimal diversion into programming .“

So wurde das Schachspiel zu einem Benchmarkproblem für die Aufgabe, die sich die KI gestellt hat. Die Geschichte des Computerschachs ist aber eine Geschichte von Irrtümern und Fehleinschätzungen der Möglichkeiten von Computerprogrammen. Obwohl Schach ein ideales Problem für Computer ist, die Regeln des Spiels sind eindeutig, das Schachuniversum ist klein; 64 Felder, 6 verschiedene Figuren und zwei Spieler. Aber selbst diese beschränkte Anzahl von Elementen führt schon zu einer sehr hohen Anzahl von Varianten, die während des Spiels entstehen können. Die Tabelle 2.1 gibt an, wieviele Stellungen nach welchem Halbzug möglich sind, so kann man sehen, dass die Anzahl möglicher Stellungen schon nach dem 11.ten Halbzug das Alter des Universums in Sekunden erreicht hat. Dies verdeutlicht auch das Problem der Computerprogramme. Wie soll ein *guter Zug* aus der Menge aller möglichen Züge gewählt werden? Wie bei vielen Problemen schlägt auch hier die Kombinatorik zu und verhindert eine einfache Lösung.

Die erste Frage muss daher lauten: Kann ein Computer überhaupt Schach spielen? Die einfachste Methode Schach zu spielen ist, es alle möglichen Züge zu erzeugen und dann zu überprüfen, welche dieser Züge legal sind. Danach wird einer der Züge gewählt. Dann befinden wir uns in der ersten Zeile der Tabelle 2.1 und man hat somit die Problematik mit dem großen Suchraum vermieden. Solche Programme werden aber im allgemeinen nach 10 Halbzügen geschlagen, was keine gute Leistung darstellt.

Da man also ein Computerprogramm schreiben kann, welches Schach spielen kann, ändert sich die Frage dahingehend, ob man ein Programm schreiben kann, was *gut* Schach spielen kann. Theoretisch kann man ein Programm schreiben, das alle möglichen Varianten von der Startstellung ausgehend berechnet, bis es das Ende des Spiels erreicht und eine *optimale Strategie* findet. Das heißt, egal welcher Zug vom Gegner gewählt wird, es ist immer möglich eine Sequenz von Zügen (Variante) zu finden, mit der das Programm gewinnt. Dieses Programm müsste den gesamten Suchbaum nur einmal aufbauen und speichern, anschließend müsste der Suchbaum nur für jede Stellung analysiert werden und dieses Programm würde *perfekt* Schach spielen. Diese Verfahren sind auch Probleme wie Tic-Tac-Toe möglich, nicht aber für Schach, denn schon nach 50 Halbzügen müssten $1,3 * 10^{80}$ Stellungen überprüft werden. Die Suche nach dem optimalen Zug unter der astronomischen Anzahl wird durch die Begrenzung der Züge, also der Reduzierung des Suchraums, durchgeführt.

Im Jahr 1950 gab Claude Shannon eine Antwort auf diese Frage und stellte drei Strategien vor, die die Größe des Suchraums reduzierten. Die *Typ A Strategie* bestand darin, dass alle legalen Fortsetzungen bis zu einer festgelegten Suchtiefe untersucht werden, es entspricht also einem Brute Force Verfahren. Die *Typ B Strategie* ist eine selektive Suche, die eine Vorauswahl sinnvoller Züge durchführt, die zur Untersuchung freigegeben werden. Schließlich kommt es noch zu einer statischen Bewertung der Stellungen. Fast alle modernen Schachprogramme arbeiten nach diesem Prinzip. Die *Typ C Strategie* war nun die Modellierung des Schachwissens, dadurch sollte ein wesentlich effizienteres Schachprogramm entwickelt werden. Bislang gab es keinen durchschlagenden Erfolg bei der Entwicklung von Programmen die nicht die Rechenleistung der Computer nutzten, sondern auf intelligentes Verhalten setzten.

Nimmt man nun die Typ A und B Strategien von Shannon, ergeben sich daraus folgende Anforderungen an ein Schachprogramm: Das Programm muss die Möglichkeit haben eine Stellung zu bewerten, die nach einer Reihe von Zügen erreicht wurde. Es muss die beste Sequenz von Zügen aus allen Zugfolgen ermittelt werden können. Ein Zuggenerator muss existieren, der alle möglichen Züge für eine Stellung berechnen kann und eine Bewertung der Züge muss möglich sein, damit gute Züge ausgewählt werden können. Da das Programm nicht bis zu einer festen Tiefe alle Züge berechnen soll sondern selektiv vorgehen soll, muss es die Möglichkeit haben seine Teile des Suchraums zu durchsuchen und andere Teile zu ignorieren.

Die *Spieltheorie*, von Morgenstern und von Neumann entwickelt, zeigt die Möglichkeit auf die beste Sequenz von Zügen aus allen möglichen Zügen zu finden. Denn ein Spiel kann als ein Baum aus Spielpositionen betrachtet werden, in dem die Knoten die Spielpositionen sind und die Kanten die erlaubten Züge zwischen den Stellungen. Mit dem *Minimax Verfahren* aus der Spieltheorie kann man so die besten Zugfolgen ermitteln (siehe dazu Kapitel 1.1 und 2.2.4). Aber selbst mit *sehr schnellen* Computern kann man mit dieser Methode keine großen Tiefen erreichen.

Im folgenden wird nun erklärt, welche Algorithmen und Datenstrukturen genutzt werden und welche Teilprobleme gelöst werden, damit ein Computer Schach effizient spielen kann.

2.2.1 Materialbewertungen

Der *Materialbewertung* kommt im Computerschach eine zentrale Rolle zu, denn die Suche nach einem guten Zug kann nicht bis zum Gewinn oder Verlust der Partie berechnet werden. Daher wird die Suche in einer Situation unterbrochen, in welcher der Ausgang der Partie noch offen ist. Schon Shannon [107] hat eine Bewertungfunktion postuliert, die die folgenden vier Punkte umfasst:

1. Die relativen Werte sollten für die Königin 9, den Turm 5, den Läufer 3, den Springer 3 und den Bauern 1 betragen. Diese Werte müssen für beide Seiten aufaddiert werden und die Seite mit dem höheren Wert hat die bessere Position.

2. Türme sollten auf offenen Linien stehen. Denn die Seite mit der höheren Mobilität hat eine bessere Stellung, wenn sonst alles gleich ist.

3. Doppelbauern, isolierte und rückwärtige Bauern sind schlecht.

4. Ein exponierter König ist schlecht, außer im Endspiel.

Dieser Aufbau für eine Bewertungfunktion hat noch heute seine Gültigkeit und neue Funktionen sind prinzipielle Erweiterungen zu der Bewertungsfunktion von Shannon. Diese Erweiterungen entsprechen den Punkten 2-4 von Shannons Funktion. Heutzutage versucht man weitere Faktoren einzubeziehen, welche die Aktivität oder die Stellung einzelner Figuren oder Gruppen mit berücksichtigen. Viele dieser Elemente findet man in dem Buch *The System* [14] von Hans Berliner , der als Schachspieler im ersten Teil des Buches verschiedene zusätzliche Merkmale beschreibt, die für eine Stellungsbewertung wichtig sind. So werden in neueren Bewertungen auch die Beherrschung des Brettzentrums mit einbezogen, da von dort aus alle anderen Positionen leichter erreichbar sind und dort platzierte, starke Figuren den größtmöglichen Einfluss haben. Aber auch der Mobilität der verschiedenen Figuren wird Rechnung getragen.

Alle Bewertungsfunktionen haben eine große Zahl von Parametern gemeinsam, diese werden im allgemeinen durch die Entwickler bestimmt. Es stellt sich aber die Frage, ob diese Werte auch die optimalen Werte sind oder ob man mit anderen Werten bessere Ergebnisse erzielen kann. Daher wurden verschiedenste Verfahren der CI auf die Problematik der Stellungsbewertung angewendet, um sie zu verbessern.

Genetische Algorithmen wurden von verschiedenen Autoren verwendet [123, 121, 82], dabei wurden Gewichtsvektoren evolviert, die die Parameter für eine Brettbewertung optimieren sollten. Obwohl die Brettbewertung in jeder Untersuchung für verschiedene Spielabschnitte eingesetzt wurde, war die Fitnessfunktion immer als Suche nach guten Zügen für bekannte Stellungen definiert.

Fuzzy Logik wurde auch zur Verbesserung der Stellungsbewertung genutzt, wozu teilweise auch mehrere Fuzzywerte für verschiedene Elemente der Bewertung genutzt werden (siehe [57, 58]).

Neuronale Netze wurden nicht nur für die Bewertung genutzt, aber auch hierzu wurden verschiedene Untersuchungen durchgeführt. So lernten Beal und Smith mit einem Netz die Figurwerte [9], die Werte der Figuren wurden dafür normalisiert, so dass der Bauer einen Wert von 1 hatte, der Läufer von $2,219$, der Springer von $3,114$, der Turm von $4,680$ und die Königin von $9,154$. Trotz der relativ geringen Änderungen zu den Werten von Shannon konnten Programme mit diesen Werten 57% der Spiele gewinnen, dabei wurden 2000 Spielen durchgeführt.

Auch die Evolutionären Verfahren wurden verwendet um die Materialwerte zu bestimmen. Die Individuen bestehen aus den einzelnen Werten für die Funktion der Stellungsbewertung. Das Verfahren evolviert zu dem gegebenen Algorithmus die Materialwerte der Figuren oder positionelle Kriterien (siehe [43, 70]).

Mysliwietz nutzt das *Simulated Annealing (SA)* als Vergleichsverfahren zum GA um die Materialvektoren anzupassen [82]. Er erreicht mit dem SA bessere Ergebnisse als mit dem GA.

Auch die Genetische Programmierung (GP) wurde für die Materialbewertung genutzt, dabei war es in dem System von ChessGP [22] nur ein GP Modul unter anderen, die zusammen Schach spielen konnten, siehe Abschnitt 5.1.1.

2.2.2 Der Zuggenerator

Der *Zuggenerator* ist ein Modul eines Schachprogramms, das nur selten erwähnt wird. Seine Aufgabe ist es, *legale Züge* für eine Stellung zu erzeugen, das heißt, der Generator muss die Regeln des Schachspiels kennen und anwenden. Von diesen Regeln kann auch nicht abgewichen werden, denn sonst würde der Generator keine legalen Züge ermitteln, daher gibt es keinen Raum für Optimierungen innerhalb des Generators und so wird in allen Systemen vorausgesetzt, dass es einen Zuggenerator gibt und dass er korrekt arbeitet. Neben den Regeln über die Gangarten der Figuren muss der Generator aus allen möglichen Zügen noch die legalen extrahieren. Dies geschieht normalerweise durch Beantwortung folgender Fragen:

- Ist das jeweilige Zielfeld noch auf dem Schachbrett?
- Ist das Zielfeld leer oder besetzt? Wenn es besetzt ist, ist es eine gegnerische Figur oder eine eigene?
- Wird der eigene König nach dem Zug angegriffen?
- Ist ein En-passant Zug für einen Bauern möglich?
- Ist eine Rochade möglich?

2.2.3 Zugbewertung

Die *Zugbewertung* ist nicht so klar definiert wie der Zuggenerator. Die Frage, die von dem Algorithmus beantwortet werden muss, ist, ob ein Zug *gut* oder *schlecht* ist. Aber dafür gibt es im Schach keine einfache Formel, nicht einmal eine zuverlässige Strategie[5]. Die Zugbewertung ist besonders für Alpha-Beta und ähnliche Verfahren sinnvoll, denn sie basieren auf einer Suchfenster-Technik, bei der stets der Wert der besten Alternativen des weißen und schwarzen Spielers im Suchfenster (α, β) gespeichert wird. Je schmaler das Fenster ist, desto größer ist die Chance von Schnitten im Suchbaum. Eine gute Vorsortierung der möglichen Züge kann somit zu einer Steigerung der Suchleistung führen, da der Algorithmus früher die Suche in schlechten Stellungen beenden kann([51, 108, 104]).

Die Zugbewertung ist ein schwieriges Problem, das aber eine große Bedeutung für die Güte von Schachprogrammen hat. So werden in vielen Programmen einfache Heuristiken oder eine Kombination aus Heuristiken verwendet um eine Sortierung durchzuführen. Folgende Heuristiken werden in Schachprogrammen genutzt:

Spielphase Als Merkmal wird dabei eine Kombination aus der Spielphase und dem Figurtyp genutzt. Hierbei werden Regeln dafür aufgestellt, wie wichtig verschiedene Figuren in verschiedenen Phasen des Spiels sind.

Figurwert Der Materialwert der ziehenden Figur wird als Kriterium für die Sortierung genutzt.

MVV/LVA *(Most Valuable Victim / Least Valuable Aggressor)* ist der Quotient der geschlagenen und der schlagenden Figur. Je höher der Wert der geschlagenen Figur und je geringer der Wert der schlagenden ist, desto höher ist das Merkmal ausgeprägt. Ist der Zug aber kein Schlagzug, ist das Merkmal 0.

Zentrumsaktivität Wenn die Zielfelder eines Zuges im Zentrum des Schachbrettes liegen, wird ein Bonus vergeben.

Killermoves *Killermoves* sind Züge, die in früheren Situationen zu einem Schnitt im Suchbaum geführt haben. Wird so ein Zug an einer anderen Stelle im Suchbaum wieder verwendet, wird er mit einem Bonus versehen.

[5] Für weniger komplizierte Spiele wie NIM sind solche Formeln bekannt.

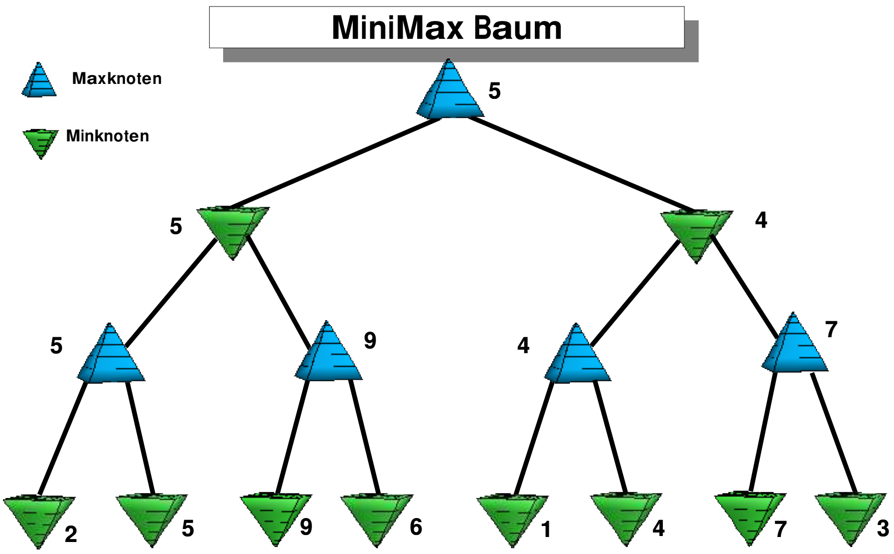

Abbildung 2.1: Die Abbildung zeigt einen Spielbaum mit den Bewertungen an den Knoten, wie sie vom Minimax Algorithmus bei bekannter Blattbewertung bestimmt werden. Die *Max-Knoten* im Baum werden durch blaue Pyramiden dargestellt und die *Min-Knoten* durch grüne Pyramiden, die auf dem Kopf stehen.

Iterative Deepening Der Spielbaum wird in Iterationen bis zur Tiefe n ausgewertet, die Informationen über die Bewertung werden in der Iteration $n+1$ zur Sortierung der Knoten genutzt (siehe auch 2.2.6.1).

Obwohl verschiedenste Heuristiken in Schachprogrammen verwendet werden, findet man nur sehr wenige Versuche, diese Probleme mit Methoden der CI zu lösen. In *Evolving Chess Playing Programs* [43] wurde für die Zugsortierung ein Evolutionärer Algorithmus verwendet, der für verschiedene Merkmale Werte evolviert, die anschließend zu einer Bewertung zusammengefasst werden.

2.2.4 Minimax / Negamax Verfahren

Die *Spieltheorie*, von Morgenstern und von Neumann entwickelt, zeigt eine Möglichkeit auf, die beste Sequenz von Zügen aus allen möglichen Zügen zu finden, indem ein Spiel als ein Baum aus Spielpositionen betrachtet wird. Die Knoten sind die Spielpositionen und die Kanten die erlaubten Züge zwischen den Stellungen. Eine optimale Strategie kann dann effizienter als durch eine vollständige Enumeration aller Strategien gefunden werden, indem man den Baum mittels des *Minimax Algorithmus* durchläuft.

Das *Minimax Verfahren* wurde von Shannon 1950 in seinem Artikel [107] beschrieben. Der Minimax Algorithmus baut einen Spielbaum bis zu einer festen Tiefe auf und ordnet jedem Blatt durch die Brettbewertung einen Wert zu. Der beste Zug wird dann unter der Annahme ermittelt, dass sowohl der Spieler als auch der Gegenspieler immer den besten Zug wählen. Die Abbildung 2.1 zeigt einen Spielbaum der Tiefe drei, mit den Werten für die Blattbewertung und die Bewertung der Inneren Knoten durch den Minimax Algorithmus.

Rekursiv lässt sich der Algorithmus für den *Minimaxwert* $v(k)$ für eine Blattbewertungsfunktion γ wie folgt aufschreiben:

$$v(k) = \begin{cases} \gamma(k) & \text{falls k ein Blatt ist} \\ \max(v(n)) & \text{falls k ein innerer Max-Knoten ist} \\ \min(v(n)) & \text{falls k ein innerer Min-Knoten ist} \end{cases} \tag{2.1}$$

Der Spieler, der am Zug ist, wählt immer den Zug, der seinen Gewinn maximiert, für den Gegner wählt er immer den Zug, der seinen Gewinn minimiert. Daher wird diese Strategie auch *Minimax* genannt. Welcher der Züge nun den Gewinn maximiert, kann exakt nur an den Blättern des Baumes bestimmt werden, wenn es zum Gewinn oder Verlust des Spiels gekommen ist. Dies ist aber nur bei einfachen Spielen möglich, wenn man einen kleinen Suchbaum hat. Für Schach ist es nicht möglich alle Stellungen zu berechnen (siehe Tabelle 2.1). Daher wird bei diesem Verfahren der Baum bis zu einer vorgegebenen Tiefe aufgespannt und die letzte Ebene als Blätter interpretiert und mit einer Stellungsbewertungsfunktion bewertet. Diese Werte werden dann in Richtung Wurzel propagiert.

Durch das exponentielle Wachstum des Spielbaums kann ein Minimax Verfahren nie sehr tiefe Bäume aufspannen und ist dadurch in der Spielstärke stark beschränkt. Denn die *optimale Strategie* kann nicht gefunden werden, da die Bewertung der *Blätter* des Minimax Baumes nur heuristisch sein kann. Für Spiele wie Tic-Tac-Toe und Vier-Gewinnt sind die Spielbäume für heutige Computer überschaubar und so kann die *optimale Strategie* durch den Spielbaum konstruiert werden.

Das *Negamax Verfahren* ist eine Variante des Minmax Verfahrens, dabei gilt, dass

$$min(a,b) = -max(-a,-b)$$

ist. Dadurch verändert sich die Definition der Minimaxwerte wie folgt:

$$v(k) = \begin{cases} \gamma(k) & \text{falls k ein Blatt auf der Max-Ebene ist} \\ \gamma(k) & \text{falls k ein Blatt auf der Min-Ebene ist} \\ -\max(v(n)) & \text{sonst} \end{cases} \tag{2.2}$$

Der Unterschied zum Minimax Algorithmus ist hauptsächlich in der Implementierung zu sehen, denn sie ist einfacher, da man nun nur eine Operation für die inneren Knoten hat, die nicht vom Typ abhängig ist. Die Effizienz des Verfahrens ist die gleiche und die Bewertung der Blätter ist gleich aufwendig. An den Blättern wird zusätzlich noch die Entscheidung getroffen, ob ein Min- oder Max-Blatt vorliegt, was aber einfach ist.

2.2.5 Alpha-Beta Verfahren

Das Alpha-Beta Verfahren wurde von Herbert Simon und Allen Newell[6] entwickelt.

Der Alpha-Beta-Algorithmus [104] ist eine Verbesserung des Minimax Verfahrens, denn bei der Anwendung des Minimax Verfahrens fällt auf, dass zum Teil Knoten des Baumes untersucht werden, die das Ergebnis, d. h. den Minimax-Wert an der Wurzel, gar nicht beeinflussen.

Die Alpha-Beta Suche basiert auf der Idee, nur die Werte zu berechnen, die untersucht werden müssen, aber ohne relevante Informationen zu verlieren. Dazu wird jedem *Max-Knoten* ein Wert α und jedem *Min-Knoten* ein Wert β zugewiesen. α stellt eine untere

[6]Sie sagten 1956 voraus, dass ein Computerprogramm in den nächsten 10 Jahren Weltmeister wird.

Halbzüge	Stellungen	$5 * \sqrt{n}$	St./Sek. bei 3 min/Zug
1	40	–	–
2	1600	200	1.1
3	64.000	1265	7
4	2.6 Millionen	8000	44
5	102 Millionen	50.000	281
6	4.1 Milliarden	320.000	1777
7	164 Milliarden	2 Millionen	11.000
8	6.5 Billionen	13 Millionen	70.000
9	262 Billionen	80 Millionen	450.000
10	10 Billiarden	512 Millionen	2.8 Millionen
11	419 Billiarden	3.2 Milliarden	17 Millionen
12	17 Trillionen	20 Milliarden	113 Millionen

Tabelle 2.2: Auswirkungen des Alpha-Beta Verfahrens auf die Größe des Suchbaumes.

Schranke, β eine obere Schranke für den Minimax-Wert des jeweiligen Knotens dar. Desweiteren unterscheidet man zwischen vorläufigen und endgültigen $\alpha-$ bzw. $\beta-$Werten. Ein vorläufiger $\alpha-$Wert für einen Max-Knoten (bzw. $\beta-$Wert für Min-Knoten) kennzeichnet eine Bewertung, die Max (bzw. Min) von diesem Knoten aus auf jeden Fall erreichen kann. Ein endgültiger $\alpha-$Wert (bzw. $\beta-$Wert) bezeichnet die Bewertung des für Max (bzw. Min) von diesem Knoten aus günstigsten Zuges. Analog zum Minimax-Verfahren versucht Max, den $\alpha-$Wert zu maximieren, während Min den $\beta-$Wert minimieren möchte. Daher wird α mit $-\infty$ initialisiert und wächst monoton, β mit $+\infty$ initialisiert und fällt monoton.

Der Algorithmus verfährt dann wie folgt: Der Baum wird mit Hilfe der Tiefensuche bis zu einer fest vorgegebenen Suchtiefe expandiert. Auf die Blattknoten wird die Bewertungsfunktion direkt angewendet. Dieser Wert entspricht nun dem $\alpha-$Wert des Knotens (bei Max-Knoten) bzw. dem $\beta-$Wert (bei Min-Knoten). Steht nun für einen Knoten der endgültige $\alpha-$Wert fest, so wird er als vorläufiger $\beta-$Wert an den unmittelbaren Vorgänger weitergereicht, wobei sich der neue $\beta-$Wert aus dem Minimum des neuen und alten Wertes ergibt. Analog wird der endgültige $\beta-$Wert als vorläufiger $\alpha-$Wert an den Vorgänger weitergereicht und der neue $\alpha-$Wert des Vorgängers aus dem Maximum des hochgereichten Wertes mit dem alten $\alpha-$Wert berechnet.

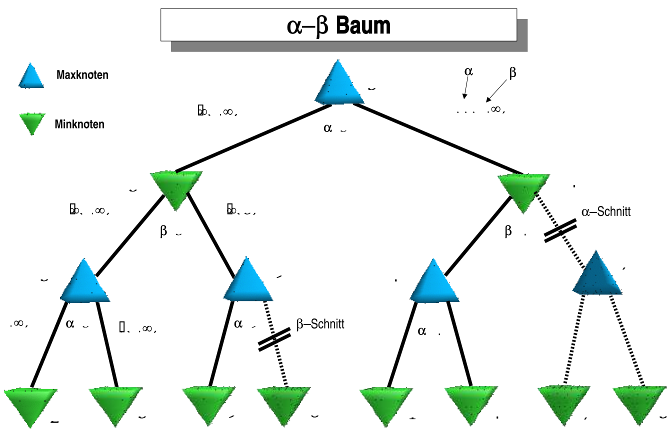

Abbildung 2.2: Der Spielbaum mit α, β Werten. Unter dem Knoten stehen immer die Werte des Knotens und an den Kanten die propagierten Werte.

Die für jeden Knoten mitgeführten $\alpha-$ bzw. $\beta-$Werte ermöglichen nun das *Abschneiden (cutoff)* von Teilbäumen. Der Spielbaum in der Abbildung 2.2 zeigt die $\alpha-$ und $\beta-$Schnitte. Der $\beta-$Schnitt für den Max-Knoten mit der Bewertung 9 ergibt sich, da der $\beta-$Wert für diesen Knoten bei 5 liegt und die Bewertung des Nachfolgers zum Wert 9 gelangte. Da der Vorgänger von 9 ein Min-Knoten ist, muss dieser Pfad nicht weiter verfolgt werden. So müssen alle weiteren Nachfolger nicht mehr betrachtet werden, man hat einen $\beta-$Schnitt durchgeführt. Der linke Sohn der Wurzel ist dann vollständig bewertet und hat einen Wert von 5, dieser wird als $\alpha-$Wert an die Wurzel weitergegeben, mit diesem $\alpha-$Wert wird dann der rechte Teilbaum bearbeitet. Dieser erhält den $\beta-$Wert, nachdem der linke Teilbaum bearbeitet wurde. Da im rechten Teilbaum ein Wert von 5 erreicht wurde, kann nun ein $\alpha-$Schnitt für die anderen Nachfolger durchgeführt werden. Die Bewertung der Wurzel hat einen Wert von 5 wie bei dem Minimax Verfahren, jedoch mussten nicht alle Knoten bearbeitet werden.

Die Einsparungen des Alpha-Beta Verfahrens hängen stark von der Reihenfolge ab, in der die Knoten bearbeitet werden. Die Möglichkeiten, dass Teile des Suchbaums abgeschnitten werden können, hängt davon ab, dass gute Züge früh gefunden werden. Die Abbildung 2.3 zeigt den Suchbaum mit den gleichen Werten, aber einer anderen Sortierung der Züge. In diesem Fall kann das Alpha-Beta Verfahren keinen Teilbaum abschneiden. Im worst case muss das Verfahren genauso viele Knoten expandieren wie das Minimax Verfahren. Im günstigsten Fall jedoch ist der zuerst gefundene vorläufige $\alpha-$Wert immer gleich der endgültige $\alpha-$Wert (analog für β). Dies führt dann zur größtmöglichen Anzahl von Schnitten. Es lässt sich zeigen, dass bei einem Baum der Tiefe T und des Verzweigungsfaktor V im best case

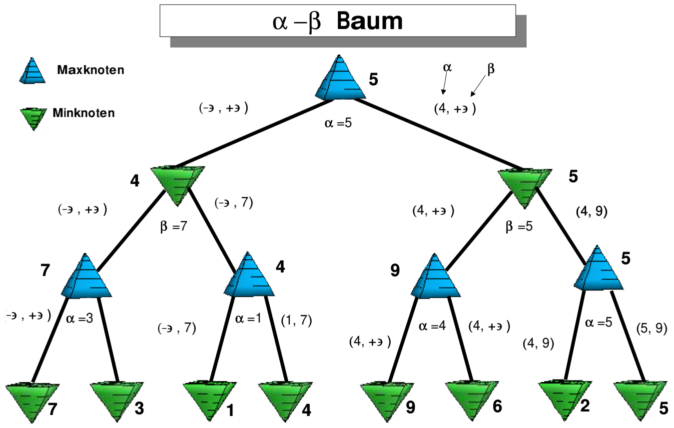

Abbildung 2.3: Der Spielbaum mit α, β Werten. Unter dem Knoten stehen immer die Werte des Knotens und an den Kanten die propagierten Werte.

$$n = \begin{cases} 2V^{T/2} - 1 & \text{für gerade } T \\ V^{(T+1)/2} - V^{(T-1)/2} - 1 & \text{für ungerade } T \end{cases} \tag{2.3}$$

von V^T möglichen Blattknoten zu analysieren sind. Moore und Knuth haben 1975 in ihrem Artikel bewiesen, dass im best case nur $V^{(T+1)/2} - V^{(T-1)/2} - 1$ Knoten im Baum expandiert werden müssen [72]. Damit kann man in perfekten Spielbäumen bei gleicher Knotenanzahl also doppelt so tief suchen wie mit dem Minimax Verfahren. Dies ist jedoch ein best case der selten bis nie erreicht wird, normalerweise werden $5 * \sqrt(V^T)$ Knoten expandiert, in der Tabelle 2.2 kann man die Entwicklung bezüglich der Suchtiefe für diesen Wert sehen.

2.2.6 Verbesserungen

Die Effizienz eines Suchverfahrens hängt im wesentlichen von der Anzahl der expandierten Knoten im Suchbaum ab. Die Zugreihenfolge ist hierbei ein sehr wichtiges Instrument zur Verbesserung der Alpha-Beta Algorithmen.

2.2.6.1 Iterative Deepening

Das *Iterative Deepening* ist eine Heuristik zur Vorsortierung von Zügen im Suchbaum [79]. Das Verfahren geht dabei so vor, dass der Spielbaum zunächst bis zu einer Tiefe n ausgewertet wird, danach werden die Informationen über die Knoten zur Sortierung der Knoten genutzt und in der nächsten Iteration wird der Spielbaum bis zur Tiefe $n+1$ ausgewertet. Da die Verfahren nicht zwingend den korrekten Minimaxwert der einzelnen Kinder an die Wurzel zurückliefern, sondern nur obere Schranken, ist die Sortierung der Knoten nicht ganz korrekt. In jeder Iteration werden bessere Werte für die Knoten ermittelt und somit verbessert sich auch die Sortierung. Der iterative Prozess wird beendet, wenn die maximale Suchtiefe erreicht wurde oder ein Zeitlimit erreicht ist. Das iterative Vorgehen ist möglich, da die Auswertung eines Spielbaumes bis zur Tiefe n billig im Vergleich zur Auswertung bis zur Tiefe $n+1$ ist. Weitere Ersparnisse kann man noch erreichen, wenn die Informationen aus früheren Auswertungen gespeichert werden und somit nicht mehrfach berechnet werden müssen.

2.2.6.2 Principal Variation Search

Die *Principal Variation Search* ist eine weitere Verbesserung des Alpha-Beta Verfahrens, die aber auf einer einfachen Idee beruht. Die Annahme ist, dass man eine gute Zugsortierungsfunktion hat. Auf dieser Annahme aufbauend, braucht man nur die *besten* Züge aus der Liste sehr genau zu untersuchen und die *schlechteren* Züge nicht so genau. Bei einem Alpha-Beta Verfahren wird das Suchfenster verkleinert, der α-Wert wird bei der weiteren Suche einfach erhöht. Wenn die Annahme richtig war, sind die besten Züge schon betrachtet worden und die weitere Suche wird schneller zu einem Näherungswert kommen, da das Verfahren mehr Bereiche des Baums abschneiden kann. Wenn es sich aber doch um einen guten Zug handelt, kommt es zu einem Fehler, da ein β-Schnitt nicht durchgeführt werden kann. Der Fehler ist ein zu hoher β-Wert und er zeigt an, dass der Zug den aktuellen α-Wert verbessern würde. Daher muss an dieser Stelle die Suche mit den Originalwerten wiederholt werden. Dies passiert, da man den α-Wert erhöht hatte und nun einen Zug gefunden hat, der ihn noch einmal verbessert.

2.2.6.3 Hashtabellen

Die *Hashtabellen* speichern Stellungen des Suchbaums mit deren Bewertung. Der Suchbaum im Schach ist eigentlich ein Graph und so kann eine Stellung durch verschiedene Zugfolgen erzeugt werden. So ist es möglich, dass eine Stellung einmal in 5 und auf einem anderen Weg erst in 7 oder 8 Zügen erreicht wird. Wenn das Ergebnis der Stellung nun gespeichert ist, braucht sie nicht noch einmal analysiert werden. Die Hashtabelle wird dabei während der Analyse der aktuellen Stellung aufgestellt. Diese Tabelle wird aber nicht nach dem Zug gelöscht, sondern im Spiel weiter verwendet. Bei der nächsten Stellung wird dann zuerst immer überprüft, ob die Stellung schon in der Tabelle vorhanden ist. Wenn nicht, wird die Suche normal weiter verfolgt. Falls die Stellung schon einmal analysiert wurde, wird noch überprüft, ob dies zum aktuellen Zeitpunkt noch ausreichend ist. Wenn der Eintrag als veraltet eingestuft wird, da die aktuelle Tiefe im Suchbaum ähnlich der Analysetiefe der Stellung in der Tabelle ist, wird die Stellung weiter analysiert und die Werte in der Hashtabelle mit den neuen Werten modifiziert. Normalerweise werden in einem Tabelleneintrag neben dem Hashschlüssel für die Stellung die Bewertung der Stellung, die Tiefe, bis zu der die Stellung bewertet wurde, und der *Typ* des Wertes gespeichert. Durch das Alpha-Beta Verfahren kann es auch dazu kommen, dass nicht die exakten Werte für eine Stellung berechnet werden, sondern nur obere und untere Schranken und diese Information wird in dem Typ Feld der Hashtabelle gespeichert. Die oberen oder unteren Schranke sind zwar nicht so gut wie die exakten Werte einer Stellung, wenn jedoch die obere Schranke für den Wert einer Stellung schlechter ist als der α-Wert, dann muss diese Stellung nicht weiter expandiert werden.

2.2.6.4 Killer Moves und History Heuristic

Die *Killer Moves* und die *History Heuristic* sind weitere Verfahren um die Zugsortierung zu verbessern.

Die *Killer Moves* Heuristik speichert Züge, die sich schon an anderen Stellen der Suche als gut herausgestellt haben. Es handelt sich dabei um Züge des Gegners, die sich als gut für den Gegner herausgestellt haben. Diese Züge haben zu β-Schnitten im Baum geführt. Sie werden dann in Listen gespeichert. Abhängig davon, wie oft sie gespielt wurden, werden sie, wenn sie in einer Zugliste wieder vorkommen, bei der Zugsortierung vorrangig beachtet. Hierbei ist zu beachten, dass immer nur Züge betrachtet werden, die auf der gleichen Ebene im Suchbaum liegen. Das heißt, ein Killer Move der Tiefe 6 wird zu Beginn der Suche nicht als solcher für Tiefe 7 genutzt, außer es stellt sich heraus, dass dieser Zug auch auf dieser Ebene ein Killer Move ist.

Die *History Heuristic* nutzt einen ähnlichen Ansatz wie die Killer Move Heuristik, der wichtige Unterschied ist aber, dass sie die guten Züge unabhängig von der Ebene im Suchbaum speichert. Die guten Züge werden gespeichert und wenn ein Zug im Verlauf des Spiels wieder verwendet wird, wird er bei der Zugsortierung wie im Falle der Killer Moves bevorzugt behandelt.

2.2.6.5 Ruhesuche

Die *Ruhesuche* soll dem Problem des *Horizonteffekts* Rechnung tragen. Der Horizonteffekt tritt immer bei Verfahren auf, die die Suche nach einer festen Tiefe abbrechen. In diesen Situationen kann es dazu kommen, dass das Programm glaubt, eine *gute* Position erreicht zu haben, aber wenn es noch einen Zug weiter suchen würde, würde es erkennen, dass es in einer *schlechten* Position ist. Um dieses Problem zu vermeiden, werden in dieser Situation alle Schlagzüge weiterverfolgt, bis man eine *ruhige Stellung* erreicht hat. Dies sind Stellungen, in denen kein Schlagzug durchgeführt werden kann. Dieses Vorgehen ermöglicht es dem Programm, auch lange Zugfolgen zu erkennen.

Diese Methode hat das Problem, dass der Suchbaum dadurch sehr groß werden kann und somit den Aufwand nicht mehr rechtfertigt. Es werden verschiedene Heuristiken angewendet, welche die Größe des Baums einschränken, um diese Problem zu verkleinern.

2.2.6.6 Null Move Heuristik

Ein *Null-Move* ist die Aufgabe des Zugrechts an den Mitspieler, es ist also kein legaler Zug. Normalerweise wäre es schlecht, von seinem Zugrecht keinen Gebrauch zu machen. Wenn man nun die Stellung analysiert, nachdem man auf sein Zugrecht verzichtet hat und der Gegner somit zwei Züge hintereinander ausgeführt hat und die eigene Stellung immer noch gut ist, dann handelt es sich um eine sehr gute Stellung. Wenn man dies nun auf das Alpha-Beta Verfahren anwendet, bedeutet es, dass, nachdem ein Null-Move ausgeführt wurde und die Suche für den Gegner einen Wert kleiner als α (oder für die Seite mit dem Zugrecht einen Wert größer als β) zurückgibt, dann braucht der Teilbaum nicht weiter untersucht zu werden, da die Stellung so stark ist, dass sie bei einer guten Strategie nie erreicht werden kann. Man erwartet also einen β-Schnitt an dieser Stelle ohne den gesamten Teilbaum zu expandieren. Dieses Verfahren wurde von Donninger 1993 entwickelt und eingesetzt [27] ein neueres Verfahren kann in [113] gefunden werden.

Die Heuristik hat neben den Vorteilen, dass der Baum stärker beschnitten wird und dass die Knoten im Suchbaum, an dem ein Null-Move eingesetzt wird, nur einen Ausgrad von eins haben, auch erhebliche Nachteile. Wenn diese Methode ohne Bedacht eingesetzt wird, kann es zu einer hohen Zahl von Fehlern kommen, besonders in Stellungen, in denen *Zugzwang* herrscht. Dies sind Stellungen, in denen die Person, die ziehen muss, verlieren wird.

Null-Moves sollten daher nicht durchgeführt werden, wenn folgende Bedingungen gelten:

- in Zugzwang Situation,
- die Seite, die zieht, nur wenige Figuren hat,
- die Seite, die zieht, nahe einem Schachmatt ist
- oder die Seite, die am Zug ist, im Schach steht.

Diese Liste kann und wird in verschiedenen Programmen noch durch andere Regeln erweitert, diese sind aber die wichtigsten.

Ein weiterer wichtiger Parameter ist noch, um wieviel die Tiefe nach einem Null-Move in dem Baum reduziert wird, diese wird normalerweise um eins bis drei reduziert. Jedoch hat Heinz et al gezeigt, dass eine adaptive Tiefenreduktion besser ist (siehe [50]).

2.2.6.7 Futility Pruning

Das *Futility Pruning* wurde erstmals in *Dark Thought* eingesetzt (siehe auch [46, 48, 49]). Das Konzept ist recht einfach und wird auf Knoten angewendet, die sich einen Halbzug vor der Ruhesuche im Suchbaum befinden. Für diese Knoten wird überprüft, ob

- die Seite, die zieht, nicht im Schach steht,
- der aktuelle Zug kein Schlagzug ist,
- der aktuelle Zug kein Schach verursacht,
- die Bewertung der aktuelle Stellung plus einen Schwellwert (normalerweise ein Bauernwert) den α-Wert nicht verbessert
- oder der aktuelle Zug schlecht ist.

Wenn eines dieser Punkte zutrifft, wird die Stellung direkt bewertet und die Suche abgebrochen. Die Grundidee der Heuristik ist also der Abbruch der Suche, wenn man den α-Wert nicht erhöhen kann.

Das Ziel dieser Heuristik ist es, die Geschwindigkeit der Suche zu erhöhen. Obwohl man durch dieses Vorgehen einige taktische Fehler in Kauf nimmt, kann man sie im allgemeinen mehr als ausgleichen, wenn man dadurch die Suche um einen Halbzug erhöhen kann.

Es gibt noch die Variante des *Extended Futility Pruning*, sie wird noch einen Halbzug vor den Knoten des Futility Pruning durchgeführt. Diese Knoten werden dann auch als *pre-frontier* Knoten bezeichnet. Diese Variante benutzt einen höheren Schwellwert für die Verbesserung des α-Wertes, dieser wird von einem Bauernwert auf einen Turmwert erhöht.

2.2.6.8 Search Extensions

Die *Search Extensions* Heuristik ist der Versuch, Züge, die *interessant* sind, genauer zu untersuchen. Normalerweise wird die Suche bei einer maximalen Tiefe abgebrochen, jedoch kann es sein, dass die Stellungsbewertung in dieser keinen Vorteil für den Zug sieht. Wenn die Suche an dieser Stelle etwas weitergehen würde, könnte man den Zug genauer beurteilen.

Genau dies wird bei dem Singular Extension Verfahren gemacht, es versucht gute Züge weiter zu analysieren, um eine bessere Bewertung für die Züge zu erhalten. Dafür werden diese Züge über die maximale Tiefe der Suche hinaus analysiert. Dieses Verfahren führt zu einer Steigerung der Spielstärke, kann aber auch zu sehr großen Suchbäumen führen, was die Vorteile wieder aufzehrt.

Es gibt verschiedene Varianten der Search Extensions, wobei es aber immer um eine Ausweitung der Suche geht. Folgende Search Extensions werden in verschiedenen Programmen verwendet:

- *Check Extensions*: Wenn die Seite, die am Zug ist, im Schach steht, wird die Suche etwas erweitert, bei einem Schachmatt wird sie drastisch erweitert.
- *Singular Extensions*: Wenn es nur einen legale Zug in einer Stellung gibt, wird die Suche fortgesetzt. Die Kosten sind minimal, da der Ausgrad des Knotens nur eins ist.
- *Threat Extensions*: Wenn es zu einer starken Bedrohung kommt, wird die Suche intensiviert, auch mit Hilfe von Null-Moves, um eine Lösung zu finden.
- *Pawn Push Extensions*: Wenn ein Bauer auf der siebten Reihe steht oder die Möglichkeit zur Umwandlung hat. Dieses Verfahren hilft besonders im Endspiel.
- Wenn die Sicherheit des Königs plötzlich stark sinkt,
- die Bewertung sich ohne den Gewinn oder Verlust einer Figur stark ändert,
- oder wenn die Verteidigung durch ein Opfer verschlechtert wird.

2.2.6.9 Datenbanken

Das Schachspiel gliedert sich grob in die Eröffnung, das Mittelspiel und das Endspiel, wann welche Phase endet und die nächste anfängt, ist nicht genau definiert. Jedoch haben die verschiedenen Phasen unterschiedliche Anforderungen an das Schachprogramm. Da ein Spiel immer mit der gleichen Position beginnt, konnten die Eröffnungen sehr genau untersucht werden. Für ein normales Schachprogramm mit seinem beschränkten Suchhorizont ist dies ein Problem. Dies liegt daran, dass es schwierig ist, positionelle Kriterien in die statische Stellungsbewertung einzufügen und zu Beginn der Partie hauptsächlich positionelle Kriterien die Güte der Stellung unterscheiden. Diese Unterschiede werden erst jenseits des Suchhorizontes eines Algorithmus zu einem Material unterschied der Figuren. Aus diesen Gründen wurden große *Eröffnungsdatenbanken* in Schachprogramme integriert, die Suche wird damit durch Wissen ergänzt oder ersetzt. Durch die Datenbank können Schachprogramme sehr viel stärker spielen, da sie dadurch eine gute Ausgangsposition für das Mittelspiel erreichen können.

Es gibt keine Datenbanken für das Mittelspiel, da die Menge der gespeicherten Stellungen zu groß wäre um einen Vorteil für das Programm zu erhalten. Für die Endspiele sieht dies jedoch anders aus, hier ist die Anzahl der Figuren so gering, dass man für das Endspiel alle Stellungen bis zum Ende der Partie abspeichern kann. Diese *Endspieldatenbanken* verbessern die Ergebnisse der Schachprogramme, da die Suchverfahren in Endspielsituationen große Schwierigkeiten haben. Dies liegt daran, dass in dieser Phase Strategien notwendig sind, die sich über 50 Halbzüge oder mehr erstrecken, um den Gegner matt zu setzen. Damit haben aber die Suchverfahren aufgrund des Horizonteffektes große Probleme. Ende des Jahres 2002 waren bereits alle Stellungen mit maximal 7 Figuren erfasst und analysiert, die Stellungen mit 8 Figuren sind derzeit in Arbeit, darunter sind Varianten, die erst nach über 200 Zügen zu einem Matt führen. Für den Endanwender sind derartige Forschungen jedoch eher uninteressant, da auch der benötigte Platz mit der Anzahl der möglichen Stellungen exponentiell zunimmt.

Wenn ein Schachprogramm zu Endspielen gelangt, die in seiner Endspieldatenbank gespeichert sind, beschränkt sich das Schachprogramm auf das Suchen der gegenwärtigen Stellung in einer Hashtabelle und das Ausführen des dort für diese Stellung hinterlegten optimalen Zuges. Aus diesem Grund werden in dieser Arbeit keine Schachprogramme mit Datenbanken evolviert, da dann die Güte des Programms stark von der Güte der Datenbank abhängt und vom eigentlichen Ziel der Entwicklung von Schachprogrammen ablenkt.

2.2.7 State-space Suchverfahren

Das *state-space Suchverfahren* wurde 1979 von George C. Stockman und unter dem Namen SSS* vorgestellt (siehe [110]). Der SSS* Algorithmus führt nicht wie das Alpha-Beta Verfahren eine Rückbewertung der Blattwerte zur Wurzel durch, sondern sucht nach der besten *Max-Strategie*. Der Spielbaum einer *Max-Strategie* unterscheidet sich von einem vollständigen Spielbaum dadurch, dass jeder Max-Knoten im Baum nur einen Nachfolger hat und alle Min-Knoten die gleichen Nachfolger haben wie im vollständigen Spielbaum. Roizen und Peal konnten in ihrem Artikel einige Jahre später zeigen, dass dieses Verfahren nicht mehr Knoten expandiert als ein Alpha-Beta Algorithmus (siehe [103]). Später wurd von Marsland et al. eine weitere Variante des Algorithmus namens DUAL* eingeführt. Obwohl beide Verfahren in empirischen Vergleichen deutlich weniger Knoten als ein Alpha-Beta Verfahren expandiert, wurde das Verfahren kaum verwendet, dies liegt an dem hohen Speicherplatzbedarf.

Um aber die beste Max-Strategie zu finden, werden partiell expandierte Max-Strategien in einer OPEN-Liste gespeichert. Der Speicherplatzbedarf der OPEN-Liste ist $\Theta(b^{\frac{d}{2}})$, wobei b der Verzweigungsfaktor und d die Suchtiefe ist, dadurch ist der Platzbedarf exponentiell und die Rechenzeit pro Knoten wächst mit der OPEN-Liste. Aus diesem Grund ist das Verfahren fünf- bis zehnmal langsamer als ein Alpha-Beta Verfahren.

In den neunziger Jahren folgten noch weitere Verbesserungen, der RecSSS* und der RecDual* Algorithmus von Reinefeld (siehe [101]). Diese Algorithmen waren aufgrund einer effizienten rekursiven Impelementierung hinsichtlich der Rechenzeit einem Alpha-Beta Verfahren leicht überlegen, jedoch blieb der exponentielle Speicherplatzbedarf bestehen.

Durch die Darstellung des SSS* Verfahrens als eine Serie von Nullfenster Alpha-Beta Aufrufen konnte eine weitere Variante entwickelt werden, der MTD(f) Algorithmus [98, 96].

Der MTD(f) Algorithmus kann als ein wettbewerbsfähiger Algorithmus betrachtet werden, so wird er auch in einem spielstarken Schachprogramm des MIT dem Clikchess, verwendet. Jedoch bleibt bei all diesen Verfahren das Problem des großen Speicherplatzbedarfs.

2.2.8 Nicht klassische Methoden

Neben den klassischen Methoden des Computerschachs, wie sie oben beschrieben wurden, gibt es auch Systeme, die Methoden des *maschinellen Lernens* oder der *computational intelligence* verwenden. Hierzu wurde 1986 ein erster Überblick von Skiena im ICCA Journal veröffentlicht, dieser war eher pessimistisch gestimmt, da diese Verfahren mit den klassischen Verfahren nicht konkurrieren konnten. Zehn Jahre später wurde von Fürnkranz [37] ein weiterer Artikel geschrieben, der die Ergebnisse der vergangenen zehn Jahre auf diesem Gebiet zusammenfasst.

Eine Methode sind die *induktiven Lernalgorithmen*, sie klassifizieren Stellungen eines Schachspiels in die Klassen *gewonnen* und *verloren*. Die Versuche mit diesem Verfahren wurden für Endspiele im Schach verwendet, hierzu soll der Algorithmus aus einer Menge Stellungen eine allgemeine Regel zur Klassifizierung von Endspielen ableiten.

2.2.8.1 Wissensbasierte Schachprogramme

Systeme, die auf erklärungsbasiertem Lernen basieren, versuchen aus vollständigen und korrektem Wissen Pläne zu konstruieren, die eine globale Erklärung für die zu lösende Aufgabe sind. Da das Ziel aber oft nur durch komplexe oder sehr viele Regeln beschrieben werden kann, muss der Erklärungsprozess oft mit Vereinfachungen und Approximationen arbeiten. Diese können aber zu Fehlern in den erzeugten Plänen führen. Wissensbasierte Systeme sollten daher auch mit unscharfem Wissen arbeiten können, um der Komplexität der Wissensbasis entgegenzuwirken. (siehe [52, 109]).

Das System von Tadepalli nutzt eine erklärungsbasierte Methode, er nennt sie *lazy explanation-based Learning, LEBL* [114]. Das System untersucht verschiedene Züge für beide Spieler mit Hilfe von *O-Plänen*, bestehend aus einem Ziel und einer Abfolge von Zügen. O-Pläne sind optimistisch, da für alle Züge angenommen wird, dass die gegnerischen Züge entweder den Erfolg des O-Plans nicht beeinflussen oder selbst Teil des O-Plans sind. Da bei der Erstellung und Analyse des Suchbaums nur auf die Möglichkeiten zurückgegriffen wird, die durch die O-Pläne vorgegeben werden, ist der Suchbaum sehr eingeschränkt. Um mehrere gegnerische Züge in Erwägung zu ziehen, werden die O-Pläne zu komplizierteren Suchbäumen, den *C-Plänen*, verknüpft.

LEBL erzeugt C-Pläne für jeden Spieler und testet sie gegeneinander, indem es den Suchbaum erweitert und neu bewertet. Wenn ein C-Plan nicht mehr durch den C-Plan des anderen Spielers geschlagen werden kann, wird die Konstruktion der Pläne abgebrochen.

LEBL arbeitet umso besser, je mehr es gelernt hat, dies liegt daran, dass es weniger Fehler macht, wenn es erschöpfender sucht. Das System sucht viel weniger Knoten als eine Alpha-Beta Suche ab und berücksichtigt auch Strategien des Gegenspielers, jedoch bleibt die Leistungsfähigkeit des Systems hinter der von Alpha-Beta Suchen zurück.

2.2.8.2 Musterbasierte Schachprogramme

Robert Levinson entwickelte ein selbstlernendes, musterbasiertes Schachprogramm [77, 78]. Das mehr auf "zusammenhängenden Urteilen" (associative-reasoning) basiert als auf einer Exploration des Suchbaums. Diese Muster sollten durch das Programm alleine entdeckt und erlernt werden. Das Schachwissen des Systems beschränkt sich auf die Spielregeln und eine Graphrepräsentation für Brettpositionen. Eine Schachposition wird durch einen gerichteten bewerteten Graphen dargestellt, wobei die Knoten die Spielfiguren und die unbesetzten Felder um die Könige darstellen. Die Kanten zeigen die Angriffs- und Verteidigungsbeziehungen zwischen den Figuren. Die Graphen werden bezüglich einer

Halbordnung gespeichert, die durch eine Relation *ist Teilgraph von* gebildet wird. Diese Relation ist gleichbedeutend mit der Aussage, dass ein Graph *allgemeiner* als der andere ist.

Als Lernverfahren wird eine Temporal Difference Methode (TD) genutzt, die durch ein dynamisches boolesches Merkmal erweitert wurde. Das boolesche Merkmal gibt die An- oder Abwesenheit eines gegebenen Musters für die aktuelle Position an. Die TD-Methode wird eingesetzt, um das Gewicht eines neuen Musters zu bestimmen.

Wenn ein neues Muster gespeichert wird, sucht das Programm zuerst in den bereits vorhandenen Mustern das dazu ähnlichste Muster und fügt einen gemeinsamen Teilgraph als neues Muster ein. Zum Schluss wird dem Muster ein Gewicht zugewiesen, das dem Durchschnittsgewicht seiner Eltern entspricht. Ziel ist es, durch eine gut gewählte Menge von Mustern einen ganzen Unterbaum der Suche abschneiden zu können.

Bei Experimenten mit dem Programm hatte es keine Probleme, Schach-Prinzipien wie *Schlagen einer höherwertigen Figur* oder *Schlagen ungedeckter Figuren* und andere zu erlernen. Das System erreichte 2001 bei einem Test im Internet Chess Club (ICC) einen Elowert von 1024, dies entspricht dem Niveau eines Anfängers.

2.2.8.3 Neuronales Schach

Neuronale Netze wurden auch verwendet um Schach zu spielen. Eines dieser Systeme ist das *NeuroChess System* von Sebastian Thrun [120]. Der zentrale Lernmechanismus ist ein erklärungsbasiertes neuronales Netz. Thrun konstruierte im NeuroChess Netz eine Bewertungsfunktion mit Hilfe der *Temporal Difference* Methode [112]. Die Aufgabe der TD(0) Methode, eine Variante der TD-Methode, ist eine Bewertungsfunktion zu finden, die Schachstellungen bezüglich ihrer Güte ordnet.

Da bei Schach jedoch eine rein induktive Lerntechnik wegen der Trainingszeiten praktisch nicht anwendbar war, verwendete Thrun erklärungsbasierte Methoden, die mit weniger Trainingsdaten Spielsituationen verallgemeinern können. Das erkärungsbasierte Neuronale Netz benutzt Wissen über das Schachspiel, welches durch ein gesondertes Netz repräsentiert wird. Dieses Netz besteht aus 175 Eingabeneuronen, 165 verborgenen und 175 Ausgabeneuronen, es wurde mittels einer Datenbank mit 120000 Großmeister-Schachpartien trainiert. Danach wird das NeuroChess System trainiert, welches die Bewertungsfunktion lernen soll. Diese Netz besteht aus 175 Eingabeneuronen, 80 verborgenen Neuronen und einem Ausgabeneuron.

Um nun die Stärke von NeuroChess zu messen wurden verschiedene Spiele gegen ein Schachprogramm (GNU-Chess) durchgeführt. Dabei ist festzuhalten, dass NeuroChess nur die Stellungsbewertung durchführt. Um ein Schachspiel durchzuführen, wird diese Bewertung in einen Suchalgorithmus eingebunden.

Ein weiteres System, welches Neuronale Netze nutzt um Schach zu lernen ist SAL, von Michael Gherrity in seiner Dissertation 1993 vorgestellt [40].

SAL kann jegliche Zweipersonen-Brettspiele mit perfekter Information lernen, die ein quadratisches Brett und Spielfiguren nutzen. Hierzu muss SAL ein Zuggenerator für das Spiel zur Verfügung gestellt werden, die Größe des Brettes, die Anzahl und die Typen der Figuren. SAL berechnet einen Zug dann mit Hilfe der Alpha-Beta Suche, verknüpft mit einer *Konsistenzsuche* (consistency search). Die Konsistenzsuche geht davon aus, dass nur ein Kind eines inkonsistenten Knotens im Suchbaum nicht korrekt berechnet wurde. Die Suche expandiert dann die Knoten weiter, bis es den fehlerhaften Knoten finden und ersetzen kann.

Als Bewertungsfunktion für die Knoten nutzt SAL ein Neuronales Netz. Die Eingabe des Netzes sind die Position der Figuren, die Anzahl der Figuren für jeden Typ, die Figur, die zuletzt bewegt wurde, die Figur, die geschlagen wurde und eine weitere Anzahl von Ad Hoc Annahmen über Angriff, Verteidigung und den Raum. Das Netz wird wie oft im Umfeld von Spielen, mit der *Temporal Difference* Methode trainiert.

SAL wurde mit drei Spielen getestet, dem Tic-Tac-Toe, connect-four und Schach. Tic-Tac-Toe konnte SAL nach einer Lernphase von 20.000 Spielen perfekt spielen, d.h. SAL hatte die optimale Strategie gefunden.

Das Spiel connect-four ist komplizierter und SAL brauchte eine Lernphase von 100.000 Spielen, um gegen ein Programm 80 % der Spiele zu gewinnen. Jedoch war nicht klar, wie gut das Programm war, somit ist die Leistung nicht genau einzuordnen.

Bei dem Test für das Schachspiel hat das SAL-System in der Lernphase 4.200 Spiele gegen GnuChess durchgeführt. GnuChess wurde dafür ohne Eröffnungs- und Endspieldatenbank gestartet und das Programm durfte eine Sekunde pro Zug rechnen, damit erreicht GnuChess einen Elowert von 1.000-1.200, was dem Niveau eines Anfängers entspricht. SAL konnte acht Spiele gegen GnuChess zu unentschieden spielen, alle anderen hat es verloren. Die Lernphase musste nach 4.200 Spielen beendet werden, da die Rechenzeit zu hoch war. Jedoch war zu diesem Zeitpunkt die Lernphase noch nicht beendet und SAL verbesserte sich zu diesem Zeitpunkt noch, was anhand der Lernkurven zu erkennen ist.

2.3 Das Elozahlen-System

Als Maß für die Stärke eines Schachspielers werden die *Elozahlen* benutzt. Diese wurden von *Arpad E. Elo*, einem amerikanischen Physiker, entwickelt. Elo wandte die Methoden der Statistik und der Wahrscheinlichkeitstheorie auf die Leistungen von Schachspielern an. Er entwickelte ein zuverlässiges Wertungssystem, das ein Maß für die relativen Spielstärken der einzelnen Spieler definiert. Im Elo-System erhält jeder Spieler eine Zahl, die sich aus einer mathematischen Auswertung der Ergebnisse seiner Tunierpartien ergibt. Hat er in einem Turnier viele Partien gewonnen, erhöht sich seine Wertung und zwar um so mehr, je stärker seine Gegner waren. Umgekehrt werden ihm Wertungspunkte abgezogen, wenn er bei einem Turnier verliert. Eine Einführung in die mathematischen Grundlagen des Elo-Systems findet man in [17].

Da ein starker Spieler nicht immer einen schwächeren Spieler besiegt, wird eine Normalverteilungsfunktion für die Performance der Spieler angenommen. Die Abweichung wird als Standardabweichung (σ) angegeben. Auf dieser Grundlage wurde eine Wahrscheinlichkeitsfunktion entwickelt, die bei bekannter Elodifferenz zwischen zwei Spielern die Gewinnwahrscheinlichkeit des besseren Spielers angibt. Die Funktion $P(D)$ ist wie folgt definiert:

$$P(D) = \frac{1}{2} \int_0^D e^{-\frac{1}{2}t^2} dt,$$

- $P(D)$: die Gewinnwahrscheinlichkeit,
- D: die Elodifferenz zwischen zwei Spielern,
- σ: die Standardabweichung.

Die Werte der Funktion $P(D)$ können auch durch die folgende Funktion approximiert werden:

$$P'(D) = \frac{1}{1 + 10^{-\frac{D}{400}}}.$$

Elo-Differenz	Gewinnwahr.	Elo-Differenz	Gewinnwahr.
0	50%	140	76%
20	53%	160	79%
40	58%	180	82%
60	62%	200	84%
80	66%	300	93%
100	69%	400	97%
120	73%	900	99%

Tabelle 2.3: Gewinnwahrscheinlichkeit in Abhängigkeit von der Elodifferenz

Die Tabelle 2.3 zeigt auf der Grundlage der Gewinnwahrscheinlichkeitsfunktion $P(D)$, wie sich die Gewinnwahrscheinlichkeit (in Prozent angegeben) bei verschiedenen Differenzen ändert. Man kann deutlich sehen, dass man ab einer Elodifferenz von 400 Punkten davon ausgehen kann, dass der bessere Spieler immer gewinnt. Jedoch selbst bei Elodifferenzen von 100 Punkten hat der schwächere Spieler immer noch gute Gewinnchancen.

Die Gewinnwahrscheinlichkeitsfunktion gibt aber keine Auskunft über die Bewertung der einzelnen Spieler untereinander.

Die erste Funktion im Elo-System ist die *Performance Rating Formel* oder auch *Periodic Rating Formel*, sie stellt die Bewertung für ein kurzes Intervall dar oder ist eine provisorische Bewertung für Spieler, die wenige Spiele gegen bewertete Spieler gemacht haben. Die Funktion ist wie folgt definiert:

$$R_p = R_c + D(P),$$

- R_p: Bewertung des Spielers,
- R_c: Bewertung des Gegners,
- $D(P)$: Bewertungsdifferenz bezogen auf die Funktion $P(D)$.

Die *Rating Funktion* wird benutzt um die neue Bewertung auf einer kontinuierlichen Basis zu berechnen. Diese Funktion fügt die neue Bewertung in die alte Bewertung ein und führt somit zu einem geglätteten Verlauf der Bewertung.

$$R_n = R_o + K * (W - W_e)$$

- R_n: die neue Bewertung nach dem Spiel,
- R_o: die alte Bewertung,
- K: Bewertungspunktzahl für ein Spiel $K \in [10, \dots 32]$
- W: Spielausgang (1 für Gewinn, 0.5 für Remis und 0 für Verlust),
- W_e: Erwarteter Spielausgang auf der Basis von R_o.

Elo-Wert	**Spielklasse**
2600+	Großmeister (GM)
2400+	Internationale Meister (IM)
2200+	Nationale Meister (FM)
2000+	Expertenlevel
1800+	Vereinsspieler Klasse A
1600+	Vereinsspieler Klasse B
1400+	Vereinsspieler Klasse C
1200+	Vereinsspieler Klasse D
1200-	Anfänger

Tabelle 2.4: Einteilung der Elowerte nach Spielklassen.

Das Wichtige am Elo-System ist, dass sich auf diese Weise Spieler miteinander vergleichen lassen, die nie gegeneinander angetreten sind. Die Tabelle 2.4 zeigt dabei, mit welcher Spielklasse welche Elozahl in Verbindung gesetzt wird.

Kapitel 3

GP Einführung

Die *Genetische Programmierung* ist eine Suchheuristik, die den Raum der *Programme* nach Lösungen durchsucht. Dabei macht sie sich das Prinzip der Evolution zunutze, um ein definiertes Ziel zu erreichen.

Das Prinzip Darwins, die natürliche Selektion, wird als Motor der Evolution aller Lebensformen auf der Erde betrachtet. Das Prinzip der Evolution wird schon über 30 Jahre in verschiedenen Bereichen der Informatik eingesetzt, wodurch sich verschiedene Forschungszweige wie die Evolutionsstrategien [105] , die Genetischen Algorithmen [53] und die Evolutionäre Programmierung [34] entwickelt haben. Diese Methoden wurden in den letzten Jahren erfolgreich auf zahlreiche theoretische und praktische Probleme angewendet.

Die Genetische Programmierung ist das neueste Forschungsgebiet, das die Evolution zur Erzeugung von Programmen nutzt. Im Unterschied zu anderen evolutionären Methoden ist der Suchraum der *GP nicht* wie oft ein Vektor von reellen, natürlichen oder binären Werten, sondern der Raum der Programme. GP-Systeme optimieren während eines evolutionären Prozesses eine Population von *symbolisch* repräsentierten Programmen mittels *Genetischer Operatoren*. Der Ablauf eines GP-Systems zur Optimierung der Individuen innerhalb der Population folgt dabei in einer Evolutionsschleife, wie sie in der Abbildung 3.1 dargestellt ist. Die Evolutionsschleife wird dabei solange ausgeführt, bis ein Abbruchkriterium erfüllt ist.

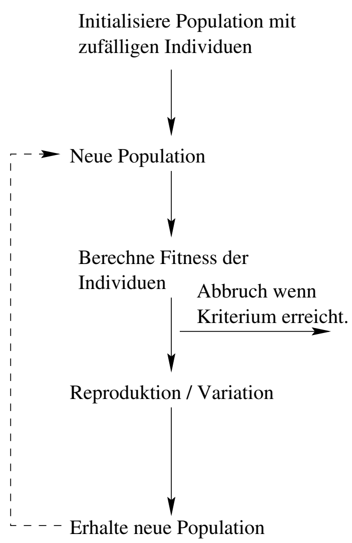

Abbildung 3.1: Evolutionsschleife

Die Methode der Genetischen Programmierung geht auf Forschungen Ende der 50ziger Jahre zurück [36], die jedoch keinen großen Erfolg hatten. Selbst als die ersten positiven Ergebnisse erzielt wurden, konnte sich die Methode der GP nicht etablieren [24]. Erst durch Koza [73], der die Individuen als Baumstrukturen repräsentierte, konnte sich die Methode der Genetischen Programmierung in der Forschung behaupten. Seitdem haben sich im Bereich der GP weitere Repräsentationsformen der Individuen mit ihren eigenen Besonderheiten gebildet, wie lineare Individuen im CGPS [89, 91, 90] und graphbasierte Individuen im PADO-System [116].

Der Grundgedanke der Genetischen Programmierung ist es, den Individuenpool von einem zufälligen Initialzustand durch einen Selektionsprozess, der die *guten* Individuen bevorzugt, zu einem Zustand höherer Ordnung zu führen. Während dieses Prozesses steigt im System die Komplexität an, solange kein Gleichgewichtszustand erreicht wird. Der Anstieg der Komplexität bezieht sich sowohl auf die Struktur der Individuen im Pool als auch auf das Ein- und Ausgabeverhalten der Individuen. Die Selektion und Rekombination ändert das Ein- und Ausgabeverhalten von einer zufälligen Abbildungsvorschrift zu einer Abbildung, die das gestellte Problem löst oder annähert.
Hier wird der Begriff der *Komplexität* in einem mehr intuitiven Sinne benutzt, als ein Zustand zwischen Ordnung und Chaos, periodisch und zufällig, und nicht im Sinne der *algorithmischen Komplexität*. Das Verhältnis zwischen Komplexität und Ordnung kann man sich vereinfacht, wie in der Abbildung 3.2 dargestellt, vorstellen. Damit liegt der Punkt der höchsten Komplexität zwischen Ordnung und Unordnung.

Somit hat ein Individuum, welches zu jeder Eingabe die gleiche oder eine zufällige Ausgabe liefert, eine kleinere Komplexität als ein Individuum, das zu einer Eingabe eine dem Problem entsprechende Ausgabe liefert.
Der Initalpool, der durch einen zufälligen Prozess entsteht, hat somit eine geringere Komplexität als der Pool einer späteren Generation, denn durch den Rekombinationsprozess haben sich Individuen gebildet, die einander ähnlich, jedoch nicht identisch sind.

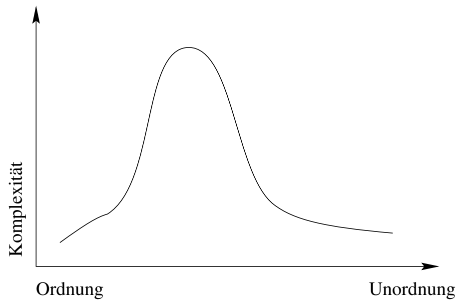

Abbildung 3.2: Komplexität liegt zwischen Ordnung und Unordnung.

Betrachtet man Komplexität als den durch Charles Bennet [12] definierten Prozess der *logischen Tiefe*, wonach Komplexität nicht von der nominalen Information, sondern von dem zugrundeliegenden Prozess der Beseitigung von (überflüssiger) Information ist, so ist sie ein Ausdruck für den Prozess, der zu einer bestimmten Menge von Information führt, die sich auf das gestellte Problem bezieht. Diese Information erhält man durch einen Prozess, der aus der Menge aller Informationen des Problems die Informationen löscht, die für die Lösung überflüssig sind.

Man kann zum Beispiel die Abbildung $f(x) = x^2$ durch eine unendliche Menge von Zahlenpaaren $\ldots, (-2, 4), (-1, 1), (0, 0), (1, 1), (2, 4), \ldots$ und die Abbildungsvorschrift selbst beschreiben, jedoch ist die Menge der Zahlenpaare für eine vollständige Beschreibung der Abbildung überflüssig. So hat nach Bennet die Abbildungsvorschrift eine höhere logische Tiefe als die Menge der Zahlenpaare.

Im Falle der Genetischen Programmierung geschieht dies auf zwei Ebenen, auf der Ebene der Individuen in der Population und auf der Ebene des Abbildungsverhaltens.

Auf der ersten Ebene werden, ausgehend von einer endlichen (aber großen) Menge aller Individuenkonfigurationen, die Konfigurationen durch den Evolutionsprozess erzeugt, die das Problem lösen können. Von der großen Informationsmenge wird eine große Anzahl von Individuen (Informationen) ausgesondert, also wird nach der Definition von Bennet in der Population *logische Tiefe* erzeugt.

Das Gleiche geschieht auf der Ebene der Abbildungsvorschriften, aus allen Abbildungsvorschriften, die die Individuen erzeugen könnten, werden durch den evolutionären Prozess die ausgewählt, die das Problem lösen.

Die Individuen der Genetischen Programmierung sind das Resultat der Simulation einer evolutionären Berechnung, vergleichbar mit biologischen Wesen, die ebenfalls das Resultat einer sehr langen evolutionären Berechnung sind.

Da aus dem Suchraum aller Individuen die Individuen ausgesondert wurden, die die Aufgabe nicht lösen, erhält man einen informationslöschenden Prozess. *GP* erfüllt damit, ebenso wie die natürliche Evolution, die Definition der logischen Tiefe.

Genetisches Programmieren ist folglich ein Prozess, der Ordnung, Komplexität und somit logische Tiefe schafft.

Viele Begriffe der *GP*, die eine zentrale Bedeutung haben, sind nicht allgemein bekannt und überschneiden sich teilweise mit Begriffen aus anderen Forschungsbereichen, wie denen der Evolutionsstrategien (**ES**) [105]. Um Fehlinterpretationen zu vermeiden, werden die Begriffe *Genetische Operatoren, Fitnessfunktionen, Fitnesscases und Selektion* im folgenden für diese Arbeit kurz genauer beschrieben.

3.1 Strukturen von GP Individuen

Die Genetischen Programme bestehen aus Operationen, die selber Funktionen, Prozeduren, Konstanten oder Terminale sein können, die in einer vorgegebenen Struktur zusammengefasst werden. Diese *Struktur* bestimmt den Aufbau, die Programmausführung und die Genetischen Operatoren. Die Standardvarianten in GP-Systemen sind

- (1) Baum GP [73, 74],
- (2) Lineare GP [89, 7] und
- (3) Graph GP [5, 99, 117].

3.1.1 Baum GP

Das *Baum GP* zeichnet sich dadurch aus, dass die Operationen in einer Baumstruktur eingebettet sind, wobei die Kinder eines Knotens immer die Operanden einer Operation enthalten. Die Blätter des Baumes enthalten dann die Terminale oder Konstanten des Individuums. Neben der Form ist aber auch die Auswertungsreihenfolge des Baumes wichtig. Die Konvention zur Auswertung von Bäumen ist, dass sie von den Blättern zur Wurzel hin ausgewertet werden. Hierzu wird oft ein Tiefendurchlauf durch den Baum gewählt um die Reihenfolge der Auswertungen zu bestimmen.

Die Abbildung 3.3 zeigt den Aufbau eines Baum Individuums graphisch, wichtige Knoten sind farblich markiert. Die inneren Knoten des Individuums enthalten die Operationen und die Blätter die Terminale und Konstanten.

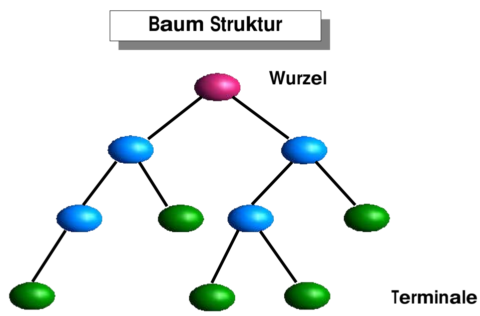

Abbildung 3.3: Aufbau eines baumbasierten Individuums. Die inneren Knoten des Individuums enthalten die Operationen und die Blätter die Terminale und Konstanten.

Die Kanten des Baumes stellen den Datenfluss zwischen den Operationen dar. Somit brauchen Baumstrukturen keinen externen Speicher, damit die einzelnen Operationen miteinander kommunizieren können. Dies ist ein Unterschied zu den meisten anderen Strukturen, die immer einen externen Speicher zur Kommunikation benutzen.

3.1.2 Lineares GP

Das *lineare GP* ordnet die Operationen in einer linearen Sequenz an. Die Operationen werden der Reihe nach von vorne nach hinten ausgewertet. Die Anordnung der Operationen stellt hier den Programmfluss dar, der Datenfluss zwischen den Operationen wird über externe Speicher durchgeführt. Daher sind lineare GP Individuen im allgemeinen Zwei- oder Dreiadress Registermaschinen [89, 54]. Der externe Speicher linearer Individuen wird deshalb auch als Register bezeichnet. Ein Instruktion eines linearen Individuums hat die Form:

$$Reg(1) = op(Reg(2), x), \text{mit } x \in \mathcal{R}, \mathcal{K}, \mathcal{T}.$$

Wobei $\mathcal{R}$ die Menge der Register, $\mathcal{K}$ die Menge der Konstanten und $\mathcal{T}$ die Menge der Terminale ist. Durch diesen Aufbau einer Instruktion wird sichergestellt, dass Ergebnisse aus früheren Instruktionen weiter verarbeitet und somit der Datenfluss zwischen den Instruktionen erzwungen wird.

Eine Besonderheit des linearen GP ist noch das Maschinensprache GP, der Aufbau der Individuen ist identisch, der Unterschied liegt in der Repräsentation, mit der die Individuen im Speicher dargestellt werden. Normalerweise ist dies eine Datenstruktur, die durch ein Programm interpretiert wird. Beim Maschinensprache GP liegen die Individuen aber direkt in Maschinensprache vor und können daher sofort von der CPU verarbeitet werden. Dies hat zur Folge, dass dieses Verfahren sehr effizient und schnell ist.

3.1.3 Graph GP

Das *Graph GP* nimmt unter den Strukturen eine Sonderrolle ein, da die Struktur nicht so eindeutig definiert ist wie die beiden anderen Strukturen und unter dem gleichen Namen verschieden aufgebaute Individuen verstanden werden.

Ein Vertreter des Graph GP's ist das *PADO-System* von Teller und Veloso [117, 65, 64]. Die Operationen werden in dieser Struktur in einem gerichteten Graph organisiert.

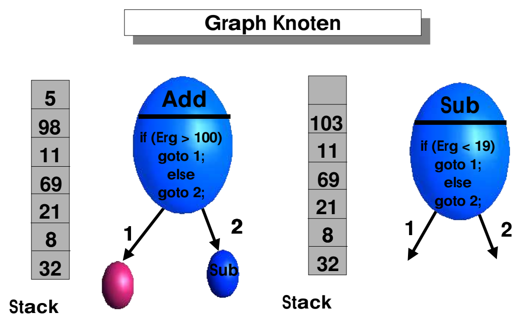

Abbildung 3.4: Aufbau eines PADO Knotens. Man kann deutlich die Zweiteilung des Knotens in den Funktions- und Branchingteil erkennen. Gleichzeitig wird der Datenfluss dargestellt, der über den Stack läuft.

Die Kanten zwischen den Knoten stellen in dieser Struktur den potentiellen Programmfluss des Individuums dar. Der Datenfluss verläuft in dieser Struktur über einen Stack, von dem die Operationen ihre Eingaben nehmen und ihre Ausgaben schreiben. Zusätzlich gibt es noch einen externen Speicher, in dem Zwischenergebnisse länger gespeichert werden können. Die Abbildung 3.4 verdeutlicht den Aufbau eines Knotens und dessen Interpretation. Die Funktion des Knotens nimmt ihre

Parameter vom Stack und schreibt das Ergebnis anschließend wieder auf den Stack. Danach wird die Branching Funktion des Knotens ausgewertet, hierzu wird mit Hilfe des vorher berechneten Ergebnisses die Kante zum Nachfolgerknoten gewählt. Über diese Funktionen wird der Programmfluss gesteuert.

Der Aufbau dieses Individuums verhindert weder Zyklen noch Rekursionen, so dass Programme entstehen können, die nicht terminieren. Um dies zu verhindern erhält jedes Programm eine Laufzeitbeschränkung. Diese bewirkt, dass ein Programm nach einer maximalen Laufzeit beendet wird. Im optimalen Fall sollten die Programme jedoch mit dem *Stopknoten* beendet werden. Die Interpretation des Individuums beginnt immer mit dem *Startknoten*.

Ein anderer Vertreter des Graph GP benutzt die Graphstruktur nicht zur Kontrolle des Programmflusses, sondern für den Datenfluss, wie es beim Baum GP benutzt wird (siehe [5, 99, 86, 87]). Bei dieser Struktur muss man jedoch darauf achten, dass es nicht zu Kreisen und Rekursionen kommt, da derartige Programme sonst nicht terminieren würden. Dies hat zur Folge, dass bei diesen Strukturen meistens auf Crossover verzichtet wird (siehe hierzu 3.2.1). Eine Ausnahme stellt das GGP System von Jens Niehaus dar, welches ein Crossoverfahren für ein Graph GP System entwickelt hat, in dem die Kanten des Graphen den Datenfluss darstellen [85]. Die erhöhte Komplexität verhindert die Verbreitung dieser Strukturen, da mit Baum und Linearem GP auch gute Ergebnisse erzielt werden.

3.2 Genetische Operatoren

Die Initialpopulation hat im allgemeinen eine schlechte Fitness, die durch den evolutionären Prozess verbessert wird. Die Verbesserung der Fitness während des evolutionären Prozesses wird mittels *genetischer Operatoren* [7] durchgeführt. Die *genetischen Operatoren* manipulieren dabei einzelne Individuen innerhalb der Population. Prinzipiell unterscheidet man *Rekombinations-*, *Mutations-* und *Replikations*-Operatoren. Der *Replikations-Operator* ist der einfachste der drei Operatoren. Er erstellt eine einfache Kopie von einem Individuum.

3.2.1 Rekombination

Die *Rekombination* ist ein Verfahren, bei dem aus mindestens zwei Individuen neue Individuen erzeugt werden, indem es zwischen den Elternindividuen *Strukturelemente* austauscht. Eine Variante der Rekombination ist der *Crossover*-Operator.

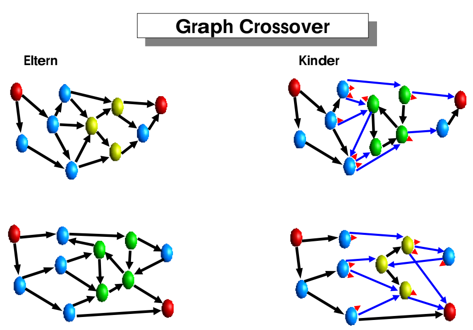

Abbildung 3.5: Graphbasiertes Crossover.

Grundsätzlich unterscheidet sich dabei das Vorgehen des Crossover-Operators bei den verschiedenen Strukturen, die die Individuen aufweisen.

Bei baumbasierten Individuen wählt der Crossover-Operator aus beiden Eltern je einen Unterbaum zufällig aus. Die ausgewählten Unterbäume werden dann zwischen den beiden Individuen ausgetauscht [73, 74].

Abbildung 3.6 skizziert dieses Vorgehen, wobei die mit Eltern bezeichneten Bäume die Ausgangsbäume und die mit Kinder bezeichneten Bäume die durch das Crossover entstandenen Bäume sind.

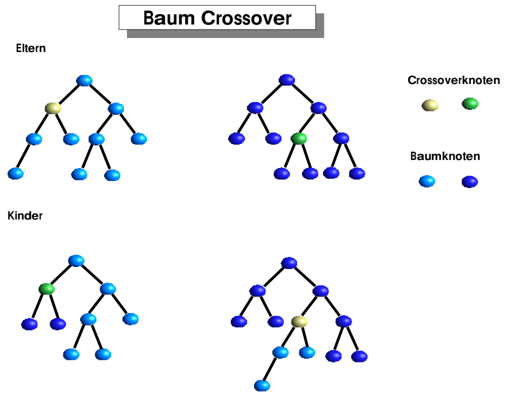

Abbildung 3.6: Baumbasiertes Crossover.

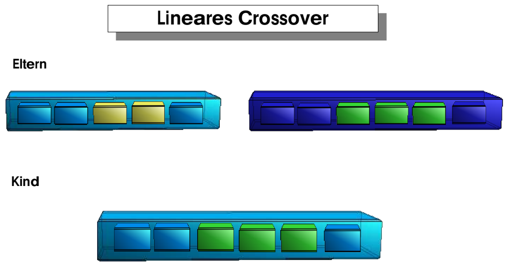

Abbildung 3.7: Linearbasiertes Zwei-Punkte-Crossover. In jedem der Individuen wird jeweils ein Segment von Knoten ausgetauscht.

Beim Crossover linearer Individuen wird zufällig ein Punkt in beiden Eltern bestimmt, von dem aus eine lineare Sequenz von Befehlen ausgetauscht wird. Dabei müssen weder die relative Lage der Punkte noch die Längen der Sequenzen übereinstimmen [89, 90, 92]. In der Abbildung 3.7 wird graphisch das Vorgehen des Crossover-Operators dargestellt.

Der Crossover-Operator des graphbasierten GP tauscht prinzipiell zwischen den beiden Elterngraphen eine zufällige Menge von Knoten aus. Die in der Abbildung 3.5 farblich markierten Knoten sind die Knoten, die während des Crossovers zwischen den Eltern ausgetauscht werden. Die roten Kanten in den neuen Graphen sind die Kanten, die durch das Crossover durchtrennt wurden. Sie werden in beiden Teilgraphen anschließend durch zufällig gewählte Kanten verbunden [115, 118].

Jedoch müssen beim Crossover zwischen zwei Graphen noch weitere Punkte beachtet werden, zum Beispiel, wie die neuen Kanten gesetzt werden. Wenn man beim Graph GP die Variante nutzt, die den Datenfluss über die Kanten befördert, so muss man darauf achten, dass die Teilgraphen die gleiche Anzahl von eingehenden und ausgehenden Kanten haben. Dies führt dazu, dass man ein exponentielles Laufzeitverhalten erhält, wenn man einen einfachen Algorithmus für dieses Problem nutzt. Daher werden in den meisten Systemen Heuristiken verwendt um die richtigen Teilgraphen zu finden, siehe hierzu [85].

3.2.2 Mutation

Die *Mutation* ist eine *zufällige*, aber strukturell erlaubte Veränderung des genetischen Codes eines Individuums. Der genetische Code kann dabei an einer Stelle oder an mehreren Stellen des Individuums geändert werden. Die Funktionsweise des Mutations-Operators hängt von der Struktur der Individuen ab.

Bei der Mutation eines Baumes wird zufällig ein Knoten gewählt, dann wird entweder die Operation in diesem Knoten ausgetauscht oder der ganze Unterbaum an diesem Knoten wird durch einen zufällig erzeugten Unterbaum ersetzt.

Bei der Mutation einer linearen Struktur wählt der Operator eine Instruktion im Individuum aus. Diese Instruktion wird mutiert, indem entweder die Instruktion durch eine andere ersetzt wird oder die Operanden der Instruktion variiert werden. Auch hier kann entweder nur eine Instruktion oder eine Folge von Instruktionen, die von einem bestimmten Punkt des Individuums ausgeht mutiert werden.

Bei graphbasierten Strukturen ergeben sich noch weitere Möglichkeiten für den Mutations-Operator. Grundsätzlich wird auch hier an einem oder mehreren Knoten der Inhalt verändert, jedoch ist der Aufbau eines Knotens in einem Graphen anders. So kann nicht nur der Knoteninhalt verändert werden, sondern auch die Kanten eines Graphen (siehe [118]).

3.3 Selektion und Fitness

GP führt keine vollständige Suche im Suchraum durch. Vielmehr bietet die GP-Population die Möglichkeit einer parallelen Suche, die durch die Rekombination und Mutation von *guten* Individuen einen Weg zu einer guten (optimalen) Lösung im Suchraum zeigt.

Um diese *besseren* Individuen zu finden, muss dem GP System eine Güte für die einzelnen Individuen zur Verfügung gestellt werden, mit der es möglich ist diese Individuen im Pool zu finden, damit sie anschließend von den *genetischen Operatoren* zur Reproduktion herangezogen werden können. Die Güte wird in GP-Systemen im allgemeinen durch eine *fitnessbasierte Selektion* erzeugt, die aus zwei Bestandteilen besteht, einer *Fitnessfunktion* und einer *Selektionsmethode*.

3.3.1 Die Fitnessfunktion

Die *Fitnessfunktion* bildet die einzelnen Individuen auf einen Wert ab, der als *Fitness* bezeichnet wird. Die *Fitness* bestimmt die Güte des Individuums bezüglich des zu lösenden Problems. Damit ordnet die Fitness jedem Individuum einen Rang innerhalb der Population zu.

In GP wird die Fitnessfunktion oft bezüglich einer *Trainingsmenge* berechnet, die aus *Fitnesscases* besteht. Ein Fitnesscase c ist ein Tupel $c = (e, a)$ von einem Eingabedatum $e \in \mathcal{E}$ und dem zum Eingabedatum gehörenden Ausgabedatum $a \in \mathcal{A}$. Eine Teilmenge aller möglichen Fitnesscases wird als Trainingsmenge bezeichnet.

3.3.2 Selektionsmethoden

Die Aufgabe der Selektion ist es, nachdem die Qualität der Individuen durch die Berechnung ihrer Fitness bestimmt wurde, die Individuen zu selektieren, die den genetischen Operatoren zur Verfügung gestellt werden und die Individuen in der Population zu bestimmen, die ersetzt werden sollen.

In der Literatur findet man verschiedene Selektionsmethoden, denn die Selektion selbst ist ein sehr entscheidender Punkt innerhalb der evolutionären Berechnung und somit ist die Auswahl der richtigen Methode eine wichtige Entscheidung. Die verschiedenen Methoden haben unterschiedliche Auswirkungen auf die Geschwindigkeit und den Erfolg des evolutionären Prozesses.

Die hier vorgestellten Selektionsmethoden sind ein kurzer Überblick über die bekannten Methoden.

Die proportionale Selektion wird in **GA**-Systemen als allgemeiner Selektionsmechanismus verwendet [53]. Die proportionale Selektion ordnet jedem Individuum einen Wahrscheinlichkeitswert p_i zu, der die Chance des Individuums angibt in die nächste Generation zu gelangen.

Die Truncation-Selektionsmethode kommt aus dem Bereich der ES, in der sie als (μ ,λ) Selektion [105] bekannt ist. Bei dieser Methode werden durch Rekombination der μ Eltern λ Nachkommen erzeugt, aus denen die besten μ Individuen als Eltern der nächsten Generation ausgewählt werden.

Eine Variante dieser Methode ist die ($\mu + \lambda$) Selektion, hierbei werden die Eltern in den Selektionsprozess mit einbezogen. Wenn alle Kinder schlechtere Fitnesswerte aufweisen als ihre Eltern, wird die nächste Generation wieder aus denselben μ Eltern gebildet.

Die Rangselektion basiert auf der Eigenschaft der Fitnesswerte, den Individuen eines Pools durch Sortieren einen Rang zuzuordnen [7]. Die Selektionswahrscheinlichkeit eines Individuums wird dann als eine Funktion aus dem Rang des Individuums innerhalb der Population bestimmt.

Die Turnierselektion [7] basiert auf der Idee des Wettkampfes einer Teilmenge von Individuen innerhalb der Population. Eine Anzahl von Individuen, die durch eine Turniergröße bestimmt werden, wird zufällig ausgewählt, die Hälfte der Individuen mit den besseren Fitnesswerten in der Turniermenge ersetzen anschließend die schlechteren durch Reproduktion (Überschreiben).

Kapitel 4

Erweiterung durch neue Individuentypen

Das Ziel von GeneticChess ist das Schachspiel, also muss es dem GP-System möglich sein, Algorithmen zu entwickeln, die dieses komplexe Problem lösen. Im Abschnitt 2.2 wurden die verschiedenen Algorithmen vorgestellt, mit deren Hilfe der Computer Schach spielen kann. Diese Algorithmen sind die Kernalgorithmen für die Berechnung des nächsten Zuges, also der Suchalgorithmus ohne Erweiterungen durch Datenbanken, wie sie heute in Schachprogrammen üblich sind. Wenn man diese Algorithmen aber analysiert, sieht man sehr schnell, dass es sich um rekursive Algorithmen handelt, deren Programmfluss stark von der Berechnung von Zwischenergebnissen abhängt. Vergleicht man dies mit dem Verhalten von GP-Programmen, sieht man sofort den Unterschied zu den bekannten Individuenstrukturen Baum GP [73, 74], Linear GP [89, 7] und Graph GP [5, 99]. Nur durch Funktionen, die Konditionale oder Sprünge enthalten, kommt es zu einem begrenzten Einfluss auf den Programmfluss. Keine dieser Strukturen ruft sich rekursiv auf. Aus diesem Grund ist eine Neuentwicklung von Individuentypen für ein GP-System notwendig, die Suchprobleme lösen sollen. Daher wurden Individuentypen entwickelt, die eine stärkere Flexibilität im Programmfluss haben und ein Individuentyp, dem es möglich ist Probleme, rekursiv zu behandeln.

Ein weiteres Hindernis bei der Entwicklung von Schachprogrammen mit evolutionären Systemen ist einerseits die Komplexität eines Algorithmus, der das Problem lösen kann, und andererseits die benötigte Rechenzeit, um ein Individuum zu bewerten. Aus diesen Gründen hat sich die Notwendigkeit ergeben, das Schachproblem in einzelne Module zu unterteilen und in einem modularen und hierarchischen Individuum wieder zusammenzufassen.

Für das GeneticChess System wurden im Rahmen der vorliegenden Arbeit drei neue Individuentypen entwickelt:

- *Linear-Tree*,
- *Linear-Graph*,
- *State-Space-GP*.

Alle neuen Individuentypen sollen zu einer größeren Komplexität der Lösungen und zu einer höheren Flexibilität der evolvierten Programme führen. Wenn im weiteren über *Individuenstrukturen* gesprochen wird, sind immer die Standardstrukturen, Linear-Tree oder Linear-Graph gemeint.

Die Individuenstrukturen *Linear-Tree* und *Linear-Graph* sind *hybride Individuen*, damit sind GP-Strukturen gemeint, die sich aus verschiedenen Strukturen zusammensetzen. Normalerweise besteht der Aufbau eines Individuentyps immer nur aus einer Struktur, also reine lineare Individuen, Bäume oder Graphen. Jedoch wurde vorher nie versucht zwei oder mehrere Strukturen in einem Individuum zu verbinden. Die Individuentypen Linear-Tree und Linear-Graph beruhen aber genau auf diesem Ansatz. Diese Verschmelzung zweier Strukturen erzwingt in den Individuen einen Aufbau, der es der Evolution erleichtern soll, ein Problem in Teilprobleme aufzuteilen.

Das *State-Space GP* unterscheidet sich von den beiden anderen Individuenstrukturen dadurch, dass es nicht eine Erweiterung der Struktur eines Individuums ist, sondern dass der Programmfluss des Individuums in gewissen Bereichen vorgegeben ist. Der Programmfluss in diesem Individuum ist so angelegt, dass es Teile seines Codes rekursiv aufrufen kann.

Die Besonderheit des Programmflusses verbindet die drei neuen Individuenstrukturen miteinander. In den Individuenstrukturen *Linear-Tree* und *Linear-Graph* wird der Baum oder der Graph als Grundlage des Programmflusses benutzt, bei dem *State-Space GP* wird der Programmfluss durch die Definition der einzelnen Individuenteile und deren Aufrufreihenfolge bestimmt. Wenn man die aktuellen Strukturen von GP-Individuen untersucht, sieht man sofort, dass die Individuenstruktur keine Kontrollstrukturen für den Programmfluss besitzt. Individuen erzielen ihr Ergebnis allgemein durch Modifikation des Datenflusses. In linearen Strukturen ist dies ein Sprungbefehl, der Knoten des Individuums überspringt, wenn die Aussage des Sprungs wahr ist. Durch Verwendung dieses Befehls können nicht nur Knoten zu Exons werden, die vorher Introns [18, 76] waren. Der Effekt des Sprungbefehls ist somit mehr als nur das Überspringen eines Knotens, sondern die Folge der Exons ab dem Sprungbefehl. Jedoch muss es der Evolution hier gelingen, mehrere Programmflüsse in eine lineare Sequenz einzubauen. Durch die Vorgabe einer Struktur, die nur den Programmfluss darstellt, erleichtert man somit der Evolution, solche Programmflüsse zu finden. Die Vereinfachung sollte den Individuen nun ermöglichen die gegebenen Resourcen zu nutzen, um schneller eine gute Lösung zu finden.

Wenn man sich den Kontrollfluss in Programmen verdeutlicht, die von Menschen geschrieben wurden, findet man nur in einfachen Routinen lineare Programmflüsse. Normalerweise enthalten diese Programme sehr komplexe Programmflüsse, betrachtet man ein einfaches Schachprogramm, so findet man auch in ihnen einen komplexen Programmfluss. Wenn man dies als Grundlage nimmt, kann man davon ausgehen, dass der Programmfluss ein wichtiges Instrument ist, um komplexe Programme zu implementieren. Daher wurden die neuen GP-Strukturen mit dem Ziel entwickelt, Individuen und damit auch Programme zu evolvieren, die nicht nur auf dem Datenfluss beruhen, sondern einen starken Gebrauch von dem Programmfluss machen.

In den folgenden Abschnitten werden nun die verschiedenen Individuenstrukturen genauer beschrieben, wobei die Reihenfolge der Kapitel deren zeitlicher Entwicklung folgt. Abschließend werden die Ergebnisse der einzelnen Strukturen auf verschiedenen Benchmarkproblemen verglichen. Hierbei wird das *State-Space GP* nicht mit analysiert, da die implementierte Variante starke Spezialisierungen für das Schachproblem beinhaltet und so nur in diesem Umfeld ihre Stärken zeigen kann.

4.1 Linear-Tree GP

Linear-Tree ist eine neue Repräsentation eines GP-Programms, die erstmals eine Kombination von Daten- und Programmfluss durch die Struktur der Individuenrepräsentation vorgibt. Der Name des Individuentyps impliziert schon, dass das Individuum eine

Kombination aus einer linearen und einer Baumstruktur ist. Die lineare Struktur des Individuums stellt den Datenfluss während der Evaluation dar und die Baumstruktur den Programmfluss. Weitere Ergebnisse zu dieser Struktur wurden auf der EuroGP Konferenz 2001 veröffentlicht [62].

4.1.1 Die Struktur

Die *Linear-Tree* Struktur ist eine Kombination aus einer Baumstruktur und einer linearen Struktur. Abbildung 4.1 verdeutlicht den Aufbau der Linear-Tree Struktur, bestehend aus einem Baum als Grundstruktur, und den Aufbau der einzelnen Knoten des Baumes.

Die Baumstruktur eines Linear-Tree Individuums ist nicht mit der Baumstruktur eines baumbasierten GP-Individuums zu vergleichen. Es ergeben sich hierbei Unterschiede in der Auswertung der Individuen und lineare Sequenzen in den einzelnen Knoten des Baums. Im Falle von Linear-Tree ist der Knoten in zwei Komponenten unterteilt, eine *lineare Sequenz* von Befehlen und einn *Branching Knoten*. Abbildung 4.2 zeigt den Aufbau des Knotens mit den beiden Komponenten.

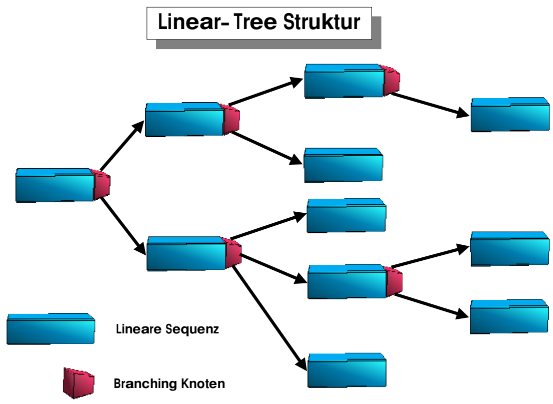

Abbildung 4.1: Struktur eines Linear-Tree Individuums.

Die *lineare Sequenz* hat den Aufbau eines normalen linearen Individuums mit variabler Länge. Jeder Knoten in dieser Sequenz ist eine Funktion, die auf indexierte Variablen, Konstanten oder Terminale zugreifen kann. Die Knoten werden dabei immer als Registeraufrufe implementiert, also in der Form:

$$Reg(1) = op(Reg(2), x), \text{mit } x \in \mathcal{R}, \mathcal{K}, \mathcal{T}.$$

Wobei $\mathcal{R}$ die Menge der indexierten Variablen oder auch Register ist, $\mathcal{K}$ die Menge der Konstanten und $\mathcal{T}$ die Menge der Terminale. Während der Interpretation eines Individuums wird über die Register der Datenfluss der linearen Sequenz etabliert. Während der gesamten Evolution gilt für die lineare Sequenz, dass ihre Länge in einem vorgegebenen Intervall liegen muss. Dadurch erzwingt man, dass sich keine Knoten bilden können, deren lineare Sequenz die Länge 0 hat und die somit nicht zum Gesamtergebnis beitragen können oder durch die Bloat Problematik zu unendlichem Wachstum tendieren (siehe auch [20]).

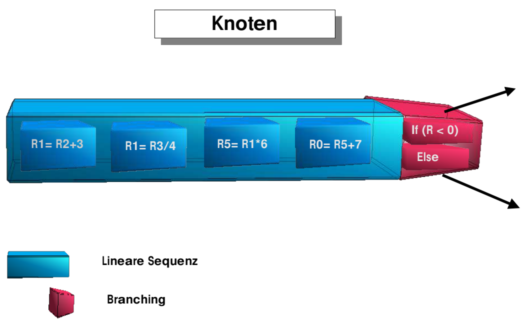

Abbildung 4.2: Der Aufbau eines Linear-Tree Knotens.

Der *Branching Knoten* enthält immer eine Entscheidungsfunktion, die auf der Grundlage der durch die lineare Sequenz berechneten Werte eine der Nachfolgerknoten des Knotens bestimmt, der dann ausgewertet wird. Dadurch bestimmt auch die *Branching Funktion* den Ausgrad des Knotens im Baums. Kann eine Funktion nur eine binäre Entscheidung treffen, hat der Knoten nur zwei ausgehende Kanten. Die Struktur des Baumes wird somit indirekt über die gewählten Branching Funktionen bestimmt. Wählt man Funktionen mit höherem Ausgrad, erhält man somit auch Bäume mit einem höheren Verzweigungsgrad. Es wird jedoch auch zugelassen, dass bei binären Funktionen beide Kanten den gleichen Nachfolger verbinden.

Wenn man diese Struktur nun auf einer abstrakteren Ebene betrachtet, bildet der Baum die Struktur eines Programms ab. Jeder Knoten ist ein Unterprogramm mit dem Branching Knoten als das Element, welches den Programmfluss steuert.

4.1.2 Evaluierung der Linear-Tree Struktur

Der entscheidende Unterschied zwischen einem Linear-Tree Individuum und einem normalen baumbasierten Individuum ist bei der Evaluierung des Individuums zu sehen. In einem Linear-Tree Individuum werden nicht alle Knoten des Baumes für jeden Fitnesscase ausgewertet, es wird immer nur ein Pfad durch den Baum bearbeitet. Die Interpretation beginnt in der Wurzel des Baumes, dort wird zuerst die lineare Sequenz ausgewertet. Sie berechnet verschiedene Werte, die in den Registern zwischengespeichert werden. Ist die gesamte Sequenz des Knotens abgearbeitet, wird die Funktion im Branching Knoten berechnet. Sie entscheidet mit Hilfe der Zwischenergebnisse, welcher der Nachfolgerknoten als nächstes ausgewertet wird. Dies wird solange durchgeführt, bis ein Knoten erreicht wird, der keine Nachfolgerknoten mehr hat. Als Ergebnis wird dann der Wert eines Registers zurückgegeben, welches vorher als Ergebnisregister definiert wurde.

Da der Weg durch den Baum, also der Programmfluss, abhängig von dem Ergebnis der Branching Funktion ist, ist er somit auch abhängig von den Eingabewerten. Damit ergeben sich für verschiedene Eingaben verschiedene Wege durch den Baum. Die Abbildung 4.3 zeigt beispielhaft verschiedenen Programmflüsse durch den Baum. Ein Knoten des Programmbaumes kann dadurch für einige Eingaben aktiv sein und für andere nicht. Dies ermöglicht es der Evolution, Module zu entwickeln, die einen Teil des Problems lösen können. Durch die Komposition verschiedener Module für verschiedene Eingabemengen kann somit ein Gesamtprogramm geschaffen werden. Anders ausgedrückt ermöglicht diese Struktur eine Dekomposition des Problems in einfache Teilprobleme.

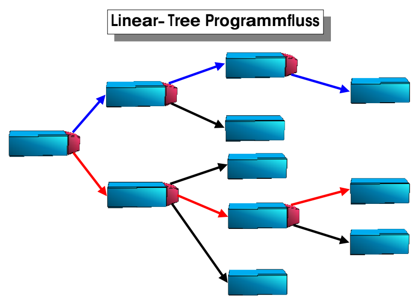

Abbildung 4.3: Der Programmfluss durch ein Linear-Tree Individuum. Mit Rot sind die Kanten für den Programmfluss für eine Eingabe X gekennzeichnet und mit Blau die Kanten für den Programmfluss für eine Eingabe Y.

Um die Module aber für verschiedene Eingaben aktivieren zu können benötigt man Funktionen, die den Programmfluss steuern. Diese werden im weiteren immer *Branching Funktionen* genannt. Die Aufgabe der Funktionen ist es aufgrund der Zwischenergebnisse zu entscheiden, welcher der Nachfolgerknoten aktiviert wird. Dazu muss eine Funktion die Zwischenergebnisse betrachten können, alle Ergebnisse einer Berechnung werden in linearen GP Systemen immer in den Registern gespeichert. Eines der wichtigsten Register ist das Ergebnisregister, also das Register, aus dem am Ende der Evaluation der Wert entnommen wird, der als Ausgabe interpretiert wird. Somit ist es logisch, diese Register während der Evaluation zu überwachen, um anhand der Änderungen des Inhaltes eine Entscheidung zu

Branching Funktion	Beschreibung der Funktion
Ergebnisregister < 0	Wenn der Wert des Ergebnisregisters größer Null ist, wird der linke Nachfolger gewählt, sonst der rechte.
Ergebnisregister > 0	Wenn der Wert des Ergebnisregisters kleiner Null ist, wird der linke Nachfolger gewählt, sonst der rechte.
Ergebnisregister $== 0$	Wenn der Wert des Ergebnisregisters gleich Null ist, wird der linke Nachfolger gewählt, sonst der rechte.
Ergebnisregister $<$ Konstante	Wenn der Wert des Ergebnisregisters größer einer Konstante des Knotens ist, wird der linke Nachfolger gewählt, sonst der rechte.
Ergebnisregister $>$ Konstante	Wenn der Wert des Ergebnisregisters kleiner einer Konstante des Knotens ist, wird der linke Nachfolger gewählt, sonst der rechte.
Ergebnisregister $==$ Konstante	Wenn der Wert des Ergebnisregisters gleich einer Konstante des Knotens ist, wird der linke Nachfolger gewählt, sonst der rechte.
Ergebnisregister $<$ Register(i)	Wenn der Wert des Ergebnisregisters größer als der Wert des Registers i ist, wird der linke Nachfolger gewählt, sonst der rechte.
Ergebnisregister $>$ Register(i)	Wenn der Wert des Ergebnisregisters kleiner als der Wert des Registers i ist, wird der linke Nachfolger gewählt, sonst der rechte.
Ergebnisregister $==$ Register(i)	Wenn der Wert des Ergebnisregisters gleich dem Wert des Registers i ist, wird der linke Nachfolger gewählt, sonst der rechte.

Tabelle 4.1: Beispiele für binäre Branching Funktionen, wie sie auch in den Tests benutzt wurden. Das Ergebnisregister ist in den meisten Fällen das Register 0.

treffen. Die Tabelle 4.1 enthält eine Sammlung von Funktionen, wie sie während der Testläufe benutzt wurden. Es sind auch andere Funktionen möglich, jedoch sind mit diesen Funktionen gute Ergebnisse erzielt worden.

Die Module oder auch Unterprogramme, die durch die Struktur gebildet werden können, stellen somit in Kombination mit einer durch den Programmfluss gesteuerten Aktivierung dieser Module einen großen Vorteil gegenüber den Standard GP-Systemen dar. Dadurch, dass ein Knoten im Baum einem Modul entspricht, werden die Module durch die Struktur nicht nur gebildet, sondern auch, anders als bei Standard GP Strukturen, während des Crossovers und der Mutation geschützt. Wie man im folgenden durch das Vorgehen des Crossover Operators klar erkennen kann.

4.1.3 Rekombination der Linear-Tree Struktur

Die Crossover Operation ist eine der wichtigsten Rekombinationsoperationen in der Genetischen Programmierung, sie ermöglicht es, dass große Mengen genetischen Materials von einem Rekombinationspartner zum nächsten gelangen. Dies ermöglicht es den Individuen, während der Evolution sogenannte *Building Blocks* auszutauschen.

Der Crossover Operator für die Linear-Tree Struktur hat zwei Möglichkeiten, genetisches

Material zwischen Individuen zu transportieren. Die eine Möglichkeit ist äquivalent der des Baum GP's, in der Teilbäume zwischen den Individuen getauscht werden (siehe [6]). Die Abbildung 4.4 zeigt die Baum basierte Crossovermethode für ein Linear-Tree Individuum. Aus jedem Eltern-Individuum werden zufällig Teilbäume ausgewählt und zwischen den Individuen ausgetauscht. Wichtig ist beim baumbasierten Crossover, dass nicht zu oft Terminale als Knoten gewählt werden, da die Bäume sonst sehr schnell degenerieren. Daher werden bei diesem Ansatz die inneren Knoten eines Baumes immer mit einer höheren Wahrscheinlichkeit ausgewählt als die Blätter des Baumes. Normalerweise werden innere Knoten mit einer Wahrscheinlichkeit zwischen 70 und 90 % gewählt. Durch diese Wahl der Crossoverpunkte im Baum ergibt sich ein starkes *Bloat Problem*. Um dies zu verhindern wird diese Crossovermethode abgebrochen und ein linearbasiertes Crossover durchgeführt. Weiteres zu den gewählten Verfahren und Parametern ist im Abschnitt 4.4.4 zu finden.

Die zweite Möglichkeit des Crossovers ist die des Crossovers zwischen den linearen Sequenzen in den Knoten (siehe [7]). Wie die Abbildung 4.5 zeigt, werden bei dieser Rekombinationsmethode lineare Segmente zufälliger Länge zwischen den beiden Knoten ausgetauscht. Da es auch bei diesem Verfahren sehr schnell zum Wachstum der linearen Sequenz kommt, wird das Crossover beschränkt, wenn es die Maximallänge der Sequenz erreicht. Falls dies passiert, werden Segmente gleicher Länge zwischen den Sequenzen ausgetauscht. Dieses Standardverfahren für lineare GP Individuen verhindert das unkontrollierte Wachstum der Sequenzen.

Für ein Linear-Tree Individuum werden beide Möglichkeiten des Austausches von genetischem Material benutzt. Es wird für jede Crossoveroperation vorher mit einer festgelegten Wahrscheinlichkeit bestimmt, welches der beiden Verfahren verwendet wird. Der Rekombinationsoperator folgt dabei dem folgenden Algorithmus:

1. Wähle die Crossoverpunkte p_1, p_2 in beiden Individuen.
2. Wähle mit der Wahrscheinlichkeit $prob_{xover}$ die baumbasierte Crossovermethode (gehe zu Schritt 3). Wähle mit der Wahrscheinlichkeit $1-prob_{xover}$ die linearbasierte Crossovermethode (gehe zu Schritt 4).
3. Wenn die Tiefe des neuen Baumes die maximale Tiefe nicht überschreitet, wird das baumbasierte Crossover ausgeführt, sonst wird mit Schritt 4 weitergemacht.
4. Führe ein linearbasiertes Crossover aus.

Für den Parameter $prob_{xover}$ wurde während der Tests ein Wert von 20 % gewählt. Dies kann dazu führen, dass das baumbasierte Crossover nur bei maximal 20 % der Crossoveroperationen durchgeführt wird. Durch das Verfahren kommt es allerdings dazu, dass der Anteil des baumbasierten Crossovers in Wirklichkeit noch geringer ist. Dies geschieht dann, wenn die Bäume zu groß werden und die lineare Crossovermethode aufgerufen wird. Die Grundidee hinter diesem Ansatz ist, dass ein baumbasierter Crossoverschritt größere Auswirkungen auf ein Individuum hat als ein linearbasierter Crossoverschritt. Die baumbasierte Crossovermethode verändert die Struktur des Individuums auf Baumebene und auf linearer Ebene. Damit verändert er sowohl die Struktur des Programmflusses als auch den Datenfluss des Individuums, ein Crossover auf linearer Ebene kann zwar auch den Programmfluss eines Individuums verändern, dadurch dass es im Branching Knoten durch ein verändertes Ergebnis zu einem anderen Ergebnis kommen kann. Diese starken Veränderungen sind aber besonders am Ende der Evolution mehr zerstörend als aufbauend, daher sollen durch den kleineren Wert für das baumbasierte Crossover die starken Veränderungen beschränkt werden. Die Wahrscheinlichkeit, dass es zu einem baumbasierten Crossover in der späteren Phase der Evolution kommt, wird durch den verwendeten

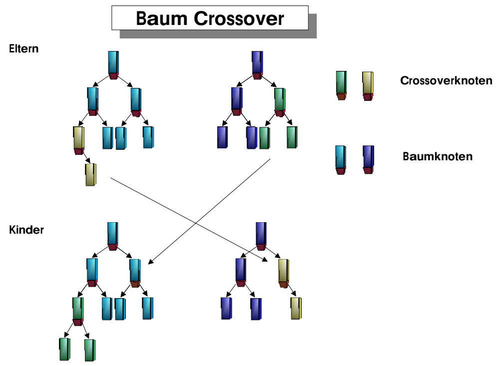

Abbildung 4.4: Crossover Operation Linear-Tree von Individuen, die die baumbasierte Crossovermethode benutzen. Diese Methode tauscht, nachdem zwei Crossoverpunkte gewählt wurden, die Teilbäume aus.

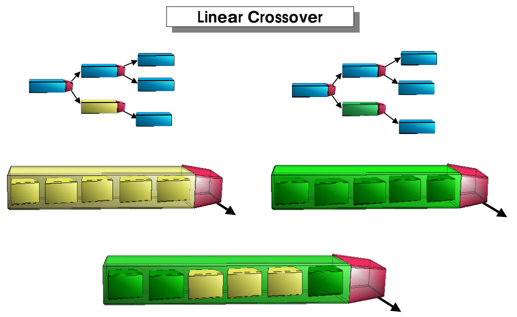

Abbildung 4.5: Crossover Operation Linear-Tree von Individuen, die die linear-basierte Crossovermethode benutzen. Diese Methode führt ein Zwei-Punkte-Crossover zwischen zwei linearen Sequenzen aus.

Algorithmus selbst noch verringert. Denn wenn ein Baum durch das Crossover zu groß wird, wird das lineare Crossover durchgeführt. Dies passiert am häufigsten am Ende der Evolution, wenn die Bäume schon sehr tief geworden sind, dies ergibt eine automatische Anpassung des Parameters durch die Evolution.

4.1.4 Mutation

Die Mutation ist die zweite Operation, die eine Veränderung des genetischen Materials während der Evolution durchführt. Wie schon der Crossoveroperator für ein Linear-Tree Individuum hat auch der Mutationsoperator zwei Phasen. In der einen Phase wird eine Mutation der linearen Sequenzen durchgeführt und in einer weiteren Phase wird eine Mutation durchgeführt, die zu einer Veränderung des Programmflusses führt.

Während der Mutation werden zufällig Knoten in der Baumstruktur des Individuums ausgewählt. Die Anzahl der Knoten wird durch einen Parameter festgelegt, der vor der Evolution festgelegt wird. Danach durchläuft jeder Knoten die beiden Phasen der Mutation, dies unterscheidet den Mutationsoperator vom Crossoveroperator, bei dem immer nur eine der beiden Möglichkeiten benutzt wurde. Die Mutation der linearen Sequenzen entspricht der Mutation von linearen Individuen (siehe [7]), hierzu werden zufällig Knoten des Individuums ausgewählt und die Funktionen, die Terminale, die Register oder die Konstanten verändert. Anschließend wird die zweite Phase der Mutation begonnen, wodurch der Programmfluss des Individuums verändert wird. Hierzu wird nicht die Baumstruktur verändert, indem Unterbäume vertauscht werden, sondern die Branchingfunktion wird ersetzt oder verändert. Dies führt dazu, dass nun bei einer Eingabe nicht der rechte, sondern der linke Sohn ausgeführt wird. Da dies eine starke Veränderung des Ein- Ausgabeverhaltens des Individuums bewirkt, wird die zweite Phase der Mutation auch nicht immer ausgeführt, sondern nur mit einer Wahrscheinlichkeit von 5 %. Dadurch wird gewährleistet, dass der Programmfluss nicht zu stark beeinflusst wird.

4.2 Linear-Graph GP

Die *Linear-Graph* Struktur ist eine direkte Weiterentwicklung der Linear-Tree Struktur aus Kapitel 4.1. Der Unterschied zur *Linear-Tree* Repräsentation eines GP-Programms, die erstmals eine Kombination von Daten- und Programmfluss durch die Struktur der Individuenrepräsentation vorgibt, ist, dass die Struktur, über die der Programmfluss kontrolliert wird, ein Graph ist. Die lineare Struktur des Individuums stellt den Datenfluss während der Evaluation dar und die Graphstruktur den Programmfluss. Die Grundstruktur ist ein Graph, in deren Knoten lineare Sequenzen von Befehlen enthalten sind. Weitere Ergebnisse zu dieser Struktur wurden auf der EuroGP Konferenz 2002 veröffentlicht [63].

4.2.1 Die Struktur

In der *Linear-Graph* Struktur wird der Programmfluss durch einen gerichteten azyklischen Graphen repräsentiert. Abbildung 4.6 verdeutlicht den Aufbau der Linear-Graph Struktur, bestehend aus einem azyklischen, gerichteten Graphen als Grundstruktur und dem Aufbau der einzelnen Knoten. Es wurde ein azyklischer gerichteter Graph gewählt, da sich hierdurch keine Zyklen in dem Programmfluss ergeben können. Ein Zyklus im Graphen würde dazu führen, dass die Evaluierung des Individuums nicht terminiert.

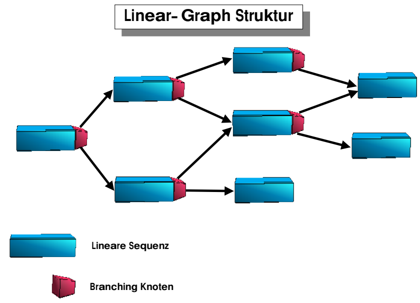

Abbildung 4.6: Struktur eines Linear-Graph Individuums. Der Graph bildet den möglichen Programmfluss ab und die linearen Sequenzen den Datenfluss in den einzelnen Knoten.

Als weitere Vereinfachung der Struktur und somit eine bessere Möglichkeit, die Individuen bei der Rekombination zu kontrollieren, wurde ein schichtweiser Aufbau gewählt. Der Startknoten des Graphen liegt immer auf der Ebene 1 und seine Nachfolger auf der Ebene 2. Jeder Nachfolger eines Knotens befindet sich immer auf einer Ebene, die mindestens um eins größer ist als die Ebene, auf der sich der Elternknoten befindet. Es sind auch keine Kanten zwischen Knoten einer Ebene oder eine Kante mit der Richtung von Ebene i nach $i-1$ erlaubt. Dadurch erhält man sofort einen azyklischen Graphen. Durch diese Definition der Graphstruktur des Individuums ermöglicht man zwar nur eine Teilmenge aller möglichen azyklischen Graphen, jedoch ist diese Beschränkung für die Entwicklung möglicher Programmflüsse zu vernachlässigen.

Die Knoten, die keine Nachfolger haben, werden auch als Terminale bezeichnet, dies wird im Abschnitt über das Crossover der Linear-Graph Struktur 4.2.3 an Bedeutung gewinnen. Die Terminalknoten des Graphen können auf jeder Ebene des Graphen liegen, sie werden nur über ihr Nicht-Vorhandensein von Nachfolgerknoten definiert. An den Terminalknoten wird die Evaluierung eines Individuums beendet, da an ihnen immer ein Pfad durch den Graphen endet (siehe Abschnitt 4.2.2). Die Abbildung 4.7 verdeutlicht noch einmal den Aufbau der Graphstruktur die möglichen Verbindungen und die verbotenen Verbindungen zwischen zwei Knoten des Graphen.

Durch die Veränderung der Grundstruktur der Linear-Graph Struktur gegenüber der Linear-Tree Struktur ermöglicht man den Individuen, gewisse Programmflüsse einfacher darzustellen. Ein Beispiel ist eine Aufspaltung des Programmflusses und ein späterer Zusammenfluss. Dies konnte in den Linear-

Tree Individuen nur durch Kopien der linearen Sequenzen geschehen, was auch oft bei Linear-Tree Programmen geschehen ist (siehe hierzu Abschnitt 4.4.3). Jedoch ist es in der Linear-Tree Struktur komplizierter, im evolutionären Prozess durch Rekombination und Mutation zu solchen Programmflüssen zu gelangen als in einer Linear-Graph Struktur. Hierzu muss nur eine Kante durch eine Mutation verändert werden, im anderen Fall kann dies erst durch die Rekombination zweier Individuen geschehen, die gleiche lineare Sequenzen enthalten. Die Komplexität des Aufbaus steigt für ein Linear-Tree Individuum mit der Anzahl der Ebenen, die zwischen der Aufspaltung des Programmflusses und der Zusammenführung liegen. Dies zeigt auch, dass durch die Graphstruktur kompliziertere Programmflüsse implementiert werden können als durch eine Baumstruktur.

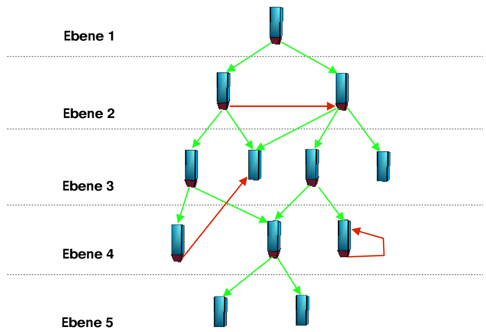

Abbildung 4.7: Auf den Ebenen 3, 4 und 5 findet man Terminalknoten. Die grünen Kanten zeigen erlaubte Verbindungen im Graphen, die roten verbotene Verbindungen.

Ein Knoten eines Linear-Graph Individuums entspricht im Aufbau dem Knoten der Linear-Tree Struktur, siehe dazu Abschnitt 4.1.1 und Abbildung 4.2.

4.2.2 Evaluierung der Linear-Graph Struktur

Die Evaluierung der Linear-Graph Struktur zeichnet sich dadurch aus, dass nicht alle Knoten des Graphen für jeden Fitnesscase ausgewertet werden.

Da der Datenfluss in der linearen Sequenz und den Knoten des Graphen untereinander geschieht, werden zu Beginn jeder Auswertung die globalen Register mit eins initialisiert und die Datenregister mit den entsprechenden Daten. Somit ergeben sich für jedes Individuum auf jeder Eingabe immer die gleichen Voraussetzungen.

Es wird immer nur ein Pfad durch den Graphen, beginnend mit dem Startknoten, ausgewertet. Durch den schichtweisen Aufbau des Graphen ergibt sich eine natürliche Reihenfolge der Knoten des Graphen. Der Startknoten, an dem jede Evaluierung des Individuum beginnt, ist der einzige Knoten auf der ersten Ebene des Graphen (siehe Abbildung 4.7). Nachdem die lineare Sequenz des ersten Knotens ausgewertet ist und das Resultat dieser Berechnung in den Registern gespeichert ist, wird der Branching Knoten ausgewertet. Die Auswertung der linearen Sequenz ist analog zu der Auswertung von linearem GP. Nach der Auswertung wird der Branchingteil jedes Knotens ausgewertet.

Die Branchingfunktion unterliegt nur einer Einschränkung durch die Individuenstruktur, die Funktion muss eine Entscheidungsfunktion sein. Sie ist aber nicht auf binäre Entscheidungen begrenzt, auch der Eingaberaum der Funktion wird nur durch die dem GP System zur Verfügung stehenden Daten oder durch das System erzeugte Daten beschränkt. In der Tabelle 4.1 werden die Branchingfunktionen beschrieben, die für die Tests benutzt wurden. Nachdem die Branchingfunktion ausgewertet wurde, wird der berechnetete Nachfolgerknoten als nächstes ausgewertet.

Vor der Interpretation des nächsten Knotens werden die Inhalte der globalen Register oder der Datenregister nicht verändert, denn der Datenfluss der Knoten untereinander funktioniert wie beim linearen GP durch die Register. Durch diesen Aufbau ergeben sich keine Probleme bei der Ausführung der verschiedenen Knoten beim Crossover und bei der Mutation der Individuen.

Die Interpretation der Struktur wird dann beendet, wenn ein Knoten ohne Nachfolger, also ein Terminal, gefunden wird. Da durch den Aufbau des Individuums Terminale auf verschiedenen Ebenen liegen können, ist die Länge des Pfades auch nicht bei jeder Eingabe gleich.

Die Abbildung 4.3 verdeutlicht, mit Hilfe der Linear-Tree Struktur, die verschiedenen Pfade durch den Graphen für verschiedene Eingaben. Hier wird auch deutlich, dass Bereiche des Graphen nicht evaluiert werden, sondern dass immer nur ein Pfad durch den Graphen genutzt wird.

4.2.3 Crossover der Linear-Graph Struktur

Der Crossover Operator für die Linear-Graph Struktur hat wie der Crossover Operator der Linear-Tree Struktur zwei Möglichkeiten genetisches Material zwischen Individuen zu transportieren. Die eine Möglichkeit ist die des Graph GP, in der Teilgraphen zwischen den Individuen getauscht werden (siehe [117, 5, 99]). Die Abbildung 4.8 zeigt die graphbasierte Crossovermethode für ein Linear-Graph Individuum. Aus jedem Elternpaar werden zufällig Teilgraphen ausgewählt und zwischen den Individuen ausgetauscht. Ähnlich wie beim linearen und Baum GP gibt es auch in Linear-Graph Strukturen ein *Bloat Problem*, diese Struktur hat dieses Problem sogar auf zwei Ebenen. Eines liegt auf der Ebene der Graphstruktur und das andere auf der Ebene der linearen Sequenzen. Das Bloat Problem auf der Graphebene hat zur Folge, dass die Graphen in ihrer Tiefe immer weiter wachsen. Dies ergibt sich automatisch aus dem Schichtenaufbau des Graphen. Damit es nicht zu einem zu starken Tiefenwachstum der Graphen kommt, wird durch einen Parameter die maximale Anzahl von Ebenen der Individuen angegeben. Falls der Graph die maximale Anzahl der Ebenen durch ein Crossover überschreiten würde, wird diese Crossovermethode abgebrochen und ein linearbasisertes Crossover durchgeführt. Eine andere Variante wäre es, gleich tiefe Teilgraphen auszutauschen, wie es auch im Falle der linearen Strukturen gemacht wird. Es hat sich aber in Tests gezeigt, dass diese Crossovervariante für die Evolution in späteren Generationen nicht mehr so wichtig ist. Da die Graphen durch den Bloat Effekt bis an die gegebenen Grenzen wachsen, kommt es dann hauptsächlich zu Crossover der linearen Sequenzen. Somit verändert sich der Anteil des Graph Crossover automatisch durch den Algorithmus, ohne einen weiteren Parameter einzuführen oder eine Selbstanpassung durchzuführen. Weiteres zu den gewählten Verfahren und Parametern ist im Abschnitt 4.1.3 zu finden.

Die zweite Möglichkeit des Crossovers ist die des Crossovers zwischen den linearen Sequenzen in den Knoten. Wie auch die Abbildung 4.5 zeigt, werden bei dieser Rekombinationsmethode lineare Segmente zufälliger Länge zwischen den beiden Knoten ausgetauscht. Da es bei diesem Verfahren sehr schnell zum Wachstum der linearen Sequenz kommt, wird das Crossover beschränkt, wenn es die Maximallänge der Sequenz erreicht. Falls dies passiert, werden Segmente gleicher Länge zwischen den Sequenzen ausgetauscht. Dieses Standardverfahren für lineare GP Individuen verhindert das unkontrollierte Wachstum der Sequenzen.

Für ein Linear-Graph Individuum werden beide Möglichkeiten des Austausches von genetischem Materials benutzt. Es wird für jede Crossoveroperation vorher mit einer festgelegten Wahrscheinlichkeit bestimmt, welches der beiden Verfahren verwendet wird. Der Rekombinationsoperator folgt dabei dem folgenden Algorithmus:

1. Wähle die Crossoverpunkte p_1, p_2 in beiden Individuen.
2. Wähle mit der Wahrscheinlichkeit $prob_{xover}$ die graphbasierte Crossovermethode (gehe zu Schritt 3). Wähle mit der Wahrscheinlichkeit $1-prob_{xover}$ die linearbasierte Crossovermethode (gehe zu Schritt 4).

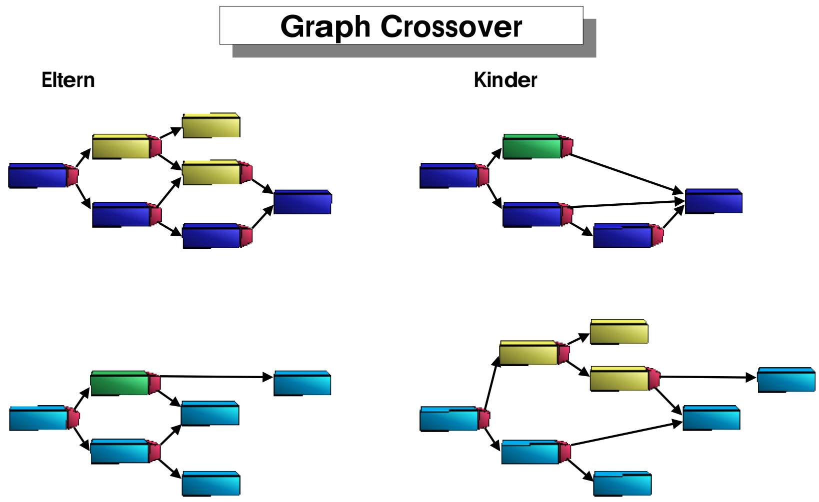

Abbildung 4.8: Crossoveroperation zweier Linear-Graph Programme. Hier wird das graphbasierte Crossover genutzt. Das Crossover tauscht zwei Teilgraphen der Individuen aus.

3. Wenn die Anzahl der Ebenen des neuen Graphen die maximale Anzahl nicht überschreitet, wird das graphbasierte Crossover ausgeführt, sonst wird mit Schritt 4 weitergemacht.

4. Führe ein linearbasiertes Crossover aus.

Der Parameter $prob_{xover}$ wurde während der Tests auf einen Wert von 20 % festgelegt. Dies führt dazu, dass das graphbasierte Crossover nur bei maximal 20 % der Crossoveroperationen durchgeführt wird. Durch das Verfahren kommt es allerdings dazu, dass der Anteil des graphbasierten Crossovers bei der gesamten Evolution geringer ist. Wie oben erwähnt nutzt der Crossover Operator das linearbasierte Crossover, wenn der Graph zu groß wird. Die Einschränkung des Bloat Problems soll durch diesen Ansatz gefördert werden. Ein graphbasierter Crossoverschritt hat einen größeren Effekt auf ein Individuum als ein linearbasierter Crossoverschritt, da es dann auch zu Veränderungen im Programmfluss des Individuums kommt. Daher ist es sinnvoll, dass am Ende der Evolution, wenn das System in einer Optimierungsphase ist, nur noch kleinere Schritte gemacht werden.

4.2.4 Mutation

Die Mutation ist eine weitere wichtige Operation zur Variation des genetischen Materials während der Evolution. Auch der Mutationsoperator wird wie der Crossoveroperator für ein Linear-Graph in zwei Phasen durchgeführt. In der ersten Phase wird eine Mutation der linearen Sequenzen durchgeführt und in der zweiten Phase wird eine Mutation durchgeführt, die an dem Individuum zu einer Veränderung des Programmflusses führt.

Während der Mutation werden zufällig Knoten der Graphstruktur des Individuums ausgewählt. Die Anzahl der Knoten wird durch einen Parameter festgelegt, der vor der Evolution festgelegt wird. Danach durchläuft jeder Knoten die beiden Phasen der Mutation, dies unterscheidet den Mutationsoperator vom Crossoveroperator, bei dem immer nur eine der beiden Möglichkeiten benutzt wurde. Die Mutation der linearen Sequenzen entspricht der Mutation von linearen Individuen (siehe [7]), hierzu werden zufällig Knoten des Individuums ausgewählt und die Funktionen, die Terminale, die Register oder die Konstanten verändert.

Die zweite Phase der Mutation, in der der Programmfluss variiert wird, wird jedoch nur bei einem kleineren Teil aller Mutationen durchgeführt. Die Wahrscheinlichkeit, dass die zweite Phase ausgeführt wird, liegt bei dem System bei 5 %. Dieser Wert ist experimentell ermittelt, er ist nicht der optimale Wert für jedes Problem, aber doch ein guter Mittelwert. An dieser Stelle scheint auch eine automatische Adaptation möglich und nützlich. Um eine Veränderung des Programmflusses durchzuführen hat der Mutationsoperator zwei Möglichkeiten: einmal kann eine Kante zu einem neuen Knoten gelegt werden oder die Branchingfunktion kann mutiert werden. Beide Möglichkeiten werden vom Mutationsoperator genutzt, wobei die Wahrscheinlichkeit für die Mutation der Branchingfunktion bei 20 % liegt, auch dieser Wert ist experimentell ermittelt worden. Dabei hat sich zwar gezeigt, dass der Wert vom gestellten Problem abhängig ist, aber auch, dass er sehr robust ist. Das heißt, der Wert liefert für ein größeres Intervall gute Ergebnisse.

4.3 State-Space GP

Das *State-Space GP* ist eine Struktur, die speziell mit dem Ziel entwickelt wurde, im *GeneticChess System* verwendet zu werden. Der Fokus dieses Individuums sind allgemeine Suchverfahren für Problemlösungsansätze, die als State-Space Ansatz modelliert wurden. Mit dieser Struktur wurden keine Tests auf den Benchmarkproblemen durchgeführt, da dieser Lösungsansatz für solche Probleme nicht sinnvoll ist. Es wird jedoch in diesem Kapitel vorgestellt, da diese Struktur nicht nur auf Schachanwendungen beschränkt ist, sondern auf alle Probleme angewendet werden kann, die eine *State-Space Repräsentation* verwenden [88].

Es stellt sich sofort die Frage, warum die Entwicklung einer speziellen GP Struktur sinnvoll oder notwendig ist. Es hat sich in der Vergangenheit gezeigt, dass gute Schachprogramme Suchverfahren sind. Alle Schachprogramme, die muster- , wissens- oder regelbasiert arbeiteten, konnten bis jetzt nicht annähernd die Ergebnisse der Programme erreichen, die mittels Suchverfahren arbeiten. Daher scheint es sinnvoll sich auf Suchverfahren zu konzentrieren, wenn man das Schachproblem lösen will. Viele der verwendeten Suchverfahren beinhalten auch Heuristiken, welche die Rekursionsentscheidung durchführen. Besonders bei der Lösung des Schachproblems kann man erkennen, wieviel verschiedene Heuristiken für die Rekursionsentscheidung entwickelt wurden (siehe Kapitel 2.2.6). Dies gilt nicht nur für die Problematik der Schachprogramme oder etwas allgemeiner für verschiedene Strategiespiele, sondern für verschiedene, schwere Suchprobleme, die nur durch Heuristiken berechnet werden können.

4.3.1 Die Struktur

Die Struktur eines *State-Space Individuums* unterscheidet sich von den oben vorgestellten GP Strukturen insoweit, dass das Individuum aus drei *Substrukturen* besteht, die jede eine eigenständige Bedeutung haben und nur durch Zusammenarbeit zu einer Lösung kommen können. Jede dieser *Substrukturen* ist selbst wieder ein GP Individuum, welches durch seine Aufgabe, nicht aber durch seine Struktur definiert ist.

Die Aufgaben der drei Substrukturen im Individuum sind:

1. das Modul der Tiefenentscheidung,
2. das Modul der Problembewertung und
3. das Modul der Blattbewertung.

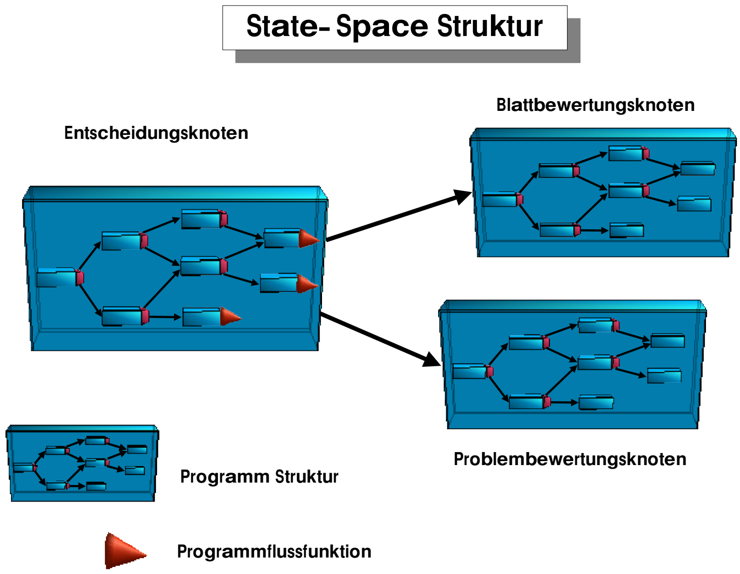

Abbildung 4.9: Struktur eines State-Space Individuums. Man kann die drei Substrukturen des Individuums erkennen und die Kombination dieser Substrukturen über den Programmflussabschnitt.

Neben diesen Substrukturen enthält das Individuum den Suchbaum der aufgespannten Knoten (State). Dieser Suchbaum ist noch eine weitere Möglichkeit für das Individuum, Daten zu extrahieren, da es auf die Ergebnisse der aktuellen Berechnung und der Vergangheit zugreifen kann. In der Abbildung 4.9 kann man die Kombination des Individuums aus den drei Substrukturen erkennen.

Die *Tiefenentscheidung* hat die Aufgabe, für den aktuellen Aufruf des Programms zu bestimmen, ob die Berechnung beendet wird, also die Blattbewertung aufgerufen wird, oder ob es durch die Problembewertung zu einem weiteren Aufruf kommt. Die Substruktur wird durch ein Linear-Graph Individuum realisiert. Im Bereich der Problemlösung oder spezieller in der von Schach, ist die Entscheidung, ob man einen Knoten im Suchbaum weiter expandiert, ein wichtiges Problem, welches die Güte eines Programms sehr stark beeinflussen kann. Daher werden auch für die Tiefenentscheidung verschiedenste *Heuristiken* angewendet um eine *gute Entscheidung* zu treffen. Die Heuristik wird aus problem- und domänenspezifischem Wissen erzeugt, wobei spezifisches Wissen oft vorhanden ist, aber die Implementierung einer guten Heuristik durch die *richtige* Kombination dieses Wissens nur sehr schwer zu erreichen ist. An dieser Stelle soll ein GP Individuum, dessen Funktions- und Terminalmengen problemspezifisches Wissen enthalten, die Aufgabe der Heuristikerstellung erhalten. Die Aufgabe der Substruktur ist es dann, eine Klassifikation der Eingabe unter Berücksichtigung der aktuellen Suchtiefe durchzuführen und zu entscheiden, ob es zu einer weiteren Suche oder zum Abbruch kommen soll. Dazu werden sowohl die Ergebnisse der Teile des Suchbaums genutzt, die schon erreicht wurden, und durch das zweite Modul bewertet wurden. Das Ergebnis der Substruktur wird als Eingabe für den *Programmflussabschnitt* genutzt. Der Programmflussabschnitt besteht aus zwei Funktionen, die miteinander kombiniert werden. Einerseits hat man eine binäre Entscheidungsfunktion, die Ergebnisse des Tiefenentscheidungsmoduls für die Entscheidung nutzen, ob die Problembewertung oder die Blattbewertung aufgerufen wird oder nicht. Durch diese Funktionen können verschiedenste Werte betrachtet werden, die während der Evaluation berechnet werden. Die Programmflussfunktionen erweitern jeden Terminalknoten der Linear-Graph Struktur der Tiefenentscheidungsstruktur. Dadurch werden verschiedene Programmflussabschnitte durch verschiedene Eingaben aktiviert.

Entscheidungsfunktion	Beschreibung der Funktion
Ergebnisregister < 0	Wenn der Wert des Ergebnisregisters kleiner Null ist, wird der Knoten expandiert, sonst wird er bewertet.
Ergebnisregister $<$ Konstante	Wenn der Wert des Ergebnisregisters kleiner einer Konstante ist, wird der Knoten expandiert, sonst wird er bewertet.
Brettwert $<$ Alpha	Die Funktion identifiziert einen Alphaschnitt.
Brettwert $>$ Beta	Die Funktion identifiziert einen Betaschnitt.
Zugerzeugung	Beschreibung der Funktion
Alle Züge	Alle erlaubten Züge werden expandiert.
aggressive Züge	Nur Schlagzüge werden expandiert.
defensive Züge	Nur defensive Züge werden expandiert.
Nullzug	Es wird ein Nullzug gemacht, also das Zugrecht an den Gegner abgegeben.

Tabelle 4.2: Die verschiedenen Funktionen die dem Individuum zur Verfügung stehen, um den Programmflussabschnitt zu implementieren.

Die zweite Funktion bestimmt, welche möglichen Züge im Suchbaum im nächsten Schritt erzeugt werden. Bei der Erzeugung der möglichen Züge kann man verschiedene Kriterien benutzen, so kann man alle möglichen Züge erzeugen oder für eine Schachstellung nur die Schlagzüge, im allgemeineren Fall wäre dies eine Greedymethode, die nur Verbesserungen der Situation zulässt. Weiterhin kann man auch nur defensive Züge erzeugen. Tabelle 4.2 gibt die verschiedenen Funktionen an, die im System implementiert sind, aus denen der Programmflussabschnitt kombiniert werden kann. Durch den Programmflussabschnitt ermöglicht man den Individuen einen sehr variablen Aufbau des Suchbaums und somit die Möglichkeit andere Wege im Suchraum zu finden.

Die *Problembewertung* ist der Teil des Individuums, der alle Knoten des Suchbaums der nächsten Ebene bewertet und so eine Reihenfolge vorgibt, in der die Teilprobleme abgearbeitet werden. Dies ist ein wichtiger Schritt, da die Ergebnisse der Knoten des Suchbaums, deren Suche schon beendet wurde, mit in die Tiefenentscheidung eingehen. Wenn schon frühzeitig gute Lösungen gefunden wurden, kann die Suche im Suchbaum eingeschränkt werden. Könnte die Aufgabe der Problembewertung exakt durchgeführt werden, würden schon diese Module ausreichen um das Problem mit einem Schritt zu lösen (siehe Abschnitt 2.2.5). Da dies jedoch für die gestellten Probleme nicht exakt möglich ist, muss auch hier eine Heuristik verwendet werden. Die Aufgabe der Problembewertung wird daher darauf reduziert zu bestimmen, ob ein möglicher Lösungsweg relativ besser ist als alle anderen Lösungswege auf dieser Ebene des Suchbaums. Für einen Suchalgorithmus ist es vorteilhaft, gute Lösungen früh während der Berechnung zu finden, da dann später nicht so gute Lösungsansätze schnell verworfen werden können. Das heißt, je besser die Heuristik an dieser Stelle ist, desto kleiner wird der Suchbaum und desto schneller wird die Berechnung abgeschlossen.

Das letzte Modul ist die *Blattbewertung*, es bewertet die Blätter des Suchbaums. Die Aufgabe des Moduls ist es, für verschiedene Situationen eine relative Bewertung zu finden. Das Problem für das Modul liegt darin, dass es die Bewertung aufgrund von unvollständigen Informationen treffen muss. Denn Entscheidungen, die während der Suche noch getroffen werden, haben entscheidenden Einfluss auf die Bewertung in der aktuellen Situation. Dieses Modul versucht nun die möglichen Entscheidungen in die Bewertung der aktuellen Situation einzubeziehen. Dadurch wird sofort deutlich, dass es sich an dieser Stelle des Individuums wieder um eine Heuristik handeln muss. Für das Schachproblem wurden für diese Probleme die verschiedensten Bewertungen entwickelt [10, 11, 15, 16, 21]. Da es sich um ein Problem handelt, welches großen Einfluss auf die Güte des Algorithmus hat, wird auch hier problemspezifisches Wissen verwendet.

Die einzelnen Substrukturen werden nun durch die Zusammenarbeit während der Ausführung zu einem Individuum, d.h. der Programmfluss vereinigt die Elemente zu einem Individuum. Durch die Unabhängigkeit der Teilstrukturen untereinander unterliegen sie keinem Zwang einer einheitlichen Struktur, so kann jede Teilstruktur theoretisch eine andere GP Struktur nutzen. Nur die Tiefenentscheidungstruktur unterliegt dem Zwang, entweder aus einer Linear-Graph oder Linear-Tree Struktur zu bestehen, da die Terminalknoten der Struktur durch die Programmflussabschnitte erweitert werden müssen. Für alle anderen Strukturen hätte diese Substruktur sonst nur einen möglichen Programmflussabschnitt. In der aktuellen Version des Systems werden aber alle Substrukturen durch Linear-Graph Strukturen implementiert. Denn in den Tests der einzelnen Strukturen hat sich gezeigt, dass die Linear-Graph Struktur die robusteste Struktur bezogen auf die Ergebnisse ist. Sie erzielte immer gute bis sehr gute Ergebnisse und die Varianz der Ergebnisse war sehr klein. Das ist für die gestellt Aufgabe sehr wichtig, da die Rechenzeiten sehr groß sind und dadurch nicht sehr viele Läufe durchgeführt werden können. Denn für das Schachproblem ist weniger das Durchschnittsverhalten des Systems interessant als die besten Ergebnisse aller Läufe.

4.3.2 Evaluierung der State-Space Struktur

Die Evaluierung eines State-Space Individuums unterscheidet sich stark von der anderer GP Individuen, denn diese Struktur ruft sich selbst wieder auf. Das hat zur Folge, dass die Evaluation ein weiteres Abbruchkriterium benötigt, da es sonst dazu kommen kann, dass das Individuum nicht terminiert. Das Abbruchkriterium des Individuums ergibt sich direkt aus der Aufgabe des Individuums und ist die maximale Tiefe und die Anzahl der Knoten im Suchbaum. Der Suchbaum wird von dem Individuum während der Evaluation erzeugt und als zusätzliche Eingabedimension verwendet.

Die Evaluation beginnt mit dem Tiefenentscheidungsmodul. Als Eingabe erhält das Individuum den aktuellen Suchbaum, dieser besteht am Beginn der Evaluation nur aus einem Knoten. Weiterhin erhält es als Zusatzinformation die aktuelle Tiefe des Knotens im Suchbaum, die maximale Tiefe des Suchbaums, die Anzahl der schon expandierten Knoten im Suchbaum, die Blattwerte und die Werte der Problembewertung der bis dahin expandierten Knoten. Die Aufgabe der Substruktur ist es nun zu entscheiden, ob die aktuelle Situation genauer untersucht werden soll oder ob sie nicht so interessant ist und die Suche hier abgebrochen werden soll und die aktuelle Stellung bewertet werden kann. Kommt das Modul zu der Entscheidung, dass der Knoten weiter expandiert werden muss, wird die entsprechende Funktion zur Erzeugung der möglichen nächsten Zustände im Suchbaum aufgerufen, die durch den Programmflussabschnitt des Terminalknotens gegeben wird.

Alle diese Knoten im Suchbaum werden dann von der Substruktur für die Problembewertung bewertet. Das Modul wird für jeden dieser Knoten aufgerufen und berechnet für jeden Knoten einen Wert. Diese Werte werden im weiteren als ein Gütemaß interpretiert und aufgrund dieser Werte wird die Reihenfolge festgelegt, in der die Knoten im weiteren Verlauf der Evaluation ausgewertet werden. Im System wird ein hoher Wert als besser interpretiert als ein kleiner Wert, d.h. der Knoten mit dem höchsten Wert wird im nächsten Schritt als erstes weiter expandiert. Für diesen Knoten wird dann wieder die Tiefenentscheidung aufgerufen. Kommt dieses Modul dann zum Ergebnis, dass der Knoten nicht weiter expandiert werden muss, wird die Substruktur für die Blattbewertung aufgerufen. Der resultierende Wert wird mit den entsprechenden Alpha- und Betawerten verglichen

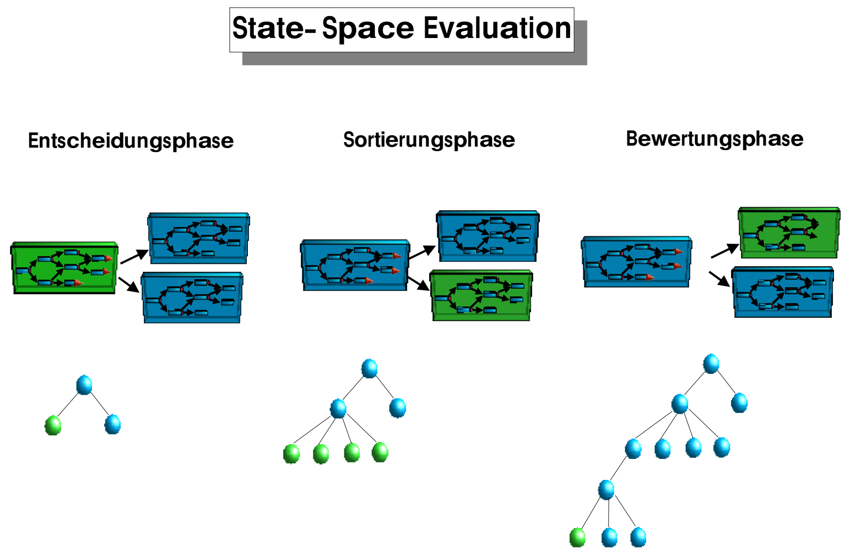

Abbildung 4.10: Der Programmfluss eines State-Space Individuums. Die Evaluation beginnt mit dem Entscheidungsmodul; wenn der Knoten expandiert werden soll, werden entsprechend der Funktionen im Programmflussabschnitt die möglichen Knoten im Suchbaum erzeugt. Dann werden alle diese Knoten in der Sortierungsphase bewertet und relativ zu diesen Werten sortiert. Danach wird für den besten Knoten wieder das Entscheidungsmodul aufgerufen. Kommt dieses Modul zu der Entscheidung, dass der Knoten nicht weiter untersucht werden muss, wird das Blattbewertungsmodul aufgerufen. Der Knoten wird bewertet und das Entscheidungsmodul wird dann für den nächsten Knoten auf dieser Ebene aufgerufen. Sind alle Knoten auf einer Ebene abgearbeitet, wird der nächste Knoten auf der höheren Ebene des Suchbaums behandelt.

und, wenn notwendig ersetzt, falls Funktionen diese Werte benutzen. Anschließend wird das Modul der Tiefenentscheidung am Knoten mit dem nächst kleineren Wert auf der höheren Ebene des Suchbaums weitergeführt. Die Abbildung 4.10 zeigt die drei Phasen, die jeder Knoten des Suchbaums während der Interpretation des Individuums durchläuft.

Man kann leicht erkennen, dass dieses Individuum eine Rekursion bei der Evaluation durchläuft, wobei die Tiefenentscheidung den Rekursionsschritt durchführt und die Blattbewertung das Rekursionsende markiert. Im Bereich der Genetischen Programmierung gibt es nicht viele Beispiele, die Rekursionen in die Individuen einbauen, ein Beispiel ist das Generic Genetic Programming System (GGP) von Man Leung Wong [127].

4.3.3 Die Evolution eines State-Space Individuums

Für die Evolution eines State-Space Individuums ergeben sich durch den modularen Aufbau des Individuums mehrere Möglichkeiten. Die einfachste Möglichkeit ist die direkte Evolution aller Substrukturen.Die zweite Möglichkeit besteht in einer mehrstufigen Evolution, in der jede Substruktur nacheinander evolviert und anschließend zu einem Individuum zusammengesetzt wird. Die letzte Möglichkeit ist eine stufenweise Evolution.

Das Verfahren, in dem das komplette Individuum evolviert wird, hat den Vorteil, dass sich so die einzelnen Substrukturen aufeinander einstellen können und zu einer Aufgabenteilung kommen.

Das zweite Verfahren, jedes Modul unabhängig zu evolvieren und anschließend zu einem Individuum zusammenzusetzen, hat zwar den Vorteil, dass man unabhängige Evolutionen parallel auf mehreren Rechnern durchführen kann. Jedoch hat sich gezeigt, dass die Module nicht sehr gut zusammenarbeiten. Da die Umgebung, also das Ausgabeverhalten der anderen Module während der Evolution simuliert werden muss, kann sich so nicht etwas wirklich Neues entwickeln. Denn die Vorgaben für die Substrukturen bestimmen das Gesamtergebnis zu stark. Da diese Art der Evolution schon in den ersten Untersuchungen nur schlechte Ergebnisse erzeugte, wurde diese Variante nicht weiter untersucht.

Das letzte Verfahren ist eine Kombination aus den beiden Verfahren, wobei es für dieses Verfahren auch zwei Varianten gibt. Die eine Möglichkeit ist es, die vorevolvierten Module während der Evolution des nächsten Moduls nicht mehr zu verändern, die andere Variante ist es, auch diese Module, jedoch mit einer kleinen Wahrscheinlichkeit, während der Evolution zu verändern. Das Ziel dieser Variante ist es, die Zeit zur Evolution eines vollständigen Individuums zu verkürzen, da die Substrukturen durch die vorhergegangenen Evolutionen schon die Aufgabe sinnvoll lösen können. Da die Evolutionen nicht ganz unabhängig voneinander durchgeführt wurden, sondern zuerst die Evolution für die Blattbewertung, dann die Evolution für die Problembewertung mit einem evolvierten Modul für die Blattbewertung und zuletzt die Bewertung für die Tiefenentscheidung, haben sich die später evolvierten Module auch auf die Ergebnisse der anderen Module einstellen können. Der Nachteil dieses Verfahrens ergibt sich daraus, dass die zuletzt evolvierten Module immer simuliert werden müssen und so das Gesamtindividuum durch das simulierte Verhalten der später evolvierten Module beschränkt wird. Die Möglichkeit diese Module in die weitere Evolution mit einzubeziehen sollte diesem Problem entgegenwirken.

Die vorbereitenden Experimente haben jedoch gezeigt, dass auch diese Variante zu keinem guten Ergebnissen führte, so dass sie nicht weiter verfolgt wurde. So konnten in beiden Fällen die Blattbewertungen keine signifikante Verbesserung der Fitness erreichen. Bei der dritten Variante ergab sich das gleiche Problem auch für die anderen beiden Module und die Fitness des gesamten Individuums war anschließend sehr viel schlechter als bei den Individuen, die mit der ersten Variante entwickelt wurden. Die Ergebnisse der zweiten Variante waren etwas besser als die der dritten Variante, jedoch erreichten auch sie nie die Ergebnisse der ersten Variante. Da die erste Variante gleichzeitig noch Rechenzeit einspart, wurden alle Haupttests mit dieser Variante durchgeführt.

4.3.4 Crossover der State-Space GP Struktur

Der Crossover-Operator für die State-Space Struktur hat maximal drei Möglichkeiten genetisches Material zwischen Individuen zu transportieren. Dabei ist jedoch zu beachten, mit welchen Verfahren das Individuum evolviert wird. Die drei Möglichkeiten sind nur dann sinnvoll, wenn das gesamte Individuum in einem Schritt evolviert wird.

Eine Variante des Crossoveroperators für eine State-Space Struktur ist der Austausch ganzer Substrukturen. Dabei wird eine Substruktur zufällig ausgewählt, dann zwischen den Individuen ausgetauscht (die Abbildung 4.11 zeigt dieses Vorgehen), diese Crossovervariante wird im folgenden *Strukturcrossover* genannt. Beim Strukturcrossover werden ganze Substrukturen ausgetaucht, dabei muss darauf geachtet werden, dass immer Module der gleichen Art ausgetauscht werden. Denn es macht für ein Individuum keinen Sinn ein Tiefenmodul als Blattbewertungsmodul aufzurufen. Da diese Variante des Crossovers nur Substrukturen austauscht, die möglicherweise mit anderen Modulen besser zusammenarbeiten, wird diese Variante mit einer sehr kleinen Wahrscheinlichkeit gewählt, da die Gefahr eines zu schnellen Verlustes der Diversität in der Population besteht.

Die Abbildung 4.12 zeigt die zweite Möglichkeit eines Crossovers. Sie entspricht der Graph-basierten Crossovermethode für ein Linear-Graph Individuum. Hierbei ist wieder zu beachten, dass gleiche Substrukturen gewählt werden, da jede Substruktur andere

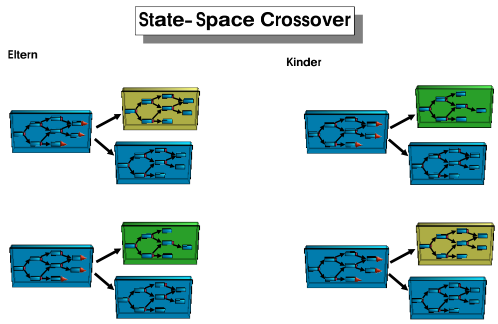

Abbildung 4.11: Die Crossoveroperation auf der Ebene des State-Space Individuums. Da das Individuum aus drei Substrukturen besteht, kann auf dieser Ebene des Crossovers nur die gesamte Substruktur getauscht werden.

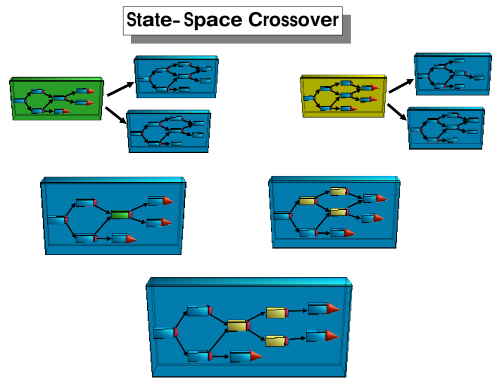

Abbildung 4.12: Die Crossoveroperation auf der Ebene einer Substruktur. Dabei wird nach der Auswahl einer Substruktur zwischen den Substrukturen der Individuen ein Crossover auf der Graphebene der Substruktur durchgeführt.

Funktions- und Terminalmengen hat und es somit zu nicht ausführbaren Individuen kommen könnte. Dabei handelt es sich um die graphbasierte Crossovervariante der Linear-Graph Struktur (siehe Abschnitt 4.2.3). Falls diese Variante des Crossovers nicht möglich ist, da die Graphen die maximale Tiefe erreicht haben, wird die Crossovervariante auf den linearen Sequenzen der Graphknoten durchgeführt.

Für ein State-Space Individuum werden alle drei Möglichkeiten des Austausches von genetischem Materials benutzt. Es wird für jede Crossovervariante vorher eine festgelegte Wahrscheinlichkeit bestimmt, mit welcher die Verfahren verwendet werden. Der Rekombinationsoperator folgt dabei dem folgenden Algorithmus.

1. Wähle gleichwahrscheinlich eine Substruktur aus.
2. Wähle mit der Wahrscheinlichkeit $prob_{sXover}$ die Strukturcrossover Methode, sonst gehe zu 3.
3. Wähle die Crossoverpunkte p_1, p_2 in beiden Substrukturen.
4. Wähle mit der Wahrscheinlichkeit $prob_{xover}$ die graphbasierte Crossovermethode, gehe zu Schritt 5.
   Wähle mit der Wahrscheinlichkeit $1-prob_{xover}$ die linearbasierte Crossovermethode, gehe zu Schritt 6.
5. Wenn die Anzahl der Ebenen des neuen Graphen die maximale Anzahl nicht überschreitet, wird das graphbasierte Crossover ausgeführt, sonst wird mit Schritt 6 weitergemacht.
6. Führe ein linearbasiertes Crossover aus.

Für den Parameter $prob_{sXover}$ wurde während der Evolution ein Wert von 2 % gewählt, für den Wert $prob_{xover}$ wurde ein Wert von 20 % festgelegt. Der Wert von 2 % für $prob_{sXover}$ führt dazu, dass ein Strukturcrossover nur sehr selten durchgeführt wird. Dadurch wird verhindert, dass der Diversitätsverlust während der Evolution zu schnell passiert und man gibt gleichzeitig den Individuen die Möglichkeiten gute Substrukturen auszutauschen. Man könnte auch noch Wahrscheinlichkeiten für die Wahl der einzelnen

Substrukturen einführen, so könnte es sinnvoll sein, zu Beginn der Evolution verstärkt die Blattbewertung zu tauschen, später die Problembewertung und erst sehr spät in der Evolution die Tiefenbewertung in das Crossover einzubeziehen. Damit könnten sich die Substrukturen, welche die Ergebnisse der anderen Substrukturen nutzen, besser auf die neuen Bewertungen einstellen. Jedoch führt dies zu mindestens drei neuen Parametern und im Fall der Gleichwahrscheinlichkeit werden sich während der Evolution natürlich gewisse Substrukturen durchsetzen, da sie zu guten Ergebnissen gelangen. Die Substrukturen passen sich durch den normalen Diversitätsverlust in der Population an und man erhält dadurch automatisch eine Anpassung des Wertes ohne ihn direkt vorzugeben [86].

Der Wert von 20 % für das graphbasierte Crossover hat sich aus den Versuchen mit dem Linear-Graph Crossoveroperator entwickelt (siehe Abschnitt 4.2.3).

4.3.5 Mutation der State-Space GP Struktur

Der Mutationsoperator für ein State-Space Individuum wird immer nur in einer Substruktur des Individuums durchgeführt, dadurch soll erreicht werden, dass sich die Substrukturen einander anpassen können. Die Wahrscheinlichkeit für die einzelnen Substrukturen ist gleich verteilt, da man hier nicht sagen kann, dass eine Substruktur stärker verändert werden muss als eine andere.

Die Mutation innerhalb der Substrukturen entspricht der Mutation eines Linear-Graph Individuums, wie es in Abschnitt 4.2.4 beschrieben wurde. Die einzige Besonderheit ist die Substruktur für die Tiefenentscheidung, da es hier noch die zusätzlichen Knoten an den Terminalen der Linear-Graph Struktur gibt. Der Mutationsoperator achtet darauf, ob er einen Terminalknoten mutiert hat. Wenn das der Fall ist, wird der Programmflussabschnitt des Knotens mit einer Wahrscheinlichkeit von 5 % mutiert. Dies wird gemacht, damit die Variation der Rekursionsaufrufe seltener durchgeführt wird, da es sonst zu starken Veränderungen des Individuums kommt, die besonders dann nicht sinnvoll sind, wenn die Evolution nur noch kleine Verbesserungen durchführen muss. Um eine Veränderung der Rekursionsaufrufe durchzuführen hat der Mutationsoperator zwei Möglichkeiten, einmal kann er die Entscheidungsfunktion variieren und zum zweiten kann er die Zugerzeugungsfunktion verändern. Während der Mutation darf immer nur eine der beiden Funktionen variiert werden, die Auswahl, welche variiert wird, ist jedoch gleich wahrscheinlich.

4.4 Empirische Analyse der Individuentypen

Die neuen Individuentypen wurden zwar mit der Intention entwickelt im GeneticChess-System verwendet zu werden, da es sich aber bei den meisten Individuen um allgemeine GP-Strukturen handelt, wurde eine empirische Analyse der Individuen auf verschiedenen Benchmarkproblemen durchgeführt. Um dabei die Performance der Strukturen bewerten zu können, wurden sie mit den Standardstrukturen Baum und linear GP verglichen.

Das *State-Space-GP* wurde auf keines dieser Benchmarkprobleme angewendet, da die Implementierung so schachspezifisch ist, dass eine Anwendung auf Klassifikations- und Regressionsprobleme nicht möglich war.

4.4.1 Testprobleme

Als Testprobleme wurden Probleme aus den Bereichen der symbolischen Regression, der Klassifikationsprobleme und ein boolesches Problem verwendet. Die Probleme wurden aus

Parameter	Wert	Beschreibung
Läufe	100	Anzahl der Läufe des GP-Systems, über die die Ergebnisse gemittelt werden.
Evaluationen	200.000	Anzahl der Fitnessauswertungen pro Lauf
Population	100/10	Die Größe der Population für einen Lauf; jeweils ein Lauf mit 100 Individuen und ein Lauf mit 10 Individuen
Crossover	0,8	Wahrscheinlichkeit, dass zwei Individuen an einem Crossover teilnehmen
Mutation	0,2	Wahrscheinlichkeit, dass ein Individuum mutiert wird
Selektion	Turnierselektion 2*2	Turnierselektion aus zwei Zweierturnieren
Initialisierung	ramped half and half	Initialisierungsart der Population
Minimale Individuengröße	10	Anzahl der Knoten, die ein Individuum minimal haben durfte
Maximale Individuengröße	200	Anzahl der Knoten, die ein Individuum maximal haben durfte
Linear-XXXX Tiefe	5	Die maximale Anzahl von Ebenen in einem Linear-Tree oder Linear-Graph Individuum
Konstanten	[-1,..,1]	Wertebereich der Konstanten
Branching-konstanten	[-10,..,10]	Wertebereich der Konstanten in den Branchingfunktionen der Individuen
Branching-funktionen	R<0, R>0, R=0, x<0, x>0, x=0	Branchingfunktionen

Tabelle 4.3: Diese Tabelle enthält alle Parameter, die für alle Tests gleich waren.

diesen verschiedenen Problemklassen gewählt, um die Analyse der neuen Individuentypen so fair wie möglich zu gestalten.

Weiterhin wurden aus Gründen der Objektivität alle Tests unter den gleichen Voraussetzungen gestartet, das heißt, die Einstellungen für wichtige Parameter des GP-Systems wurden für alle Tests gleich gewählt. Einige Parameter unterschieden sich für die einzelnen Problemklassen, wurden dann aber in diesen nicht variiert. Die Parameter, die in allen Tests nicht verändert wurden, sind in der Tabelle 4.3 zu finden.

Für alle Probleme wurde in die Funktionsmenge der Individuen die Funktion *ifThen* benutzt. Diese Funktion implementiert einen bedingten Sprung im linearen Genom der Individuen und ermöglicht eine Verzweigung im Baum GP. Diese Funktion wurde in die Funktionsmenge aufgenommen, um bei der empirischen Analyse der neuen Individuentypen, im Vergleich zu den Standard GP-Strukturen, diesen die Möglichkeit zu geben, ebenfalls den eigenen Programmfluss zu modifizieren. Wichtig ist noch hervorzuheben, dass die Parameter für Crossover und Mutation denen entsprechen, die von Koza für Baum GP verwendet werden.

4.4.1.1 Symbolische Regression

Als Benchmarks für die symbolische Regression wurden Probleme ausgewählt, die häufig in Publikationen als Benchmarks herangezogen werden.

Die Fitness eines Individuums p ist für die Testprobleme definiert als die Summe der quadratischen Fehler über alle Ausgabewerte. Für eine gegebene Funktion $f(x)$ und den durch das Individuum berechneten Wert $p(x)$ ist die Fitness des Individuums p wie folgt definiert:

$$\text{fitness}(p) = \frac{\sum_{i=1}^{n}(p(x_i) - f(x_i))^2}{n}.$$

Als Benchmarkprobleme für die symbolische Regression wurden die folgenden Probleme benutzt:

- Sinus im Bereich von $[0, 2\pi]$ mit 20 Fitnesscases. Die Fitnesscases wurden uniform aus dem Intervall genommen und die beiden Endpunkte des Intervalls wurden auch in die Menge aufgenommen.
- Sinus im Bereich von $[0, 4\pi]$ mit 40 Fitnesscases. Die Auswahl der Fitnesscases entsprach den Nebenbedingungen für das Sinusproblem mit einer Periode.
- Rastrigin, $f(x) = x^2 - 5 * cos(2\pi * x)$, im Bereich von $[-4, 4]$ und 40 Fitnesscases.

Die Parameter, die in allen Läufen zur Regression nicht verändert wurden, sind in der Tabelle 4.3 zu finden. Als Funktionsmenge wurden die Funktionen $+, -, *, /$ und ifThen verwendet.

4.4.1.2 Klassifikationsprobleme

Als Benchmarks für die Klassifikationsprobleme wurden die folgenden Probleme verwendet:

- Das Chain Problem, ein Klassifikationsproblem besteht daraus, einen Punkt im Raum einer von zwei Klassen zuzuordnen. Die Punkte der beiden Klassen sind in zwei Ringen im Raum angeordnet, wie Abbildung 4.13 verdeutlicht.
- Das Spiral Problem von Koza besteht aus zwei Spiralen in der Ebene. Eine Spirale entspricht der Klasse eins und die zweite Spirale der Klasse zwei, die Abbildung 4.14 zeigt das Spiralproblem mit der Klassenzugehörigkeit.
- Das 11-Multiplexer Problem, das Problem ist eigentlich ein boolesches Problem, jedoch entspricht die Fitnessfunktion der der Klassifikationsprobleme. Das Problem verwendet jedoch eine andere Funktionsmenge, die nur aus booleschen Funktionen besteht.

Die Fitness eines Individuums ist die Fehlklassifikation auf den Fitnesscases in Prozent, das heißt, die beste Fitness ist 0. Für ein gegebenes Klassifikationproblem $k(x)$ und den durch das Individuum p berechneten Wert ist die Fitness des Individuums p, als die Anzahl der Fehlklassifikationen definiert.

$$\text{fitness}(p) = \sum_{i=0}^{max} \begin{cases} 1 & \text{wenn die Eingabe richtig klassifiziert wurde,} \\ 0 & \text{sonst.} \end{cases} \tag{4.1}$$

Die Parameter, die in allen Läufen zur Klassifikation nicht verändert werden, entsprechen denen in Tabelle 4.3. Als Funktionsmenge wurden für das Chain- und Spiral Problem die Funktionen $+, -, *, /$, ifThen, Sinus und Cosinus verwendet, für das Multiplexerproblem die booleschen Funktionen *und, oder, nicht* und *exclusive oder*.

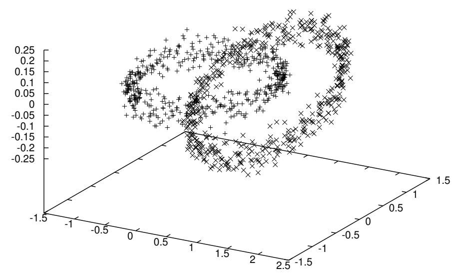

Abbildung 4.13: Die Abbildung zeigt das Chain Problem mit den beiden Klassen, die durch zwei Ringe im Raum dargestellt werden. Das Problem wurde in dem Artikel [19] zum ersten Mal verwendet.

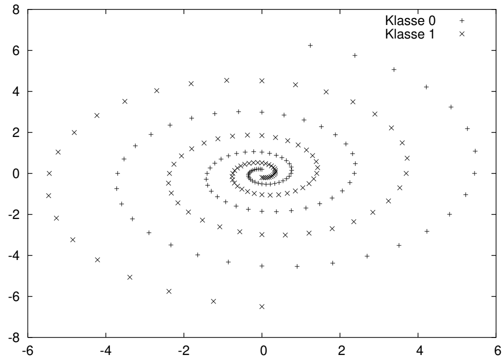

Abbildung 4.14: Die Abbildung zeigt das Spiral Problem, welches von Koza in [73] veröffentlicht wurde. Das Problem besteht aus zwei Spiralen, die zu zwei verschiedenen Klassen gehören.

4.4.2 Experimentelle Ergebnisse

Im folgenden Abschnitt werden die Ergebnisse der verschiedenen GP-Strukturen auf den sechs Benchmarkproblemen diskutiert. Alle Graphiken zeigen den Mittelwert der Fitness des besten Individuums von 100 Läufen. Für jedes Problem wurden 200.000 Evaluationen durchgeführt, die Populationsgröße variierte dabei zwischen 10 und 100 Individuen. Die Populationsgröße wird bei jedem Plot angegeben.

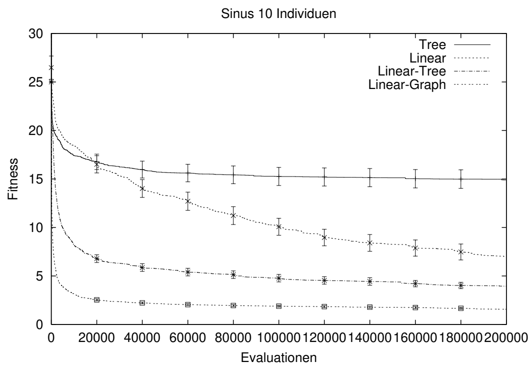

Abbildung 4.15: Die Kurven beschreiben den Mittelwert der Fitness über 100 Läufe mit jeweils 10 Individuen pro Pool. Die guten Fitnesswerte für die Linear-Tree und -Graph Strukturen und ihre kleinen Standardabweichungen im Vergleich zu den linearen und Baumstrukturen sind sofort zu erkennen.

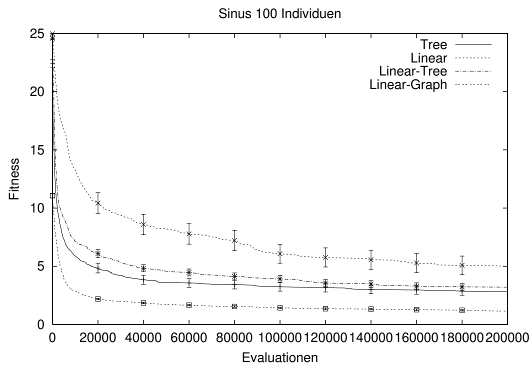

Abbildung 4.16: Die 100 Läufe wurden mit 100 Individuen pro Pool durchgeführt. Alle Strukturen zeigen einen typischen Fitnessverlauf für evolutionäre Verfahren. Es zeigt sich hier, dass das Baum GP sehr viel bessere Ergebnisse liefert als mit 10 Individuen und ungefähr so gut ist wie die Linear-Tree Struktur.

In den Abbildungen 4.15 und 4.16 zeigen die Fitnesswerte der verschiedenen Strukturen für das Sinusproblem mit 10 beziehungsweise 100 Individuen pro Pool. In beiden Fällen ist die Anzahl der Fitnessevaluationen gleich, trotzdem zeigt sich für das Baum GP und das lineare GP ein Unterschied während der Evolution und in den erreichten Ergebnissen. Die Linear-Tree und Linear-Graph Strukturen erreichen in beiden Fällen ähnliche Ergebnisse, dies ist ein deutlicher Hinweis, dass diese Struktur von der Populationsgröße nicht stark abhängt und die Diversität während der Evolution auch in kleinen Populationen aufrecht erhalten kann. Sehr gut sind auch die kleinen Standardabweichungen im Vergleich zu den linearen und Baumstrukturen. Hierauf wird noch einmal später eingegangen, wenn die Ergebnisse der letzten Generation für alle Läufe analysiert wird. Interessant ist hier auch die Fitnessentwicklung des linearen GP's, in der Abbildung 4.15, denn es zeigt nach 200.000 Evaluationen noch kein Konvergenzverhalten wie die anderen Strukturen. Das deutet darauf hin, dass es noch zu Verbesserungen kommen würde, wenn man die Evolution weiter laufen lassen würde.

In der Abbildung 4.16 zeigen alle Strukturen einen typischen Fitnessverlauf für evolutionäre Verfahren. Es zeigt sich nun, dass das Baum GP sehr viel bessere Ergebnisse liefert als mit 10 Individuen und ungefähr so gut ist wie die Linear-Tree Struktur. Auch die Standardabweichung der Ergebnisse ist viel kleiner geworden als mit nur 10 Individuen. Dies ist ein deutlicher Hinweis darauf, dass Baum GP größere Populationen benötigt als die anderen Strukturen, dies zeigt sich immer wieder im Verlauf der Untersuchung. Die Linear-Graph Struktur erreichte auch mit 100 Individuen die besten Ergebnisse im Vergleich der verschiedenen Strukturen. Die Standardabweichung hat sich von 0,0928409 auf 0,0770319 verringert, was im Vergleich zum Baum GP gering ist, denn dort war der Schritt von 0,960098 zu 0,357485 zwar größer, bleibt aber absolut gesehen sehr viel schlechter als beim Linear-Graph GP.

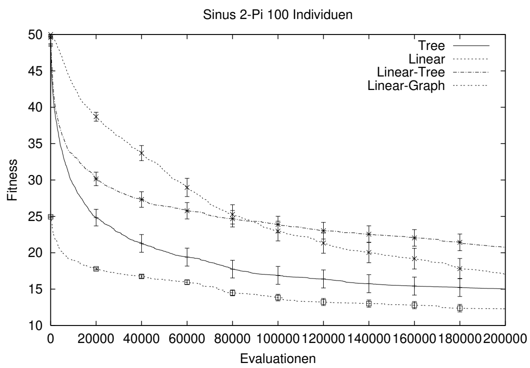

Abbildung 4.17: Dieser Plot zeigt die Fitness für die Regression einer Sinuskurve über 4π, also 2 Perioden. Auch hier zeigt die Fitnessentwicklung des linearen GP's nach 200.000 Evaluationen noch kein Konvergenzverhalten wie die anderen Strukturen.

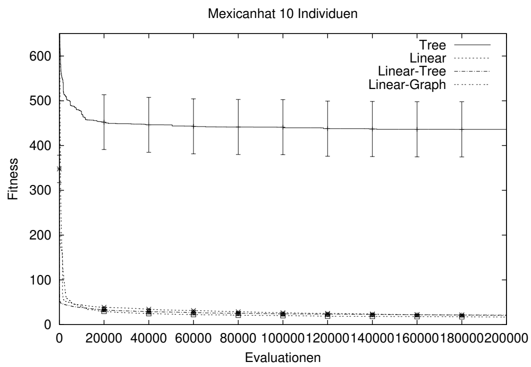

Abbildung 4.18: Der Plot zeigt das Verhalten der GP Strukturen für das Rastrigin Problem. Hier ist die Fitness der Baumstruktur im Vergleich zu allen anderen Strukturen sehr schlecht, so dass der weitere Plot 4.19 die Ergebnisse der anderen Strukturen vergrößert.

Dieser Unterschied ist sehr wichtig, da das Ziel der Dissertation die Evolution von Schachprogrammen ist und die Evolution von Schachprogrammen durch die Fitnessbewertung sehr rechenzeitintensiv ist. Bei den Ergebnissen der Schachprogramme ist auch das absolut beste Programm interessant und nicht die durchschnittlichen Ergebnisse. Somit sind geringe Variationen in den Ergebnissen verschiedener Läufe gewünscht.

In der Abbildung 4.17 sieht man die Ergebnisse der Strukturen auf das Sinusproblem über 2 Perioden, mit 100 Individuen pro Lauf. Die Ergebnisse mit 10 Individuen werden hier nicht gezeigt, da bis auf die Linear-Graph Struktur alle anderen Strukturen an dieser Aufgabe gescheitert sind. Interessant ist hier jedoch die Ähnlichkeit zu dem Ergebnis für das einfache Sinusproblem mit 10 Individuen, siehe Abbildung 4.15. Hier schneidet nur das Baum GP deutlich besser ab, ähnlich dem Ergebnis für das einfache Sinusproblem mit 100 Individuen. Aber das Verhalten der linearen Struktur ist direkt vergleichbar mit den Ergebnissen mit 10 Individuen, dies ist ein erster Hinweis darauf, dass man zur

Lösung von schweren Problemen größere Populationen benötigt. Gleichzeitig kann man aber auch durch andere Strukturen den evolutionären Prozess unterstützen und somit wieder mit kleineren Populationen arbeiten.

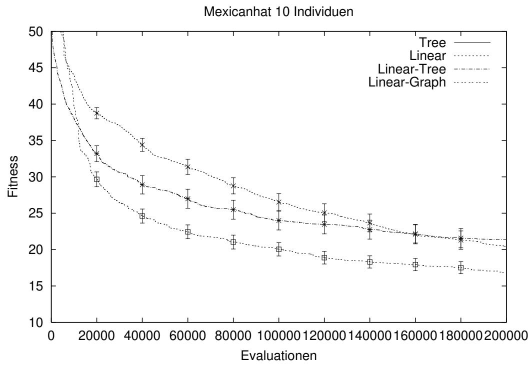

Abbildung 4.19: Die Ergebnisse der Evolution mit 10 Individuen zeigen für die lineare Struktur wieder ein anderes Verhalten als für 100 Individuen. Die Kurve fällt über die ganze Zeit viel gleichmäßiger und zeigt kein Konvergenzverhalten auf dem Bereich mit 200.000 Evaluationen. Das beste Ergebnis mit der kleinsten Standardabweichung erreicht wieder die Linear-Graph Struktur.

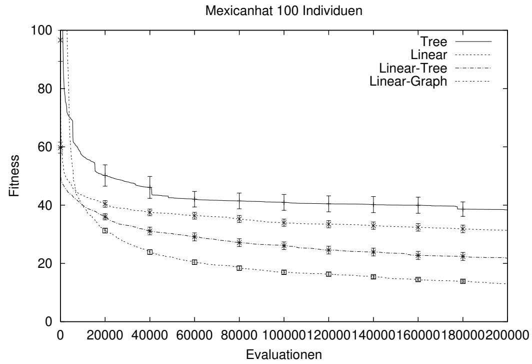

Abbildung 4.20: Die Evolution mit 100 Individuen zeigt für das Rastrigin Problem ein ganz anderes Ergebnis als mit nur 10 Individuen. Jetzt ist auch die Baumstruktur konkurrenzfähig. Sie erreicht zwar die schlechtesten Ergebnisse, der Unterschied ist aber nicht mehr so dramatisch. Auch hier erreicht die Linear-Graph Struktur wieder die besten Ergebnisse.

Die Abbildung 4.18 zeigt das Verhalten der GP Strukturen für das Rastrigin Problem. Hier ist die Fitness der Baumstruktur im Vergleich zu allen anderen Strukturen sehr schlecht, so dass der weitere Plot 4.19 die Ergebnisse der anderen Strukturen vergrößert. Gleichzeitig zur schlechten Fitness der Struktur ist auch die Standardabweichung für die Baumstruktur im Vergleich sehr schlecht. Die Ergebnisse der Evolution mit 10 Individuen, wie sie in Abbildung 4.19 gezeigt werden, zeigt für die lineare Struktur wieder ein anderes Verhalten als für 100 Individuen. Die Kurve fällt hier über die ganze Zeit viel gleichmäßiger aus und zeigt kein Konvergenzverhalten im Bereich von 200.000 Evaluationen. Das beste Ergebnis mit der kleinsten Standardabweichung erreicht wieder die Linear-Graph Struktur. Die Ergebnisse der Linear-Tree Struktur sind sogar etwas schlechter als die der linearen Struktur und auch die Standardabweichung ist für die Linear-Tree Struktur nicht so gut wie bei den anderen Problemen. Ein Grund hierfür liegt in dem Problem selber, das Rastrigin Problem hat auf dem betrachteten Intervall sehr viele Schwingungen, so dass die Baumstruktur ein Polynom hohen Grades evolvieren muss, um die Problemstellung zu lösen. Für die Linear-Tree Struktur ist hier die maximale Tiefe des Baumes ein Problem, bei größeren Tiefen sind hier bessere Ergebnisse möglich, für den Vergleich wurde dies aber nicht gemacht. Die Linear-Graph Struktur kann hier die Vorteile der Graphstruktur gegenüber der Baumstruktur im Programfluss der Individuen nutzen um kompaktere Individuen zu erzeugen.

Die Evolution mit 100 Individuen zeigt für das Rastrigin Problem ein ganz anderes Ergebnis als mit nur 10 Individuen. Die Abbildung 4.20 zeigt jetzt, dass auch die Baumstruktur konkurrenzfähig ist. Sie erreicht zwar die schlechtesten Ergebnisse, der Unterschied zu den anderen Strukturen ist aber nicht mehr so dramatisch. Auch hier erreicht die Linear-Graph Struktur wieder die besten Ergebnisse. Gut sind die großen Unterschiede in der Standardabweichung zwischen Linear-Graph (0,48) und Baum (2,50) in der Abbildung zu erkennen.

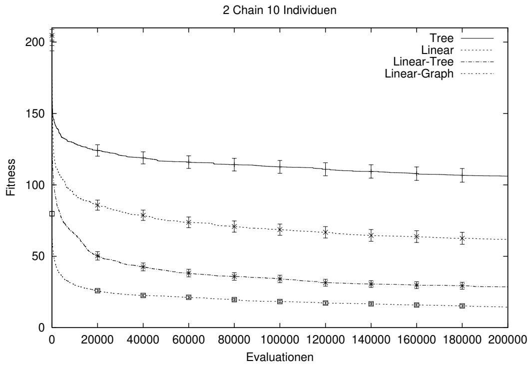

Abbildung 4.21: Die Abbildung zeigt die Fitness für das 2 Chain Problem mit 10 Individuen. Keine der Fitnesskurven zeigt ein außergewöhnliches Verhalten. Das Baum GP schneidet in diesem Versuch am schlechtesten ab und das Linear-Graph GP am besten.

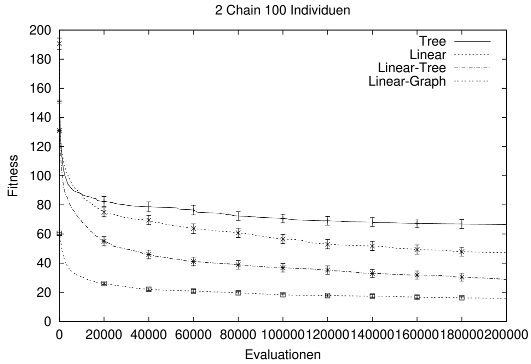

Abbildung 4.22: Die Abbildung zeigt die Fitness für das 2 Chain Problem mit 100 Individuen. Alle Verläufe der Fitnesskurven sehen typisch aus. Die Standardabweichung hat sich für alle Strukturen im Vergleich zu 10 Individuen verkleinert.

Die Abbildungen 4.21 und 4.22 zeigen die Ergebnisse für das Klassifikationsproblem Chain. Im direkten Vergleich der Abbildungen erkennt man kaum einen qualitativen Unterschied zwischen den Fitnessverläufen der einzelnen Strukturen. Am stärksten kann man noch einen Unterschied bei der Baum GP Struktur erkennen, die bei 100 Individuen sehr viel bessere Fitnesswerte erreicht und auch die Standardabweichung über alle Fitnessläufe reduziert sich erheblich. Deutlich zeigt die Linear-Graph Struktur auch für dieses Problem die Überlegenheit gegenüber den anderen Strukturen, diese kann sowohl durch die bessere Fitness als auch durch die kleinere Standardabweichung gesehen werden.

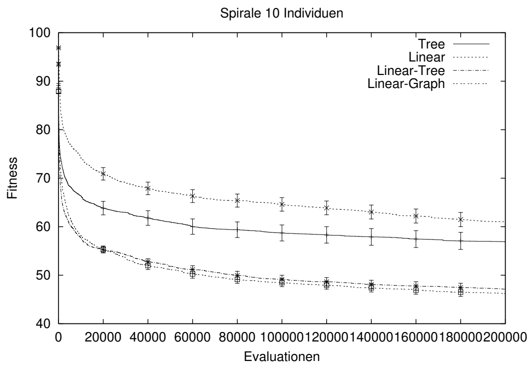

Abbildung 4.23: Die Fitness für das Spiral Problem mit 10 Individuen zeigt ein ähnliches Bild wie für das Chain Problem mit 10 Individuen. Das Besondere hier ist das gute Abschneiden der Linear-Tree Struktur, die hier etwa die gleichen Ergebnisse wie die Linear-Graph Struktur erreicht.

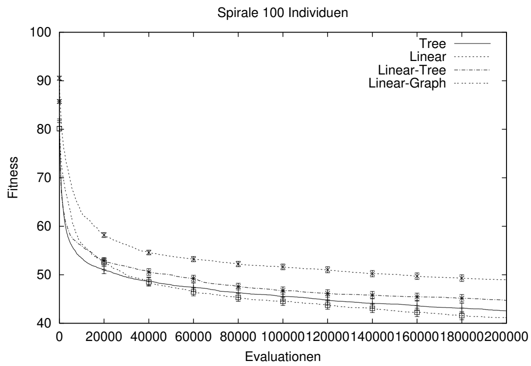

Abbildung 4.24: Die Fitnessverläufe für das Spiral Problem mit 100 Individuen unterscheiden sich stark von den mit 10 Individuen. Hier ist besonders das gute Abschneiden der Baumstruktur im Vergleich zu dem Lauf mit 10 Individuen zu erkennen.

Die Abbildungen 4.23 und 4.24 zeigen die Fitnessverläufe für das Spiral Problem mit 10 und 100 Individuen. Der Unterschied zwischen den Läufen mit 10 und 100 Individuen ist für dieses Problem ähnlich unterschiedlich wie für das Sinus Problem. Für das Sinus und das Spiral Problem verbessert sich die Fitness der Baumstruktur so stark, dass es im Durchschnitt mit 100 Individuen die zweitbeste Fitness erreicht. In diesem Fall ist sogar der Unterschied der Fitness zwischen der Baum (42,54) und Linear-Graph Struktur (41,12) so gering, dass man aufgrund der Standardabweichung von 1,01828 (Baum) und 0,856771 (Linear-Graph) das Ergebnis der Linear-Graph Struktur nicht als eindeutig besser bezeichnen kann.

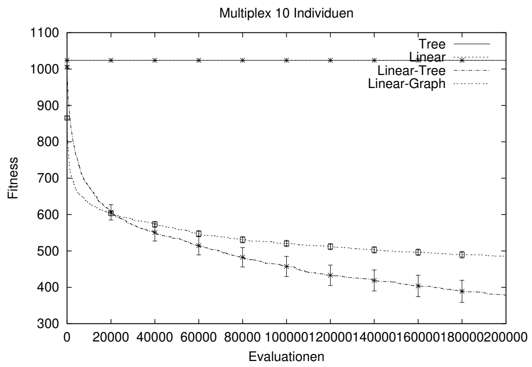

Abbildung 4.25: Die Fitness für das 11-Multiplexer Problem mit 10 Individuen. Die lineare und Baumstruktur können keine Lösung finden.

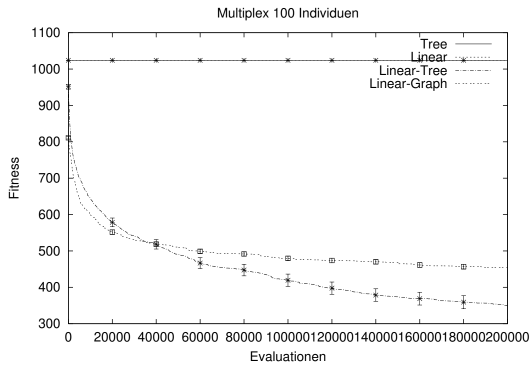

Abbildung 4.26: Die Fitness für das 11-Multiplexer Problem mit 100 Individuen. Die lineare und Baumstruktur können keine Lösung finden.

Ein ganz anderes Ergebnis präsentiert sich für das Multiplexer Problem, hier können sowohl die lineare als auch die Baum Struktur keine Lösung für das Problem finden. Dies gilt sowohl für Populationen mit 10 als auch mit 100 Individuen. Die Abbildungen 4.25 und 4.26 zeigen die Ergebnisse für die Läufe. Es hat sich gezeigt, dass sowohl die lineare als auch die Baumstruktur für das Multiplexer Problem Lösungen finden können, die auch gut sind, jedoch benötigen sie dazu größere Populationen. So können beide Strukturen mit einer Population von 200 Individuen zu Lösungen kommen, besser sind jedoch Populationen von 500 Individuen, da dann der Anteil der guten Lösungen größer ist als der Anteil der schlechten Lösungen. Für dieses Problem generiert die Linear-Tree Struktur die besten Ergebnisse, die Linear-Graph zeigt hier zwar auch gute Ergebnisse, aber herausragend ist die geringe Standardabweichung in den Ergebnissen.

Die folgenden Graphiken zeigen immer die Fitness des besten Individuums in der letzten Generation für jeden der 100 Läufe. Die Ergebnisse wurden hierzu vorher sortiert, sie zeigen also keine zeitliche Abfolge. Durch diese Graphiken kann man nun die Daten sehen, aus der sich die Standardabweichung berechnet.

Die Abbildung 4.27 und 4.28 zeigen die Ergebnisse der Läufe für das Sinus Problem. Es zeigt sich immer, dass die Variation der Ergebnisse für die Baumstruktur am größten ist im Vergleich zu den anderen Strukturen, was auch zu den hohen Standardabweichungen in den Fitnessplots führt. Gleichzeigt sieht man auch die erhebliche Verringerung bei der Variation der Ergebnisse für die Baumstruktur in den Abbildungen mit Populationen von 100 Individuen.

Es hat sich sogar gezeigt, dass die Baumstruktur für das Rastrigin Problem mit 100 Individuen Lösungen finden kann, die sehr gut sind, jedoch sind dies sehr wenige und gleichzeitig werden auch sehr viele schlechte Lösungen evolviert, siehe Abbildung 4.30. Ein ähnliches Argument gilt auch für die Linear-Tree Struktur, sie erzeugt hier die besten Lösungen, aber auch mehr schlechtere als die Linear-Graph Struktur.

Für das Chain Problem kann die Baumstruktur nicht so gute Ergebnisse liefern wie für das Rastrigin Problem, hier ist nicht nur die Varianz der Ergebnisse größer als für die Linear-Tree und Linear-Graph Strukturen, sondern auch die besten Lösungen sind viel schlechter, siehe Abbildung 4.31 und 4.32. Beim Spiral Problem sind die Unterschiede zwischen den einzelnen Ergebnissen nicht so groß, jedoch kann man auch hier die geringe Varianz in den Lösungen der Linear-Graph Struktur erkennen, siehe Abbildung 4.33 und 4.34.

Die Abbildungen 4.35 und 4.36 zeigen das extreme Verhalten der linearen und Baumstruktur noch einmal sehr deutlich. Gleichzeitig sieht man jetzt aber auch, dass die Linear-Tree Struktur sehr viel bessere Lösungen schafft als die Linear-Graph Struktur, so kann die Linear-Tree Struktur mehrmals die optimale Lösung erreichen, was der Linear-Graph Struktur nicht gelingt.

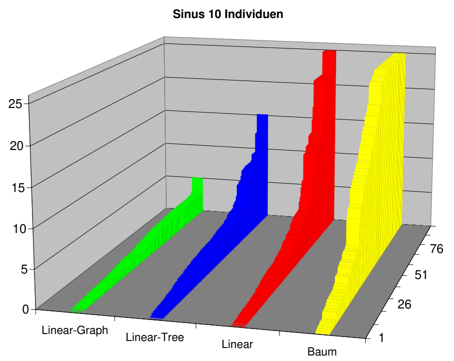

Abbildung 4.27: Die Fitness des besten Individuums in der letzten Generation von allen Strukturen. Für das Sinus Problem mit 10 Individuen.

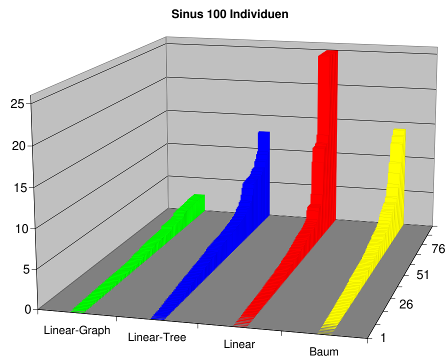

Abbildung 4.28: Die Fitness des besten Individuums in der letzten Generation von allen Strukturen. Für das Sinus Problem mit 100 Individuen.

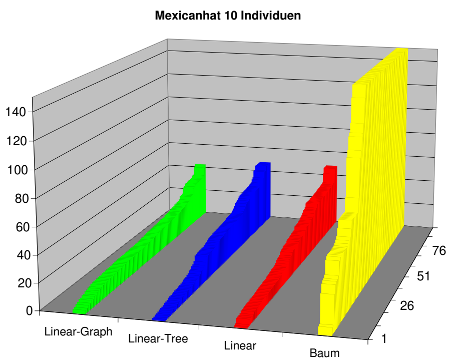

Abbildung 4.29: Das Rastrigin Problem mit 10 Individuen.

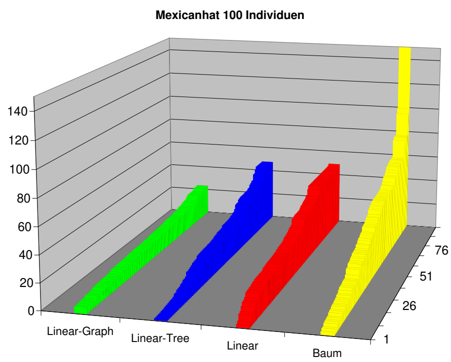

Abbildung 4.30: Das Rastrigin Problem mit 100 Individuen.

4.4.3 Analyse der Linear-Tree und Linear-Graph Struktur

Die Analyse der Linear-Tree und Linear-Graph Struktur zeigte zwei Verhaltensweisen der neuen Strukturen. Eine bezieht sich auf das Verhalten der Individuenstruktur während der Evolution und die zweite auf den Aufbau der Individuen selber. Es zeigte sich nämlich bei dem Aufbau der Individuen, dass in verschiedenen Knoten des Baums, aber auch des Graphen die gleichen linearen Segmente existierten. Dies hatte einerseits zur Folge, dass die Möglichkeit der Verzweigung umgangen wurde, da beide Nachfolger eines Knotens den gleichen Code enthielten, und anderseits, dass der Programmfluss virtuell wieder zusammenfloss. Denn wenn die Blätter des Baumes die gleichen Knoten enthielten, hätte

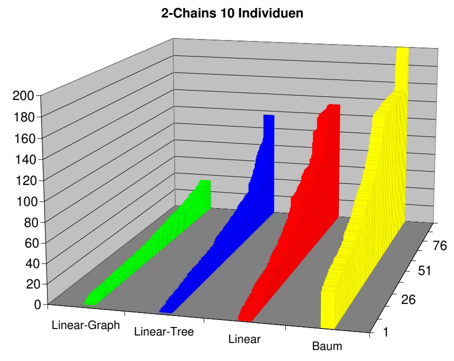

Abbildung 4.31: Das Chain Problem mit 10 Individuen.

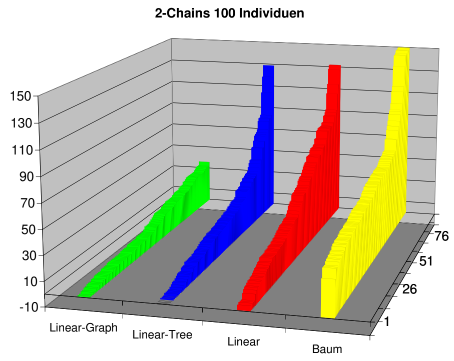

Abbildung 4.32: Das Chain Problem mit 100 Individuen.

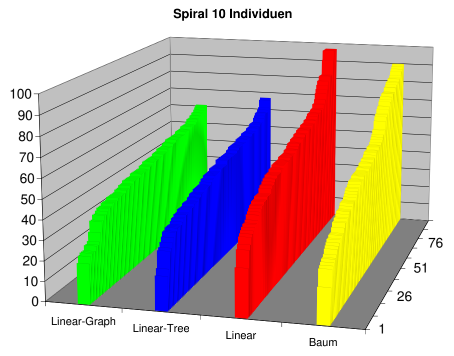

Abbildung 4.33: Das Spiral Problem mit 10 Individuen.

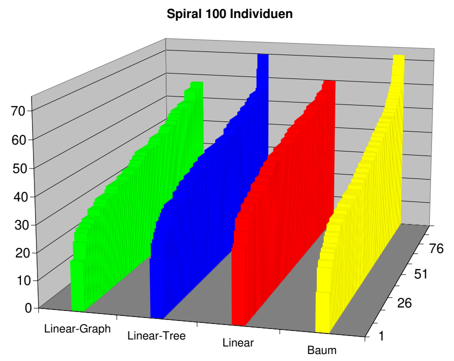

Abbildung 4.34: Das Spiral Problem mit 100 Individuen.

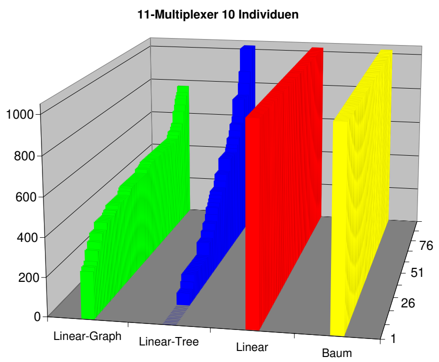

Abbildung 4.35: Das Multiplexer Problem mit 10 Individuen.

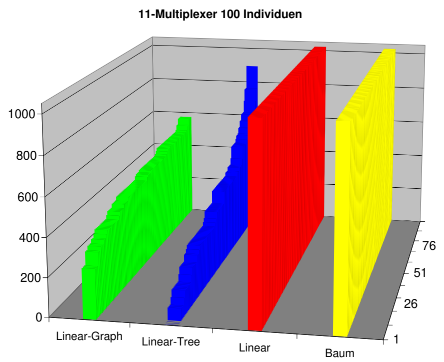

Abbildung 4.36: Das Multiplexer Problem mit 100 Individuen.

man sie auch zu einem Knoten zusammenfassen können und hätte somit einen Graphen erzeugt. Dies war auch einer der Gründe, die Linear-Graph Individuenstruktur zu implementieren. In den meisten Individuen wurden allerdings die Verzweigungen genutzt. Hierbei ist hervorzuheben, dass die Verzweigungen sowohl für Klassifikationsprobleme als auch für Regressionsprobleme von Vorteil sind. Wenn man die Ergebnisse der Individuen während der Evolution für ein Regressionsproblem verfolgt, kann man sofort den Unterschied zwischen Linear oder Baum GP Programmen und den Linear-Tree Programmen erkennen. Anhand der Abbildungen 4.37 und 4.38 kann man die Unterschiede der In-

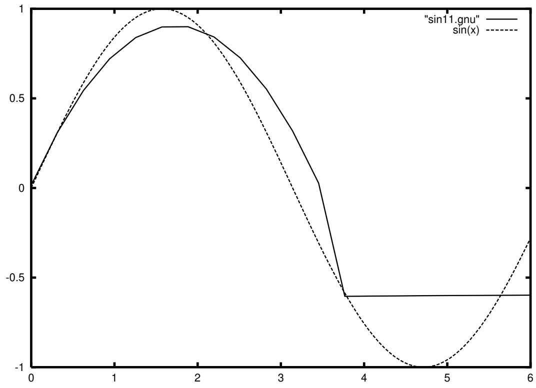

Abbildung 4.37: Das Linear-Tree Individuum evolviert zuerst die erste Hälfte der Sinusfunktion und beginnt erst nach der Generation 11, die zweite Hälfte der Funktion zu approximieren.

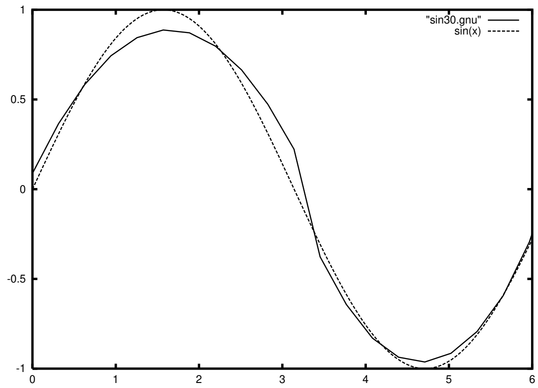

Abbildung 4.38: In dieser Graphik sieht man das Ergebnis der Evolution nach 30 Generationen, nun ist auch die zweite Hälfte der Funktion approximiert worden.

dividuenstrukturen Linear-Tree und Linear-Graph bezüglich der Evolution im Vergleich zu den anderen Strukturen erkennen. Es wurde exemplarisch die Entwicklung der Sinuskurve gewählt, da man hier sehr gut den Unterschied in dem Evolutionsprozess zwischen der Linear-Tree und den anderen Strukturen erkennen kann. So wird in diesem Beispiel von einem Linear-Tree Individuum zuerst die erste Hälfte der Sinusfunktion approximiert und die Evolution beginnt erst nach der Generation 11, die zweite Hälfte der Funktion zu approximieren. Bei den anderen beiden Strukturen erkennt man, dass verschiedene Polynome verwendet werden, um die Sinusfunktion zu approximieren. Wenn man sich nun das Individuum selbst betrachtet, sieht man, dass ein Linear-Tree oder Linear-Graph Individuum oft bei einer Branchingentscheidung überprüft, ob die Eingabe ungefähr größer als 3 ist. Dies kann man auch deutlich in der Abbildung 4.37 erkennen, da sich an dieser Stelle eine Unstetigkeit gebildet hat. Weiter kann man erkennen, dass die Evolution der zweiten Hälfte der Sinusfunktion in einer kürzeren Zeit geschieht als für die erste Hälfte. Man kann auch einen qualitativen Unterschied in der Entwicklung erkennen. So beginnt die Approximationsphase für die zweite Hälfte der Sinusfunktion sehr schnell mit einer Kurve, die der Kurve der ersten Hälfte der Funktion entspricht.

Man kann neben dem objektiven Unterschied in der Evolution der verschiedenen Strukturen, der sich durch die unterschiedliche Entwicklung der Fitness ergibt, auch einen eher subjektiven Unterschied erkennen, wie er oben am Beispiel der Evolution für das Sinusproblem beschrieben wurde. Dieser subjektive Unterschied gibt aber auch einen Hinweis auf die *Building Blocks*, die während der Evolution zwischen den Individuen getauscht werden. Der Unterschied zu der *Building Block Hypothese*, wie sie in [53] und [35, 119] beschrieben werden, ist, dass die Blöcke in den Linear-Tree und -Graph Strukturen durch den Aufbau des Individuums selbst vorgegeben sind. Hier werden die Linear Segmente als *Building Block* benutzt, was durch die Struktur und durch die Rekombination unterstützt wird.

4.4.4 Analyse des Crossover Operators

Da der Crossover Operator der Linear-Tree und Linear-Graph Strukturen aus zwei Phasen besteht, ergibt sich automatisch der Parameter $prob_{xover}$, der angibt, mit welcher Wahrscheinlichkeit eine linearbasierte Crossovermethode oder eine baumbasierte- bzw. graphbasierte Crossovermethode verwendet wird (siehe Abschnitt 4.2.3 und 4.1.3). Da es sich bei beiden Strukturen um neue GP Strukturen handelt, ist nicht genau klar, welchen Wert dieser Parameter haben sollte. Eine hohe Wahrscheinlichkeit für $prob_{xover}$ hat zur Folge, dass hauptsächlich das baum- bzw. graphbasierte Crossover angewendet wird. Dadurch werden große Teilsegmente des Individuums beim Crossover ausgetauscht und der Programmfluss des Individuums verändert sich durch die neuen Elemente stark. Es stellt sich automatisch die Frage, ob es einen optimalen Wert für diesen Parameter gibt und wie er lautet. Hierzu wurden für das Sinus- und das 2-Chain Problem 100 Läufe mit Werten von 0% bis 100% in 10% Schritten durchgeführt. Da es nur um die Analyse des Crossoveroperators geht, wurde die Mutation für diese Untersuchung zuerst nicht verwendet.

In den Abbildungen 4.39 und 4.40 sind die Ergebnisse für die Linear-Graph Struktur für die beiden Probleme dargestellt. In den Abbildungen sind nur die Fitnesskurven für die Parametereinstellung von 0, 30, 50 und 70 % dargestellt, da sie unter allen Ergebnissen das Minimum, das Maximum und durchschnittliche Werte darstellen. In der Abbildung 4.41 sind noch mal die Fitnesskurven mit allen Parametern für das Sinusproblem abgebildet. In der Abbildung kann man deutlich ein Cluster erkennen, in dem fast alle Einstellungen liegen. Die besten Einstellungen für den Parameter sind 20 und 30% und die schlechteste Einstellung ist 0%. Dieses Bild ähnelt dem für das Chain-Problem, nur die Werte für die Extremwerte sind andere, so ist die beste Einstellung für den Parameter 50% und die schlechteste 30%. Anhand dieser beiden Beispiele erkennt man direkt, dass es keine optimale Einstellung gibt, die von der Problemstellung unabhängig ist. Daher wäre eine Adaptation des Wertes während der Evolution auf der Grundlage dieser Ergebnisse die beste Lösung. Es hat sich jedoch gezeigt, dass der Wert des Parameters unkritisch ist, wenn man bei den Evaluationen die Mutation mitverwendet. Denn dann liegen alle Werte so nahe zusammen, dass man keinen signifikanten Unterschied für die verschiedenen Werte erkennen kann. Somit wird der Einfluss, den der Parameter auf die Evolution hat, durch die Mutation wieder aufgehoben. Gleichzeitig verbessern sich die Ergebnisse der Evolution, so erreichen die Evolutionen ohne Mutation im Durchschnitt im besten Fall eine Fitness von $1,7$ und mit Mutation von $1,1$. Dass der Mutationsoperator die Einstellung der Crossoverparameter überschattet, ist nicht leicht zu erklären. Für die Linear-Tree Struktur ergibt sich ein ähnliches Bild bezüglich des Crossover- und Mutationsoperators: Wenn beide Operatoren aktiv sind, kann man keinen Unterscheid zwischen den verschiedenen Parametereinstellungen für den Crossoveroperator erkennen. Die Abbildung 4.42 zeigt daher das Verhalten der Linear-Tree Struktur für das Sinusproblem ohne Mutation für die Parametereinstellungen 10%, 50% und 90%. Hier zeigt sich deutlich, dass 90% der beste Wert für dieses Problem wäre, wie aber schon erwähnt ergeben sich Unterschiede für die verschiedenen Einstellungen, wenn der Mutationsoperator auch aktiv ist.

Abschließend kann man für die Crossoveroperatoren der Linar-Tree und Linear-Graph Strukturen sagen, dass die Verwendung des Mutationsoperators wichtig ist, wenn man gute Ergebnisse erreichen möchte. Der Parameter $prob_{xover}$ hat nur bei einem deaktivierten Mutationsoperator Einfluss auf die Evolution, da die Ergebnisse für die verschiedenen Parametereinstellungen aber nicht eindeutig sind, ist es schwer hierfür eine Antwort zu finden. So führte ein kleiner Parameterwert von 20% für das Chain-Problem zu schlechten Ergebnissen, wobei er beim Sinus-Problem zu guten Ergebnissen führte. Obwohl der Crossoveroperator für verschiedene Werte bei verschiedenen Problemen zu den besten Lösungen gelangte, wird dies durch den Mutationoperator ausgeglichen. Man kann also nicht behaupten, dass der Mutationsoperator nur eine geringere Wahrscheinlichkeit des linearbasierten Crossovers ausgleicht, denn er kann auch eine geringere Wahrscheinlichkeit des graphbasierten Crossovers ausgleichen. Man kann sich aber anhand der Arbeitsweise des Mutationsoperators klar machen, dass er sowohl das linearbasierte als auch das graphbasierte Crossover ersetzen kann (zur Arbeitsweise siehe Abschnitt 4.2.4 für das

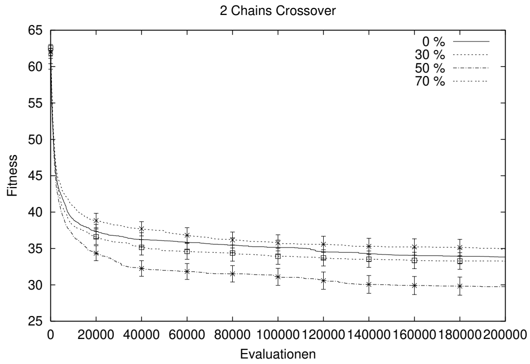

Abbildung 4.39: Die Abbildung zeigt die Fitness für das 2 Chain Problem ohne Mutation für verschiedene Werte des Crossoverparameters $prob_{xover}$ für die Linear-Graph Struktur.

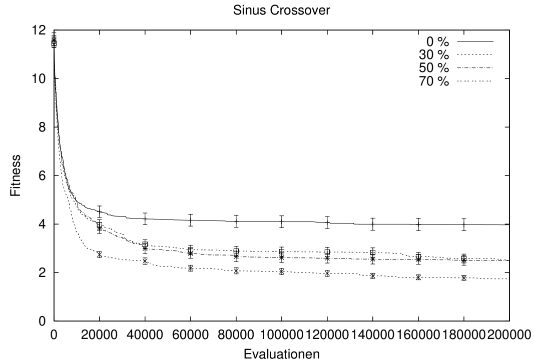

Abbildung 4.40: Die Abbildung zeigt die Fitness für das Sinusproblem ohne Mutation für verschiedene Werte des Crossoverparameters $prob_{xover}$ für die Linear-Graph Struktur.

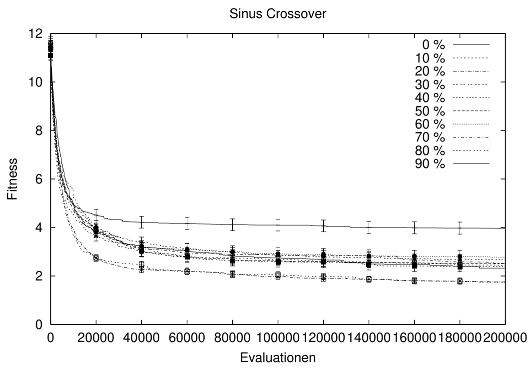

Abbildung 4.41: Die Abbildung zeigt die Fitness für das Sinusproblem ohne Mutation für alle getesteten Werte des Crossoverparameters $prob_{xover}$ für die Linear-Graph Struktur. Die besten Ergebnisse werden für das Problem mit 20 oder 30%, das schlechteste Ergebnis mit 0% Wahrscheinlichkeit erreicht.

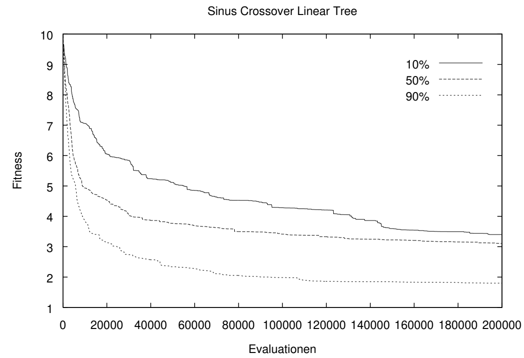

Abbildung 4.42: Die Abbildung zeigt die Fitness für das Sinusproblem ohne Mutation für alle getesteten Werte des Crossoverparameters $prob_{xover}$ für die Linear-Tree Struktur. Die Kurve 'Mutation' zeigt zum Vergleich das Verhalten ganz ohne Crossover an.

linearbasierte und Abschnitt 4.1.4 für das graphbasierte Crossover). Ein hoher Anteil des linearbasierten Crossovers führt während der Evolution dazu, dass sich die Baum- oder Graph-Struktur des Individuums nicht sehr stark verändern kann. Der Mutationsoperator kann aber durch die Mutation der Branchingfunktion oder die Veränderung der Kanten zu einem ähnlichen Effekt kommen wie der graphbasierte Crossoveroperator. Dies gilt dadurch, dass es in die Individuen ganze Pfade gibt, die Introns sind und durch eine Mutation aktiviert werden können und somit wie eine graphbasierte Crossoveroperation wirken [4, 76]. Das gleiche Argument gilt auch für das linearbasierte Crossover, erstaunlich ist nur, dass während der Evolution eine Mutationsrate von 20% ausreichend ist, um den negativen Effekt eines falschen Parameters zu kompensieren. Um dies abschließend klären zu können, müssten jedoch genaue empirische Tests mit verschiedensten Parameterkombinationen und Problemen durchgeführt werden, in dieser Arbeit ist aber nicht der Raum um diese Fragestellung genauer zu untersuchen.

4.4.5 Fazit der verschiedenen Strukturen

Dieser Unterschied ist sehr wichtig, da das Ziel der Dissertation die Evolution von Schachprogrammen ist und die Evolution von Schachprogrammen durch die Fitnessbewertung sehr rechenzeitintensiv ist. Bei den Ergebnissen der Schachprogramme ist weiter das absolut beste Programm interessant und nicht die durchschnittlichen Ergebnisse. Somit sind möglichst geringe Variationen in den Ergebnissen verschiedener Läufe gewünscht.

Bei der empirischen Analyse hat sich gezeigt, dass die Baumstruktur für das Rastrigin Problem mit 100 Individuen Lösungen finden kann, die besser als die Ergebnisse der Linear-Graph Struktur sind, jedoch passierte dies während der Analyse selten. Durch sehr viele schlechte Ergebnisse werden diese Vorteile wieder aufgehoben, siehe Abbildung 4.30. Das Bild ändert sich aber sofort, wenn man die Anzahl der Individuen auf 10 reduziert, die Anzahl der schlechten Resultate steigt für die Baumstruktur stark an und sie kann auch nicht das Ergebnis erreichen, welches für diese Einstellung durch die Linear-Tree Struktur erreicht wird, siehe Abbildung 4.29. Ein ähnliches Argument gilt auch für die Linear-Tree Struktur, sie erzeugt für dieses Problem die besten Lösungen, aber auch mehr schlechtere als die Linear-Graph Struktur. Für fast alle anderen Probleme lieferten die neuen GP Strukturen immer bessere Ergebnisse als die lineare oder Baumstruktur. Die Ausnahme waren hier die Testläufe für das Spiralproblem mit 100 Individuen, in der sich die Baumstruktur zwar schlechter als die Linear-Graph Struktur schlug, aber besser als die Linear-Tree Struktur, wie man in der Abbildung 4.24 gut erkennen kann. Das Bild ändert sich aber sofort, wenn man sich die Testläufe mit 10 Individuen betrachtet. Dann können die Ergebnisse der Baumstruktur nicht mehr mit den neuen GP Strukturen konkurrieren, siehe Abbildung 4.23. Dies ist aber ein wichtiger Punkt für die Evolution von Schachprogrammen, denn durch den hohen Rechenaufwand muss man mit wenigen Individuen und wenigen Evaluationen auskommen. Die neuen Strukturen konnten schon nach weniger als 10.000 Evaluationen die gleiche Fitness erreichen, welche die Baumstruktur erst nach 200.000 Evaluationen erreichen konnte. Ein ähnliches Verhalten erkennt man für das Chain-Problem, hier ist es nicht nur bei den Tests mit 10 Individuen, sondern auch mit 100 Individuen wiederzufinden. Für einige andere Probleme sind die Ergebnisse der Linear-Tree und -Graph Strukturen so dramatisch besser, dass man diese Strukturen den anderen vorziehen muss.

Während der Analyse hat sich aber auch klar gezeigt, dass die Linear-Graph Struktur gegenüber der Linear-Tree Struktur einen wichtigen Vorteil hat, denn die Ergebnisse der Linear-Graph Struktur haben eine viel kleinere Varianz als die der anderen Strukturen. Dies ist schon an den Standardabweichungen in den Plots zu erkennen, aber besonders gut kann man es in den Abbildungen 4.27 bis 4.36 erkennen. Hier ist immer die Fitness der letzten Generation für alle durchgeführten Läufe abgebildet. Man erkennt gut, dass bis auf die Läufe für das Sinusproblem mit 100 Individuen und das Spiralproblem mit 10 Individuen, in beiden Fällen hatte die lineare Struktur die schlechtesten Ergebnisse erzeugt, immer die GP Läufe mit Baumstruktur die schlechtesten Ergebnisse lieferten. Diese Struktur hat zwar auch sehr gute Ergebnisse erzielt, erreichtet diese aber sehr selten. Für die Aufgabenstellung in dieser Arbeit, wird eine GP-Struktur benötigt, die mit kleinen Pools arbeiten kann, die gute Ergebnisse findet und die eine möglichst geringe Varianz in den Ergebnissen erzeugt. Der letzte Punkt ist eigentlich auch der wichtigste Punkt für diese Arbeit, da Schachprogramme evolviert werden sollen, ist das Ziel, *ein* sehr gutes Programm zu erhalten und da die Rechenzeiten sehr hoch sind, möchte man dieses Ziel mit so wenig Evolutionsläufen wie möglich erreichen. Daher fiel für das GeneticChess System die Wahl für die interne Struktur der Individuen auf die Linear-Graph Struktur.

Kapitel 5

Das GeneticChess-System

GeneticChess ist kein allgemeines GP System, sondern ein spezialisiertes System mit dem Ziel Schach spielende Individuen zu generieren. Genauer definiert besteht das Ziel darin, eine *gute* Suchheuristik für das Schachspiel zu finden. Denn die Suchheuristik ist das zentrale Modul eines Schachprogramms, kann aber auch für viele andere Probleme eingesetzt werden. Viele Problem können direkt durch Suchverfahren gelöst werden, andere können durch eine entsprechende Tranformation mittels Suchverfahren, auch *State-Space Verfahren* genannt, gelöst werden [88, 59]. Die Ergebnisse dieser Arbeit können durch entsprechende Anpassungen auf andere Suchprobleme angewendet werden, um so Problemstellungen zu lösen, die noch nicht durch GP-Programme gelöst werden. Beispiele wären hier Optimierprobleme wie das TSP oder das Vehicle Routing Problem und auch andere Probleme, die durch Suchverfahren gelöst werden können.

Wenn man sich nun das Schachproblem genauer anschaut, stellt man fest, dass man es in verschiedene Teilprobleme aufteilen kann, die sowohl durch den zeitlichen Ablauf als auch durch den Aufbau des Spiels selber gegeben sind. Die wichtigsten Teilprobleme des Schachspiels, die von Schachspielern und guten Schachprogrammen gelöst werden müssen, sind:

1. Die *Eröffnung*, sie umfasst den zeitlichen Bereich zu Beginn des Spiels, sie ist nicht genau definiert, normalerweise umfasst sie die ersten 10 bis 15 Züge einer Partie. Im allgemeinen werden eine große Datenbasis von verschiedenen Eröffnungen und auch verschiedene Regelmäßigkeiten in den Stellungen untersucht um gute Eröffnungen zu spielen.

2. Das *Mittelspiel* versucht ganze Stellungen, aber auch wichtige Konstellationen verschiedener Figuren zu erkennen, um so einen taktischen Vorteil in einer Situation erringen zu können. In dieser Phase des Spiels werden im Computerschach Suchalgorithmen wie die Alpha-Beta Suche verwendet.

3. Das *Endspiel* ist die Situation im Spiel, wenn nur noch wenige Figuren auf dem Brett stehen. Hierfür wurden verschiedene Regeln entwickelt, aber auch Datenbanken mit der vollständigen Lösung von verschiedenen Figurkombinationen aufgebaut.

4. Die *Materialbewertung* bewertet eine gegebene Schachstellung, dabei werden sowohl die Figuren als auch die Stellung der Figuren bewertet.

5. Die *Variantenberechnung* berechnet die möglichen Konsequenzen einer gegebenen Stellung und ermittelt daraus einen *guten Zug*.

6. Die *Planung* definiert für eine gegebene Stellung ein langfristiges Ziel. Die Aufgabe der langfristigen Planung wird von Computerschach Programmen selten durchgeführt. Hier wird anders als bei menschlichen Spielern das Gewicht stärker auf die Variantenberechung gelegt.

Alle folgenden GP Systeme haben nicht das Ziel die Aspekte der *Eröffnungstheorie*, der *Theorie des Mittelspiels* oder der *Endspieltheorie* zu lösen. Es werden keine Module für diese Aufgaben evolviert, da diese Aspekte immer auf einer Wissensbasis und die Lösung auf Data Mining Verfahren beruhen. Die Praxis zeigt, dass alle diese Verfahren die Spielstärke jedes Programms um mehrere hundert Elopunkte steigern können, aber eine gute Suchheuristik ist der Kernpunkt jedes Programms. In der Literatur und in der Praxis findet man sehr viele verschiedene Varianten der Suchheuristiken und immer noch werden Verbesserungen an bekannten Verfahren vorgeschlagen. Daher konzentriert sich die Arbeit auf die Suchheuristiken und nicht auf alle möglichen Elemente des Schachspiels.

Um dieses Ziel zu erreichen, wird den GP Systemen ein allgemeines Modul einer Suchheuristik und deren Programmfluss vorgegeben. Das Programm und seine Module sollen sich dann automatisch durch den evolutionären Prozess bilden. Dieses Vorgehen führte letztlich dazu, dass das *State-Space Individuum* entwickelt wurde, welches einen festen Rahmen für ein Suchprogramm vorgibt (siehe Kapitel 4.3). Diese Erweiterung der Genetischen Programmierung ist jedoch nicht ausreichend, um ein Individuum zu generieren, das Schach spielt. Hierzu sind spezielle Elemente notwendig, die auf das Schachproblem ausgerichtet sind. Darunter fallen insbesondere die Fitnessfunktion, Funktionen und Terminale sowie zusätzliche Datenstrukturen, welche die Problemlösung unterstützen.

5.1 Vorarbeiten

Das GeneticChess System, das in den folgenden Kapiteln vorgestellt wird, hat sich aus verschiedenen Vorarbeiten der letzten Jahre entwickelt. Die Ergebnisse, aber auch Fehlschläge dieser Systeme, sind in das Design der verschiedenen Komponenten des Systems eingeflossen. Sie motivierten aber auch die Suche nach flexibleren und angepassteren Individuenstrukturen um ein gegebenes Problem zu lösen. Die Reihenfolge der Kapitel geben die zeitliche Entwicklung der einzelnen Projekte wieder.

Die Grundidee aller dieser Systeme war es, ein Schachprogramm zu entwickeln, welches evolutionäre Verfahren verwendet. Daher stellten sich zwei zentrale Fragen: Erstens, wie evolviert man ein GP Programm, das Schach spielt und zweitens, wie kann man dies mit der aktuellen Rechenkraft in einer angemessenen Zeit tun. Diese Fragen werden in den folgenden Kapiteln bearbeitet und mit verschiedenen Methoden gelöst.

5.1.1 ChessGP

Im *ChessGP* System wird das Problem zur Evolution von Schachprogrammen dadurch gelöst, dass man es in verschiedene Teilprobleme zerlegt. Diese Teilprobleme werden dann in einer *hierarchischen Evolution* entwickelt, die letztlich in einem evolvierten Schachprogramm endet. Die Teilprobleme werden durch eigenständige Teilindividuen gelöst und anschließend in einer *hierarchischen Struktur* zusammengefasst. Die Teilindividuen werden jeweils einzeln evolviert, dabei wird jedoch auf den hierarchischen Aufbau des Gesamtindividuums geachtet. Dies bedeutet, dass Teilindividuen, die einer höheren Hierarchieebene angehören, erst evolviert werden können, wenn die Teilindividuen der tieferen Hierarchieebenen fertig evolviert wurden. Diese werden dann den Teilindividuen der höheren Hierarchieebene als zusätzliche Terminale zur Verfügung gestellt (siehe Abschnitt 5.1.1.2).

Eine wichtige Erweiterung des Systems, gegenüber bekannten Systemen der Genetischen Programmierung, ist die Einführung von Datenstrukturen für die Individuen. Die Individuen in GP Systemen arbeiten normalerweise mit einem einfachen Datentyp, entweder Double oder Integer, jedoch nie mit einer komplexen Datenstruktur. Es hat sich gezeigt, dass viele Funktionen und Terminale für das Schachproblem komplexe Datenstrukturen benötigen. Somit ist es sinnvoll den GP Individuen die Möglichkeit zu geben, direkt auf diesen Datenstrukturen zu arbeiten und nicht diese Strukturen auf Double- oder Integerwerte herunterbrechen zu müssen (mehr zu diesem Thema in Kapitel 5.1.1.6).

Während der Implementierung stellte sich schnell die Frage, welche GP Strukturen für die einzelnen Teilindividuen verwendet werden sollten. Zu Beginn des Projektes gab es die neuen Individuentypen noch nicht, so fiel die Wahl auf lineare GP Strukturen. Weiteres zu diesem Thema ist in Kapitel 6.1 zu finden.

5.1.1.1 Die Fitnesscases

Die Ergebnisse maschineller Lernverfahren hängen sehr stark von der Beschaffenheit der Trainingsdaten ab. Für die gestellte Aufgabe sind es Schachstellungen. Nun kann man nicht wie für andere Probleme alle möglichen Stellungen betrachten, sondern muss eine Auswahl treffen. Daher haben wir folgende Anforderungen an die Trainingsdaten gestellt:

- Die Anzahl der Beispiele sollte nicht zu klein sein.
- Die Stellungen sollten realistische Stellungen sein.
- Die Diversität der Trainingsdaten sollte möglichst hoch sein.

Die verwendete Trainingsmenge beruht auf 640 real gespielten Partien, ihre Grundlage ist die Sonderausgabe [80] einer Schachzeitschrift, die in Deutschland als der Schachinformator bekannt ist. Diese Partiensammlung wurde in eine Menge von etwa 40.000 Schachstellungen transformiert. Es wurde dieser Weg beschritten, da versucht werden sollte hochwertige Partien als Trainingsmenge zu nutzen. Viele Partien in kommerziellen Datenbanken haben aber Fehler, enthalten Partien doppelt oder sind wenig repräsentativ. Daher haben wir uns entschieden einen eigenen Weg zu gehen und nicht auf kommerzielle Datenbanken zurückzugreifen. Durch das hohe Spielniveau der Partien ergab sich leider das Problem, dass die Partien frühzeitig durch die Spieler abgebrochen wurden, dieses wurde schon oft nach einem Materialverlust von nur einem Bauern getan. Daher wurden diese Partien von dem Schachprogramm *Crafty* zu Ende gespielt, um so auch Stellungen vom Ende der Partien als Fitnesscases zu erhalten [55].

5.1.1.2 Struktur der Schachindividuen

Die Struktur der Schachindividuen ist einer der zentralen Punkte, der die Möglichkeiten der GP Programme und somit auch ihre Güte begrenzt. Das Hauptziel der Evolution ist es, eine *gute* Suchheuristik für das Schachspiel zu finden. Daher analysierten wir verschiedene Heuristiken. Es zeigte sich, dass die wichtigsten Elemente in den Heuristiken die *Materialbewertung*, die *Variantenberechnung* und die *Planung* sind. Auf diesen drei Elementen beruht der Aufbau des Individuums, die Grafik 5.1 verdeutlicht die interne Struktur der Individuen. Es wurde eine hierarchische Struktur gewählt, da auch die drei Heuristiken einer Hierarchie unterliegen. Auf der untersten Ebene der Hierarchie liegt die Materialbewertung, auf der nächsten Ebene liegt das Modul, das kurzfristige Pläne vorschlägt und auf der letzten Ebene das Modul zur Variantenberechnung.

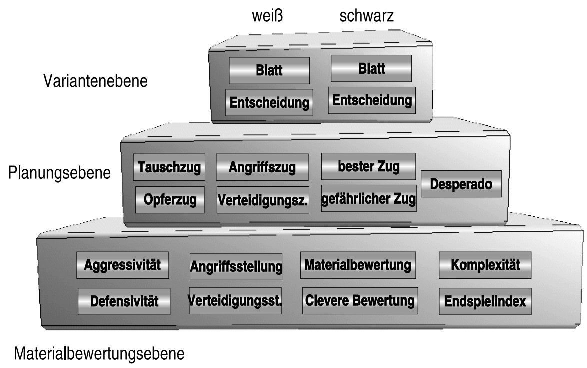

Abbildung 5.1: Die Struktur der Schachindividuen. Die Module der höheren Ebenen können die Module der tieferen Ebenen aufrufen.

Durch diese Struktur können die Module einer höheren Hierarchieebene jedes Modul der tieferen Hierarchieebenen aufrufen. Die oberste Hierarchieebene entscheidet, welcher Zug für eine gegebene Stellung gemacht wird, dazu kann und muss dieses Modul die Ergebnisse der unteren Hierarchieebenen nutzen, indem es diese aufruft. Dabei ist es wichtig zu erwähnen, dass jede Ebene wiederum aus mehreren GP Programmen besteht, die wie bei der Materialbewertung und der Planung gleichberechtigt nebeneinander stehen. Anders im Varianten Modul, hier stehen die Module durch einen vorgegebenen Programmfluss miteinander in Verbindung.

Die Grafik 5.1 zeigt den hierarchischen Aufbau des Individuums und der einzelnen Ebenen, auf die Struktur der einzelnen Ebenen wird erst später eingegangen. Jedoch kann man schon erkennen, dass die Materialbewertungsebene aus acht Modulen besteht, die Planungsebene aus sieben und die Variantenebene aus vier Modulen. Wichtig ist noch zu erwähnen, dass Module der Material- und Planungsebene sich nicht gegenseitig aufrufen können, nur in der Variantenebene rufen sich die Module gegenseitig auf.

5.1.1.3 Materialbewertungsebene

Die *Materialbewertungsebene* hat die Aufgabe für eine gegebene Stellung eine Bewertung zu finden. Bei der Bewertung handelt es sich um eine statische Bewertung, da nur von der gegebenen Stellung ausgegangen wird und nicht von möglichen Zügen oder Entwicklungen der Stellung. Die Bewertung ist somit eine Funktion, die eine Stellung auf eine reelle Zahl abbildet. Diese wird dann als ein Gütemaß für die untersuchte Stellung interpretiert. Die Ebene in der hierarchischen Struktur selbst besteht aus mehreren Modulen, die die Aufgabe aus verschiedenen Blickwinkeln betrachten sollen.

Die einfachste Funktion um eine Materialbewertung beim Schachspiel durchzuführen, wurde von Shannon um 1950 vorgestellt. Hierzu gibt er jedem Figurtyp einen ganzzahligen Materialwert zwischen 100 für einen Bauern und 900 für die Königin. Die Funktion summiert dann die Materialwerte der Figuren jeder Seite auf und bildet anschließend die Differenz der beiden Summen. So kann man erkennen, welche Seite ein Materialübergewicht hat und dies entsprechend positiv oder negativ bewerten. Innerhalb der Computerschachwelt haben sich verschiedenste Materialbewertungsfunktionen entwickelt, die meisten Funktionen wurden für genau ein Schachprogramm entwickelt und auch nur in ihm eingesetzt.

An dieser Stelle wollten wir einen anderen Weg gehen, indem das Modul nicht nur aus einem GP Programm für eine Materialbewertung besteht, sondern aus einer Menge von verschiedenen Programmen. Jedes dieser Programme hat einen etwas anderen Blickwinkel auf die Aufgabe, so dass dem Gesamtindividuum die Möglichkeit gegeben werden soll die passende Materialbewertung für die verschiedenen Spielsituationen zu nutzen oder diese zu kombinieren.

Wir haben dazu acht verschiedene Fitnessfunktionen für Materialbewertungsfunktionen entwickelt, wobei die ersten vier Materialbewertungen neutral sind, das heißt, sie bewerten die Stellung unabhängig vom Spieler, der am Zug ist. Die letzten vier Bewertungen bewerten die Stellung von der Seite der GP Individuen aus; da die GP Individuen im ChessGP immer weiß spielen, sind die Bewertungen nur für die weiße Seite sinnvoll. Wir wollen acht verschiedene Materialbewertungen vom ChessGP-System entwickeln lassen. Dazu haben wir die folgenden Bewertungen entwickelt, die gleichzeitig die Zielfunktion der Module darstellen.

1. Die *Komplexitätsbewertung* bestimmt für jede Figur die Anzahl der möglichen Züge. Züge, die zum Schlagen einer anderen Figur führen, werden doppelt gewertet.

$$\begin{aligned} komp = \sum_{f \in \text{Figuren}} &(\text{Anzahl der von f erreichbaren Felder} \\ &+ \text{Anzahl der von f möglichen Schlagzüge}) \end{aligned} \tag{5.1}$$

2. Die *Endspielbewertung* ist ein Maß für den Endspielcharakter der Stellung. Dabei wird eine Stellung mit 13 Figuren als Endspiel betrachtet und erhält einen Wert von 100. Hat eine Stellung weniger Figuren, soll der Wert linear weiter ansteigen, bei weniger Figuren linear fallen, bis er für 32 Figuren den Wert null erreicht. Die folgende Funktion berechnet dieses Maß:

$$endIndex(\text{Anzahl der Figuren}) = \frac{32 - \text{Anzahl der Figuren}}{32 - 13} * 100 \tag{5.2}$$

3. Die *Aggressivitätsbewertung* ermittelt über die Anzahl der Gewinnschlagzüge[1] die Aggressivität der Stellung. Die Gewinnschlagzüge werden sowohl für die weißen als auch für die schwarzen Figuren ermittelt, der Wert für die Aggressivität ist dann:

$$agg = \sum_{f \in \text{Figuren}} \text{Anzahl der von f ausführbaren Gewinnschlagzüge} \tag{5.3}$$

4. Die *Defensivitätsbewertung* ermittelt über die Anzahl der gedeckten Figuren einen Wert für die Defensivität der Stellung. Dazu wird die Anzahl der Figuren gezählt, die eine Figur decken, und dies dann über alle Figuren summiert. Der Wert für die Defensivität ist dann:

$$def = \sum_{f \in \text{Figuren}} \text{Anzahl der f deckenden Figuren} \tag{5.4}$$

5. Die *einfache Materialbewertung* bildet die Summe der Figurenwerte jeder Seite und anschließend die Differenz. Die Figuren werden für die Fitnessfunktion wie folgt bewertet:

    - Bauer: 100
    - Springer: 300
    - Läufer: 300
    - Turm: 500
    - Dame: 900

[1] Dies sind Züge, bei denen die zu schlagende Figur höher- oder gleichwertig ist oder die zu schlagende Figur ungedeckt ist.

Der Materialwert berechnet sich dann wie folgt:

$$\begin{aligned} simple = & \sum_{w \in \text{weiße Figuren}} \text{Materialwert von } w \\ & - \sum_{s \in \text{schwarze Figuren}} \text{Materialwert von } s \end{aligned} \tag{5.5}$$

6. Die *clevere Materialbewertung* setzt sich aus dem Wert der einfachen Materialbewertung, zusätzlichen positionellen Kriterien und einer variablen Wertung der Bauern während des Spielverlaufes zusammen. Die Berücksichtigung der Brettposition erfolgt mittels einer Positionsmatrix, die es für jeden Figurtyp gibt. Die Positionsmatrizen wurden in Anlehnung an die Matrizen von GnuChess [23] erstellt. Die höhere Gewichtung der Bauern wird mit Hilfe des Endspielindexes realisiert. Für die clevere Materialbewertung ergibt sich dann folgende Formel:

$$\begin{aligned} clever = \text{simple } + & \sum_{w \in \text{weiße Figuren}} \text{Positionswert}(w) \\ & - \sum_{s \in \text{schwarze Figuren}} \text{Positionswert}(s) \\ & + 0,5 * \text{Endspielindex} \\ & * \text{Anzahl weiße Bauern} \\ & - 0,5 * \text{Endspielindex} \\ & * \text{Anzahl schwarze Bauern} \end{aligned} \tag{5.6}$$

7. Die *Angriffsbewertung* bewertet inwieweit die weißen Figuren eine angreifende Position einnehmen. Dabei wird jede angegriffene schwarze Figur in der Höhe ihres Materialwertes in die Bewertung einbezogen. Gefesselte Figuren werden mit einem Bonus von 300 Punkten bewertet. Der Angriff auf den König wird bei dieser Funktion stark gewichtet, so wird der Materialwert des Königs mit 3.000 und die benachbarten Felder des Königs mit 75 bewertet. Die Summe wird durch den Materialwert aller schwarzer Figuren ohne König geteilt, um so eine Relation der materiellen Stärke von Schwarz zu erhalten. Zur Normalisierung wird der Wert noch mit $\frac{10.000}{\sum_{s \in \text{schwarze Figuren}}}$ multipliziert und gerundet.

$$\begin{aligned} angriff = (& \sum_{f \in \text{angegriffene Figuren}} \text{Materialwert von} f \\ & + 300 * \text{Anzahl gefesselter Figuren} \\ & + 75 * \text{Anzahl bedrohter Königsfelder}) \\ & * \frac{10.000}{\sum_{s \in \text{schwarze Figuren}} \text{Materialwert von } s} \end{aligned} \tag{5.7}$$

8. Die *Verteidigungsbewertung* bewertet die Stellung unter dem Aspekt, inwieweit die weißen Figuren eine verteidigende Stellung einnehmen. Die Bewertung wird analog zu der Angriffsbewertung durchgeführt. Hierzu wird aber von der Materialdifferenz der gedeckten und ungedeckten weißen Figuren ausgegangen und zusätzlich noch die Verteidigung des Königs in die Bewertung mit einbezogen. So werden die Felder, die an den König grenzen und von weißen Figuren besetzt sind, mit plus 100 gewertet, unbesetzte Felder mit minus 100 und Felder, die von schwarzen Figuren besetzt sind, mit minus 200.

$$
\begin{aligned}
verteidigung = (&\sum_{f \in \text{gedeckte wieße Figur}} \text{Materialwert von } f \\
&- \sum_{f \in \text{ungedeckte wieße Figur}} \text{Materialwert von } f \\
&+ 100 * \text{Anzahl weiß besetzter Königsfelder} \\
&- 100 * \text{Anzahl unbesetzter Königsfelder} \\
&- 200 * \text{Anzahl schwarz besetzter Königsfelder}) \\
&* \frac{10.000}{\sum_{s \in \text{schwarze Figuren}} \text{Materialwert von } s}
\end{aligned}
\tag{5.8}
$$

Jede dieser Bewertungen ist zu unterschiedlichen Zeiten im Schachspiel sinnvoll und kann somit zu einer Verbesserung der Bewertung gegenüber einer Standardbewertung führen. Durch den Aufbau des Individuums sollte die Nutzung der verschiedenen Module verstärkt werden, denn sowohl die Planungsebene als auch die Variantenebene konnten auf die einzelnen Module zugreifen. Inwieweit unsere Erwartungen erfüllt wurden, ist im Abschnitt 6.1.4 beschrieben. Die Ergebnisse der verschiedenen Materialbewertungsmodule sind in Kapitel 6.1.1 zusammengefasst.

5.1.1.4 Planungsebene

Die *Planungsebene* des Individuums soll für eine gegebene Stellung eine Zugliste generieren. Die einfachste aller Zuglisten ist die Liste aller möglichen Züge. Werden aber alle Züge in jeder Stellung expandiert, hat man ein exponentielles Wachstum des Suchraums, um dies zu verhindern, werden in den klassischen Verfahren die Suchräume durch verschiedene Verfahren beschnitten, siehe hierzu die Kapitel 2.2.4, 2.2.5 und 2.2.6. In unserem Modell sollte es nun aber nicht zu einer Bewertung aller Züge kommen, sondern es sollte ein kurzfristiger Plan für die aktuelle Stellung entwickelt werden und entsprechend des Plans sinnvolle Züge zur Verfügung gestellt werden. Dieses Modul enthält dafür sieben verschiedene Individuen, die Zuglisten mit verschiedenen Ausrichtungen generieren sollen.

- *Beste Züge* versucht eine qualitative Einschätzung der Situation zu treffen und ermittelt die Züge, die zu einer Verbesserung der Stellung führen.
- *Gefährliche Züge*, sind Züge, die zu einem Materialgewinn führen können, jedoch auch die Gefahr von Verlusten in sich tragen. So wurden die gefährlichen Züge für die Fitnesscases dadurch ermittelt, dass Weiß zwei Züge machen kann und Schwarz keinen. Wenn Weiß durch diesen Doppelzug eine signifikant bessere Stellung erreichen kann, so wird dieser Zug als *gefährlich* gewertet. Eine Stellung wurde als signifikant besser gewertet, wenn sie 300 Punkte mehr hat als die Ausgangsstellung, 300 Punkte entspricht ungefähr einem Läufer oder Springer.
- *Desperado Züge* sind Züge, die eine für Weiß verlorene Stellung wieder öffnen können. Dabei wird versucht die geplante Taktik des Gegners zu durchbrechen, da er mit einem solchen Zug nicht gerechnet hat. Um diese Züge zu finden, wird ein Spielbaum der Tiefe eins aufgespannt, mit einer veränderten Materialbewertung bewertet und durch zusätzliche Merkmale erweitert. Der Materialwert für die weißen Figuren bei Desperado Zügen ist 75 für die Bauern, 320 für den Springer, 280 für den Läufer, 450 für den Turm und 950 für die Dame. Die Materialwerte für

die schwarzen Figuren bleiben unverändert. Ein weiteres Kriterium ist die Stellung des schwarzen Königs, für jedes den König umgebenden Feld sowie das Königsfeld, wird der Materialwert um 30 verringert, wenn das Feld von einer weißen Figur angegriffen wird. Ist ein solches Feld von einem schwarzen Bauern besetzt, wird der Materialwert um 80 erhöht. Der Materialwert wird erhöht, wenn Weiß noch die Dame oder einen Springer besitzt, der Wert ist abhängig von der Entfernung (*Nähe*) zwischen den Figuren und dem schwarzen König. Die *Nähe* zweier Figuren wird auf einer Reihe, Spalte oder Diagonale bewertet. Benachbarte Felder haben einen Wert von 90 und mit jedem Feld zwischen den Figuren sinkt dieser Wert um 10. Als letzter Faktor geht die Anzahl der möglichen Spielzüge mit in die Bewertung ein. Jeder Zug von Weiß erhöht den Wert um 12 und jeder von Schwarz senkt ihn um 10.

Zusammenfassend ergibt sich folgende Formel für die Bewertung:

$$\begin{aligned} desp = {} & \text{Material Weiß} - \text{Material Schwarz} \\ & + \sum_{f \in \text{weiße Dame/Springer}} \text{Nähe}(f, \text{schwarzer König} \\ & + 30 * \text{Anzahl angegriffener Königsfelder} \\ & - 80 * \text{Anzahl mit schwarzen Bauern besetzte Königsfelder}) \\ & + 12 * \text{Anzahl weißer Spielzüge} \\ & - 10 * \text{Anzahl schwarzer Spielzüge}. \end{aligned} \tag{5.9}$$

- *Angriffszüge* sind eine Fortführung der Angriffsbewertung aus Kapitel 5.1.1.3. Hierbei wird ein Spielbaum mit der Angriffsbewertung aufgespannt. Die Bewertung eines Zuges ist dann die Differenz der Ausgangsstellung und des kleinsten Materialwertes eines Blattes im Spielbaum.

- *Verteidigungszüge* werden analog zu Angriffszüge bewertet. Hierzu wird nur die Verteidigungsbewertung benutzt.

- *Tauschzüge* sind im Schach Zugfolgen, bei der beide Seiten Figuren schlagen, das Materialverhältnis jedoch gewahrt wird. Hier werden aber alle Schlagzüge als Tauschzüge gewertet, bei der es zu keiner Verschlechterung des Materialverhältnisses kommt.

- *Opferzüge* werden hier als Züge definiert, die dem Gegner das Schlagen einer Figur ermöglichen. Der Zug muss aber dann in maximal vier Halbzügen zu dem Gewinn der Partie führen, einen Materialgewinn von mindestens 400 Punkten nach der cleveren Materialbewertung oder einen taktischen Vorteil ermöglichen. Als Vorteil wird dabei das Ansagen von Schach, das Schlagen eines Bauern zum Schutz des Königs und das Schlagen eines Königsbauern gewertet.

Die Aufgabe der Individuen ist es, eine Liste von Zügen zu erzeugen, neben dieser Anforderung soll die Liste der Aufgabenstellung entsprechen. Die Zugliste eines Individuums für Angriffszüge soll hauptsächlich Angriffszüge enthalten und diese Liste soll auch nicht zu lang sein, da die Länge der Liste dem Outgrad eines Knotes im Suchbaum entspricht und somit direkt die Laufzeit der evolvierten Individuen beeinflusst. Ein weiteres Merkmal der Liste sollte eine Sortierung der Liste sein, die besten Züge für die gestellte Aufgabe sollten am Anfang der Liste stehen. Da man nicht für jede Stellung der Datenbank alle Zuglisten generieren konnte, enthält die Datenbank nur die Bewertung der einzelnen Züge, die in einer Stellung möglich sind. Die Fitness einer Zugliste wird mit

Hilfe der Daten aus der Datenbank über die einzelnen Züge ermittelt. Die Funktion ist entsprechend der Kriterien *Qualität, Sortierung* und *Länge* unterteilt.

Die *Qualität der Zugliste* ist die Summe der Bewertung der einzelnen Züge (im folgenden *Zugwerte* genannt). Die Zugwerte wurden in der Datenbank auf dem Intervall von $[0, 1000]$ normiert. Der beste Zug hat dann den Wert 1.000 und der schlechteste den Wert 0. Für einige Bewertungskriterien wie die *Gefährlichen Züge, Opferzüge* oder *Tauschzüge* werden nur Züge aufgenommen, die dem entsprechenden Kriterium genügen. Daher sind einige legale Züge nicht in der Datenbank enthalten, diese Züge enthalten den Zugwert -250, wenn sie vom Individuum ausgewählt werden.

Die *Sortierung der Liste* ist ein Strafterm in der Fitness, der sich aus der Summe der Differenzen der Zugwerte zwischen der Zugliste des Individuum und der Zugliste der Datenbank ergibt:

$$\text{Sortierung} = \sum_{i \leq m} abs(\text{Zugwert}_i - \text{Zugwert}_{Pos_i}) \text{ für alle } i < Pos_i \tag{5.10}$$

Wobei m die Länge der Zugliste, i die Position des Zuges in der Zugliste und Pos_i die Position des gleichen Zuges in der Datenbank sind. Hat ein Zug also in der Zugliste die Position 1 (mit dem Zugwert 1.000) inne und in der Datenbank die Position 2 (Zugwert 800), so ergibt sich für diesen Zug ein Strafwert von 200.

Die *Länge der Liste* ist ein weiterer Strafterm in der Fitness, dieser ist wichtig, um die Listenlänge der Individuen sowohl nach oben als auch nach unten zu beschränken. Als optimale Listenlänge wird die Länge der Liste in der Datenbank für die betrachtete Brettstellung genommen. Wird die optimale Listenlänge (l_{opt}) unter- oder überschritten, so wird die Bewertung der Zugliste wie folgt bestraft:

$$\text{Länge} = abs(l_{opt} - m) * \text{Zugwert}_{l_{opt}} \tag{5.11}$$

Zusammenfassend ergibt sich daraus folgende Fitness für die Bewertung der Zugliste:

$$fit = \text{Qualität} - \text{Sortierung} - \text{Länge} \tag{5.12}$$

Abschießend wird die Bewertung der Zuglisten noch auf das Intervall $[0, 1000]$ normiert. Die beste Zugliste hat dann den Wert 1.000 und die schlechteste den Wert 0.

5.1.1.5 Variantenebene

Die evolvierten Programme der Bewertungsebene sollen statische Bewertungen einer Brettstellung liefern. Die Planungsebene soll bezüglich einer Stellung eine Zugliste berechnen, was eine semi-dynamische Aufgabe ist. Diese beiden Ebenen sind aber nur Bausteine für die *Variantenebene*, sie stellt mit den Bausteinen das Gesamtindividuum dar, welches Schach spielen soll. Der hierarchische Aufbau des Individuums mit den verschiedenen Ebenen und den einzelnen Bausteinen ist in der Abbildung 5.1 dargestellt. Die Aufgabe des Gesamtindividuums ist es nun zu einer Stellung einen *Zug* zu berechnen. Dazu werden in einer Baumsuche verschiedene Varianten für die Stellung durchgerechnet und anschließend eine davon ausgewählt.

Um den GP-Programmen eine Baumsuche zu ermöglichen, wird das Programm rekursiv ausgeführt. Die Eingabe für das Individuum ist eine Stellung, dann wird das *Entscheidungs-Modul* für die weiße oder schwarze Seite aufgerufen. Es entscheidet darüber, ob die Stellung weiter expandiert wird oder nicht und welches Programm für die

Berechnung der Zugliste zuständig ist. Dann wird das entsprechende Programm aufgerufen und eine Zugliste generiert. Für jede Stellung in der Zugliste wird nun wieder das Individuum aufgerufen. Das Entscheidungs-Modul der entsprechenden Seite wird aufgerufen, bis entweder das Entscheidungs-Modul die Stellung nicht weiter expandiert oder die Zugliste, die von dem Individuum der *Planungsebene* zurückgegeben wird, leer ist. Wenn dies der Fall ist, wird das *Blatt-Modul* der entsprechenden Seite aufgerufen, es entscheidet darüber, welches Programm der Bewertungebene aufgerufen wird. Die Bewertungen der Blätter werden dann zurückgereicht. Dies wird für alle Züge einer Zugliste durchgeführt und so entsprechend einer Baumsuche die Minima oder Maxima der Blattbewertungen berechnet und an die nächste Ebene weitergereicht.

Den GP-Programmen stehen verschiedenste Funktionen zur Verfügung um adequate Programme für diese Ebene zu entwickeln, sie sind alle im Abschnitt 8.1.3 zusammengefasst. Die zwei wichtigsten Funktionen sind allerdings die *exit*- und *cut*-Funktion für das Entscheidungs-Modul. Wenn die *exit*-Funktion ausgeführt wird, wird die Rekursion abgebrochen und die Materialbewertung aufgerufen, weiterhin erfolgt ein Abschneiden aller noch nicht bewerteten Züge in der Zugliste. Die *cut*-Funktion hat ein ähnliches Verhalten, jedoch ist die Auswirkung auf die Stellung beschränkt, alle weiteren Züge aus der Liste werden weiter bearbeitet.

Auch wenn zur Rückbewertung das Minimaxverfahren eingesetzt wird, so unterscheidet sich die Suche von diesem Verfahren jedoch in den folgenden Punkten:

- Es werden nicht alle legalen Züge betrachtet.
- Das Entscheidungs-Modul bestimmt für jede Stellung einen eigenen Zuglistengenerator. Somit ändert sich das Verhalten der Suche in jeder Stellung.
- Es findet keine konventionelle Stellungsbewertung statt, die für jedes Blatt dieselbe ist, sondern das Blatt-Modul entscheidet für jede Stellung, welche Bewertung sinnvoller ist.
- Eine maximal erlaubte Suchtiefe wird nicht vorgegeben, es ist ein Attribut des Individuums, welches durch Mutation geändert werden kann.

Die Fitness der Individuen wurde mit Hilfe des Elo-Verfahrens bestimmt (siehe Kapitel 2.3). Im Elo-System können Partien nur mit den Werten $0, 0,5$ und 1 bewertet werden, also *Verlust, Remis* und *Gewinn.* Im Gegensatz zur realen Schachpraxis, können im ChessGP-System aber alle Werte im Intervall von $[0, 1]$ angenommen werden. Dadurch wird die Bewertung der Individuen beschleunigt, denn nun muss nicht bis zum Gewinn einer Seite gespielt werden, sondern die Partie kann nach einer maximalen Anzahl von Zügen beendet werden. Der Maximalwert für die Partielänge wurde auf 70 Halbzüge festgelegt; falls zu diesem Zeitpunkt noch kein Gewinner feststeht, wird der Materialwert als Bewertung genutzt. Eine Partie wird aber auch dann abgebrochen, wenn eine Seite über mindestens vier Halbzüge eine *große* Materialüberlegenheit hat. Eine *große* Materialüberlegenheit von Individuum A zu Indivdium B ist dann erreicht, wenn der Materialwert von Individuum A größer als zweimal dem Materialwert von Individuum B ist und der Materialwert von A mindestens neun Bauerneinheiten entspricht, oder wenn der Materialvorteil von A größer als neun Bauerneinheiten ist. Dieses Abbruchkriterium ist ausreichend, da Schachgroßmeister teilweise bei einem Materialnachteil von einem Bauern kapitulieren.

5.1.1.6 Datenstruktur

Die Entwicklung einer *Datenstruktur* für die GP Individuen war eines der wichtigsten Elemente zur Lösung des Schachproblems. Viele der Funktionen, die für die Schachindividuen entwickelt wurden, haben verschiedene Eingabe- und Ausgabetypen, so brauchen einige Funktionen eine Position des Bretts und andere Funktionen einer Figur oder eine Zugliste. Alle Funktionen mit einer genaueren Beschreibung für die ChessGP Anwendung finden sich im Kapitel 8.1.

Wenn keine Datenstrukturen eingeführt worden wären, hätte man alle Funktionen so implementieren müssen, dass die entsprechenden Eingabe- und Ausgabewerte der Funktionen den gleichen Typ haben, da kein grammatikalisches GP verwendet wurde [94, 95]. Jedoch würde dies auch nicht alle Probleme lösen, da viele Werte zwar den gleichen Typ, jedoch verschiedene Bedeutungen besitzen. Der Semantik der Ein- und Ausgabewerte der Funktionen sollte durch die Verwendung einer Datenstruktur eine größere Bedeutung beigemessen werden. Dadurch sollte für das System die Komplexität zur Erzeugung sinnvoller Programme reduziert werden, da nun die Wahrscheinlichkeit, dass semantisch falsche Werte von den Funktionen genutzt werden, stark herabgesetzt wird.

Ein anderer Aspekt der Datenstrukturen ist, dass in einem linearen Genom nun parallele Programmflüsse auftreten können, da der Datenfluss parallel über die verschiedenen Elemente der Datenstruktur läuft. Verschiedene Funktionen lassen die parallelen Programmflüsse zusammenlaufen und andere trennen sie auf, so dass es zu einem komplexen Programmablauf kommen kann.

Die Abbildung 5.2 zeigt die verschiedenen Datenstrukturen für die einzelnen Ebenen der GP Individuen. Deutlich ist hierbei die Komplexität der Datenstruktur für die Planungsebene zu erkennen, die innerhalb der Datenstruktur ganze Zuglisten verwalten kann. Dies war notwendig, da das Ergebnis dieser Ebene eine Zugliste war. Die einfachste Datenstruktur hat die Variantenebene, sie hat einen Wert und zwei Indizes für die Programme der Material- und Planungsebene.

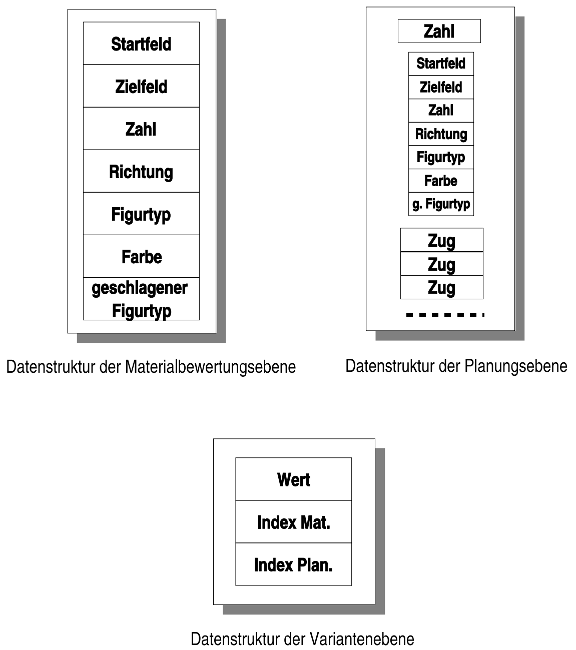

Abbildung 5.2: Die Datenstrukturen der einzelnen Ebenen des GP Systems.

Die Datenstruktur der Materialebene ist ein Dreier-Tupel und besteht aus einer Feld-, Figur- und Zahlenkomponente. Wichtig an dieser Datenstruktur ist eine Besonderheit der Feldkomponente, sie speichert zwar ein Feld des Brettes als Zahl zwischen 0 und 63, jedoch können einige Funktionen der Individuen diese auch nur bezüglich des Zeilen- oder Reihenwertes verändern.

Die Datenstruktur der Planungsebene besteht aus drei Hauptkomponenten, von denen zwei wieder aus Datenstrukturen bestehen. Die Zahlkomponente ist einfach eine reelle Zahl, die für Berechnungen verwendet wird. Die Suchmaskenkomponente ist wiederum ein Siebener-Tupel, das einen Zug beschreibt. Es besteht aus einem Startfeld, einen Zielfeld, einer Zahl, einer Richtung, dem Figurtyp, der Figurfarbe und dem Typ der geschlagenen Figur. Die letzte Komponente der Datenstruktur ist eine Zugliste, in der jedes Element der Liste dem Siebener-Tupel der Suchmaskenkomponente entspricht.

Die Datenstruktur der Variantenebene ist auch ein Dreier-Tupel und besteht aus einer Zahlenkomponente und jeweils einem Index für die Programme der Material- und Planungsebene.

5.1.1.7 Evolution

Die Evolution des Chess-GP Systems teilt sich in drei Evolutionen auf. Jede hierarchische Ebene des Individuums durchlief eine eigene Evolution mit unterschiedlichen Fitnesscases und unterschiedlichen Fitnessfunktionen. Durch den hierarchischen Aufbau der Individuen konnten die Evolutionen der einzelnen Ebenen aber nicht parallel durchgeführt werden, da ja die hierarchisch höhere Ebene immer die evolvierten Individuen der tieferen Ebene benötigt. Die Materialbewertungs- und Planungsebene bestehen aus mehreren Individuen, die verschiedene Zielsetzungen verfolgen. Diese Individuen sind nicht voneinander abhängig und können somit parallel evolviert werden. Die Variantenebene besteht zwar auch aus vier Individuen, diese sind jedoch eine Einheit und müssen innerhalb einer Evolution evolviert werden. Da das Gesamtindividuum hierarchisch aufgebaut ist, ergibt sich eine natürliche Reihenfolge für die Evolutionen der Ebenen.

Die Evolution eines schachspielenden Individuums in ChessGP beginnt mit der Evolution der Materialbewertungsebenen. Hierzu wurde ein Lineares GP-System verwendet, wobei eine Tunierselektion verwendet wurde. Die Individuen wurden mit Beispielen verschiedener Stellungen trainiert. Da die einzelnen Individuen für die verschiedenen Aspekte der Materialbewertungsebenen unabhängig voneinander sind, kann die Evolution der acht verschiedenen Materialbewertungsindividuen parallel ablaufen. Nachdem die Evolution dieser Individuen abgeschlossen ist, kann die Evolution der Planungsebene beginnen.

Vor der Evolution der sieben verschiedenen Individuen der Planungsebene, werden die besten Individuen der Materialbewertungsebene ausgewählt und jeder Evolution der Planungsebene zur Verfügung gestellt. Auch hier wurde ein linearer Individuentyp verwendet und eine Turnierselektion verwendet die genauen Parameter der Evolution können im Abschnitt 6.1.2 nach gelesen werden. Nach Abschluss der Evolution der Planungsebne kann die Evolution der Variantenebene durchgeführt werden.

Der Unterschied der Evolution der Variantenebene zu den Evolutionen der beiden hierarchisch untergeordneten Ebenen liegt in der Fitnessberechnung. Hier werden keine Fitnesscases genutzt, an denen die Individuen trainiert werden, sondern es werden vollständige Schachspiele durchgeführt um die Fitness der Individiuen zu bestimmen. Die Schachspiele werden zwischen den Individuen durchgeführt, die während der Selektion ausgewählt werden. Ergebnisse und Probleme bei diesem Vorgehen sind in Kapitel 6.1.3 ausführlich beschreiben.

Im Abschnitt 6.1 und in [22] sind alle Ergebnisse des Systems und seine Probleme vollständig beschreiben.

5.1.2 qoopy

Das *qoopy-System* baut auf den Erfahrungen auf, die wir mit *ChessGP* gesammelt haben. Das Ziel war es hier, spielstärkere Individuen zu evolvieren, somit haben wir uns näher an einen Alpha-Beta Algorithmus gehalten und den Individuen nicht so viele Freiheiten gelassen wie im ChessGP-System. Denn viele der möglichen Freiheiten wurden vom System nicht genutzt. Außerdem führten sie zu einer sehr langsamen Evolution, da das System durch die vielen Freiheitsgrade sehr lange brauchte sinnvolle Kombinationen für ein Individuum zu erzeugen. Der Rahmen des Individuums sollte daher ein Alpha-Beta Algorithmus sein, der durch evolutionäre Verfahren verbessert werden sollte. Eine weitere Vereinfachung der Schachindividuen ist die Evolution der Individuen, anders als im ChessGP-System werden die Individuen hier in einem Schritt evolviert und nicht in drei aufeinander folgenden Evolutionen. Um diese zu ermöglichen, wird die Fitness der Individuen nicht durch Fitnesscases bestimmt, sondern durch vollständige Schachspiele gegen feste Schachprogramme verschiedenster Leistungsstufen, wodurch die Spielstärke der Individuen bestimmt werden konnte. Ergebnisse zu diesem System wurden in [43] veröffentlicht.

5.1.2.1 Das Schachindividuum

Im *qoopy-System* wurde ein Alpha-Beta Algorithmus [104, 56] als Kern eines Schachindividuums genutzt. Dieser Kernalgorithmus wurde durch GP- und EA-Module erweitert. Der Alpha-Beta Algorithums ist eine Heuristik, die für das Schachspiel erfolgreich im Mittelspiel verwendet wird. Der Algorithmus kann drei verschiedene Module nutzen, die über schachspezifisches Wissen verfügen. Das Stellungsbewertungsmodul wird genutzt, um eine erreichte Stellung zu bewerten. Diese Bewertung geht dann in die Alpha- und Beta-Werte ein und ermöglicht das Schneiden des Baumes. Eine weiteres Modul vieler Schachprogramme ist die Zugsortierung [41, 42, 3], sie ermöglicht es dem Alpha-Beta Algorithmus gute Züge zuerst zu expandieren, was zu einer Verkleinerung des Suchbaumes führt (siehe auch Abschnitte 2.2.5 und 2.2.3). Ein weiterer wichtiger Punkt für Schachprogramme ist die Beschränkung des Suchhorizontes [8, 15]. Normalerweise werden Programme dadurch beschränkt, dass ihnen eine maximale Suchtiefe erlaubt wird, die Anzahl der Knoten im Suchbaum hängt dann von der Spielsituation und dem Algorithum ab.

Im Fall des *qoopy-Systems* wurde die Suchtiefe zwar beschränkt, jedoch nur um es den Individuen während der Evolution unmöglich zu machen, 50 oder mehr Züge in die Zukunft zu schauen, was auch keinen Sinn macht, da eine Partie in der Regel dann schon beendet ist. Die Beschränkung der Suchtiefe schränkt das Individuum nicht in der Rechenkraft ein, bei einer maximale Suchtiefe von 10 kann das Individuum immer noch 10 Billiarden Knoten expandieren. Um die Rechenkraft der Individuen im qoopy-System zu beschränkten, wurde die Anzahl der im Suchbaum zu expandierenden Knoten für ein Individuum auf 100.000 Knoten im Durchschnitt beschränkt. Für diese Entscheidung sprachen die folgenden Gründe:

- Die Beschränkung ermöglicht eine akzeptable Evolutionsgeschwindigkeit, denn für die Fitnessberechnung eines Individuums müssen mehrere Spiele durchgeführt werden.
- Programme, die 100.000 Knoten expandieren haben schon eine akzeptable Spielstärke.

Die Evaluation des Schachindividuums erfolgt nach folgendem Schema. Zuerst wird das Tiefenmodul für die aktuelle Stellung aufgerufen, es berechnet die verbleibende Suchtiefe für die Stellung im Suchbaum. Wenn die Suchtiefe Null ist, wird die Stellung als ein Blatt interpretiert und das Stellungsbewertungsmodul wird aufgerufen. Der Wert der Berechnung wird nun der Stellung zugeordnet und während der Ausführung des Rahmenprogrammes genutzt. So wird dieser Wert für die $\alpha-$ und $\beta-$ Schnitte im Algorithmus genutzt und ob ein Zug in der Killertabelle aufgenommen wird oder nicht. (Zur Funktionsweise der Killertabellen siehe Abschnitt 2.2.6.4.) Wenn das Tiefenmodul eine Tiefe größer als Null berechnet hat, wird die nächste Ebene im Suchbaum geöffnet. Dazu werden durch das Rahmenprogramm alle Züge der aktuellen Stellung zur Verfügung gestellt,

```
ind_MAX (position K; integerα, β)
BEGIN
  integer i, j, value ;
  nodes = nodes + 1;
  IF Positionsmodue(K) THEN
   RETURN Positionsmodul(K);
  IF (depth == maxdepth) THEN
   RETURN Positionsmodul(K);
  Tiefenmodul restdepth;
  IF (((restdepth == 0) OR
   (nodes > maxnodes))
   AND depth >= mindepth) THEN
    RETURN Positionsmodul(K);
  determine successor positions K.1, ..., K.w;
  Zugsortireungsmodul(K.1, ..., K.w);
  value = α;
  FOR j = 1 TO w DO
  BEGIN
    restdepthBackup = restdepth;
    restdepth–;
    value= max(value, inf_MIN(K.j, value, β));
    restdepth= restdepthBackup
    IF value ≥ β THEN
     { ADJUST KILLERTABLE;
     RETURN value};
  END
  RETURN value
END
```

Tabelle 5.1: Der Pseudocode beschreibt den Rahmenalgorithmus für das Schachindividuum. Es ist der Max-Teil des Algorithumses, die evolutionären Module im Algorithmus sind fett gedruckt.

dann wird jeder mögliche Zug durch das Zugsortierungsmodul bewertet und anschließend die Zugliste nach ihrer Güte durch das Rahmenprogramm sortiert.

Für jede Stellung der Zugliste wird das Programm rekursiv wieder aufgerufen. Wenn die maximale Anzahl von Knoten erreicht wurde, wird für die noch nicht ausgewerteten Stellungen jeweils das Positionsmodul aufgerufen, die Bewertung durch den Spielbaum an die Wurzel propagiert und dort der *Beste Zug* zurückgegeben. Der Pseudocode 5.1 verdeutlicht noch einmal das Vorgehen des Rahmenprogrammes, hier wurde der *Max*-Teil des Algorithmus aufgeschrieben, der *Min*-Teil des Schachindividuums funktioniert analog (Zur Aufteilung eines Schachalgorithmuses in Max- und Min-Algorithmen siehe Abschnitte 2.2.4 und 2.2.5).

5.1.2.2 Das Stellungsmodul

Das *Stellungsmodul* bewertet eine gegebene Stellung, es handelt sich dabei um eine Materialbewertung, wie sie auch in Kapitel 2.2.1 beschrieben wird. Die Aufgabe dieses Moduls ist es, einen besseren Parametersatz für den Stellungsbewertungsalgorithmus zu finden. Die Werte, die normalerweise in solchen Algorithmen verwendet werden, gehen auf die Erfahrung des Programmierers oder auf die Literatur zurück, wobei hier immer noch Werte zu finden sind, die schon Shannon[2] für die Bewertung von Schachpositionen eingeführt hat.

Das Stellungsmodul im qoopy System ist kein GP-Individuum, sondern ein fest vorgegebener Algorithmus, der die Stellung anhand von evolvierten Werten für die verschiedenen Figuren und strukturellen Elemente bewertet. Hierzu wird eine einfache $(\mu + \lambda)$ Evolutionsstrategie mit fester Mutationsschrittweite verwendet [105]. Die Initialisierung der Vektoren war zufällig, jedoch wurden die Werte durch Intervalle begrenzt, die den einzelnen Merkmalen entsprachen, die Zuordnung der Intervalle zu den einzelnen Merkmalen werden weiter unten beschrieben. Bei der Rekombination wurden zwei Techniken genutzt. Zum einen die intermediäre Rekombination, bei der die reellen Zahlen, die zu denselbem Merkmalen gehören, gemittelt werden. Dabei ist die Wahrscheinlichkeit, dass ein bestimmtes Merkmal variiert wird, vom Benutzer einstellbar. Die zweite Möglichkeit ist die diskrete Rekombination, bei der sich eine Ausprägung des Merkmals durchsetzt.

Die Evolutionsstrategie soll Werte für die folgenden Merkmale der Stellungsbewertung optimieren:

1. die Werte für die einzelnen Figurtypen [0-1000]
2. einen Bestrafungswert für einen Läufer in der Anfangsposition [0-30]
3. ein Bonus für einen Bauern, der das Zentrum des Schachbrettes erreicht hat [0-30]
4. eine Bestrafung, wenn zwei Bauern auf der gleichen Linie stehen [0-30]
5. ein Bonus für einen Bauern, der auf beiden Nachbarlinien keinen gegnerischen Bauern hat [0-40]
6. eine Bestrafung für einen Bauern, der keinen Bauern auf den Nachbarlinien hat, der näher an der ersten Reihe des Gegners ist [0-30]
7. ein Bonus für einen Freibauern, der durch einen Turm beschützt wird [0-30]
8. ein Bonus für einen Bauern, wenn er in der Nähe der ersten Reihe des Gegners ist (Promotionsmöglichkeit) [100-500]
9. ein Bonus, wenn beide Läufer vorhanden sind [0-40]
10. eine Bestrafung für einen Läufer, der sich in der Startposition aufhält [0-30]
11. eine Bestrafung für Läufer in einer blockierten Stellung. Eine blockierte Stellung ist in diesem Zusammenhang eine Stellung, in der drei eigene Bauern sich im Bereich des erweiterten Zentrums befinden und Felder besetzen, die die gleiche Farbe haben wie die Felder auf denen sich der Läufer bewegt [0-40]
12. ein Bonus für den Springer, wenn es sich um eine blockierte Stellung handelt. Für einen Springer handelt es sich um eine blockierte Stellung, wenn sich mehr als sechs Bauern im erweiterten Zentrum aufhalten. Durch die Fortbewegungsweise des Springers wird er aufgewertet [0-50]
13. ein Bonus für den Springer, wenn er mehr als sechs Felder erreichen kann [0-30]
14. ein Bonus für den Springer in einer geschlossenen Position [0-40]
15. eine Bestrafung, wenn auf jeder Seite des Springers ein generischer Bauer steht [0-50]

[2] Shannon hat schon 1950 einen Artikel über den Aufbau von Schachprogrammen geschrieben, seine Vorstellungen sind bis heute in fast allen Programmen wiederzufinden.

16. ein Bonus für den Turm, falls er von dem anderen Turm gedeckt wird [0-20]
17. ein Bonus für den Turm, wenn er auf einer halboffenen Linie steht [0-30]
18. ein Bonus für den Turm, wenn er auf einer offenen Linie steht [0-30]
19. ein Bonus für den Turm für andere Vorteile [0-20]
20. eine Bestrafung für den König, wenn er die erste Reihe während des Eröffnungs- oder Mittelspiels verlässt [0-20]
21. ein Bonus, nachdem eine Rochade ausgeführt wurde [0-40]
22. eine Bestrafung für die Rochade, wenn es eine Schwäche in der Bauernstruktur vor dem König gibt [0-30]
23. eine Bestrafung, wenn die Möglichkeit einer Rochade vergeben wurde [0-50]
24. ein Zufallswert, der zum Wert der Stellung addiert oder subtrahiert wird [0-30]

Die Werte in den Klammern geben die Intervalle an, in denen sich die Werte für die Merkmale bewegen können. Diese Intervalle wurden vorgegeben, damit der Stellungsbewertungsalgorithmus schon mit zufälligen Werten noch relativ sinnvolle Bewertungen errechnet. Denn durch die verteilte Evolution, wie sie in Abschnitt 5.1.2.5 beschrieben wird, werden viele kleine Demes verwendet, so dass es sehr lange dauern würde, bis sich gute Parameter in der gesamten Population durchgesetzt hätten. Dies würde die Attraktivität des Systems für die Nutzer, die uns ihre Rechenkraft zur Verfügung stellen, senken. Daher sahen wir darin eine Gefahr wieder schnell einige Nutzer zu verlieren, wenn *ihre* Evolutionen geringere Fortschritte machen würden als die der anderen Nutzer.

Der Algorithmus summiert alle Figurwerte und die Boni, die für die Stellung zutreffend sind auf, anschließend werden alle Bestrafungswerte subtrahiert, die durch die Stellung erfüllt werden. Die Bonuswerte und die Bestrafungswerte sind für die strukturelle Untersuchung der Stellung verantwortlich. Hier ist auch die stärkste Einschränkung durch die vordefinierten Intervalle zu erkennen, denn diese Werte können zwar den Wert der reinen Materialbewertung vergrößern oder verringern, jedoch nicht den Verlust starker Materialwerte ausgleichen. Dieses Vorgehen ist auch sinnvoll, denn selbst Großmeister bewerten den Materialwert höher als die Struktur der Stellung (siehe auch [14]).

Dieser Algorithmus zur Stellungsbewertung wurde auch von den Gegnerprogrammen während der Fitnessevaluation genutzt, jedoch wurden bei diesen die Werte für die Gewichtung vorher festgelegt und während der gesamten Auswertung nicht verändert (siehe Abschnitt 5.1.2.7). Die Werte wurden hierbei mit Hilfe von Schachexperten bestimmt und durch Tests überprüft.

5.1.2.3 Das Zugsortierungsmodul

Das *Zugsortierungsmodul* hat die Aufgabe die möglichen Züge der aktuell betrachteten Stellung so zu gewichten, dass eine Sortierung der Züge nach ihrer Güte möglich ist (siehe auch 2.2.3). Die Sortierung der Züge erfolgt nach der Bewertung jedes Zugs durch einen Quicksort Algorithmus. Nachdem diese Schritte durchgeführt wurden, werden die Züge in der neuen Reihenfolge expandiert. Wenn der Algorithmus über eine gute Zugsortierung verfügt, ist es möglich die Anzahl der Knoten, die bei der Suche expandiert werden, stark einzuschränken. Denn werden bei der Alpha-Beta Suche zuerst die besten Züge[3] betrachtet, können die weiteren Züge mit einem stark eingeschränkten Suchfenster

[3]beste Züge im Sinne des Minimax-Prinzips

ohne qualitative Einbuße bearbeitet werden, dies führt zu einer stärkeren Beschneidung des Suchbaumes(siehe [51, 108, 104]). So expandieren selbst die spielstärksten Schachprogramme 20-30% mehr Knoten, als der optimale Suchbaum expandieren würde. Eine Verbesserung der Zugsortierung könnte somit eine starke Reduktion in der Größe des Suchbaums bedeuten.

Die Bewertung des Zugsortierungsmoduls ist unabhängig davon, ob ein Zug von der schwarzen oder weißen Seite durchgeführt wird. Jedem Zug wird eine reelle Zahl zugewiesen und anschließend absteigend sortiert. Die Güte des Zuges ergibt sich aus der Summe der Merkmale, die der Zug erfüllt. Der Algorithmus untersucht die folgenden Merkmale für jeden Zug:

- Werte, die den einzelnen Schachfiguren während der Eröffnungsphase zugeordnet werden
- Werte, die den einzelnen Schachfiguren während des Mittelspiels zugeordnet werden
- Werte, die den einzelnen Schachfiguren während des Endspiels zugeordnet werden
- Schlagzüge versus nicht schlagende Züge:
  Schlagzüge bekommen ein anderes Gewicht beigemessen als solche Züge, die keine Figur schlagen.
- Schachzüge:
  Züge, die den gegnerischen König ins Schach stellen, bekommen einen extra Bonus zugestanden.
- MVV/LVA (Most valuable victim/Least valuable aggressor):
  Dieser Wert ist das Verhältnis der Materialwerte der geschlagenen Figuren zu den schlagenden Figuren.
- Promotion:
  Züge, die einen Bauern umwandeln, bekommen einen Bonus, jedoch nur, wenn zu einer Dame gewandelt wird. Dies wird gemacht, da es sich hier nur um eine Zugsortierung handelt, so kann sich in der Suche noch herausstellen, dass die Umwandlung in eine andere Figur ratsamer ist.
- Zentrumsaktivität:
  Durch diese Merkmale wird dem Zug direkt ein Bonus zugesprochen, dabei hängt dieser nur vom Start- und Zielfeld des Zuges ab. Der Bonus ist die Summe des Bonus für das Zielfeld und der Hälfte des Bonus des Startfeldes. Die Boni der Felder ergeben sich durch die folgende Matrix:

0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	0
0	1	2	2	2	2	1	0
0	1	2	3	3	2	1	0
0	1	2	3	3	2	1	0
0	1	2	2	2	2	1	0
0	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0

- Bauernzüge:
  Bauernzüge, die eine gegnerische Figur bedrohen, erhalten einen Bonus.

- Killermoves:
  Killermoves sind Züge, die in einer früheren Situation zu einem Abschneiden eines Teiles des Suchbaumes geführt haben. Sie sind oft auch in anderen Situationen vorteilhaft, daher werden sie in einer sogenannten *Killer Table* gespeichert. Es werden maximal 4 Züge pro Ebene des Suchbaumes gespeichert.

Der Zusortierungsalgorithmus untersucht sowohl statische Elemente der Stellung als auch, durch die Killermoves und die Art des Zuges, dynamische Aspekte des Zuges. Die Bewertung des Zuges hängt aber kaum von der vorherigen Suche ab, dies kann nur durch die Killermoves geschehen, somit kann sich der Wert des gleichen Zuges in verschiedenen Spielen leicht unterscheiden, dies ist jedoch kein Problem, da die Zugsortierung nur zur Unterstützung beim Aufbau des Suchbaumes genutzt wird.

Um die Werte für die einzelnen Merkmale zu erhalten, wird ein $(\mu + \lambda)$ Evolutionsstrategie mit fester Mutationsschrittweite benutzt, wie sie auch schon beim Stellungsmodul verwendet wird. Es wurde keine ES mit einer Anpassung der Mutationsschrittweite genutzt, da man im allgemeinen nicht von der Spielstärke eines Individuums auf die Güte des Zugsortierungsmoduls oder des Stellungsmoduls schließen kann (siehe [105]). Daher ist eine Anpassung der Mutation durch die Fitness des Gesamtindividuums, als problematisch anzusehen und wurde daher nicht verwendet.

Bei der Mutation und der Rekombination kommen die gleichen Techniken zum Einsatz, wie sie für das Stellungsmodul eingesetzt wurden, also die intermediäre und die diskrete Rekombination. Bei der Mutation wird jedes Merkmal mit einer Wahrscheinlichkeit mutiert, die durch den Benutzer vorgebenen wurde.

5.1.2.4 Das Tiefenmodul

Das *Tiefenmodul* entscheidet für jede Position, ob die verbleibende Suchtiefe verringert, vergrößert oder nicht verändert wird. Diese Entscheidung wird nicht direkt getroffen, sondern indirekt über die *Restsuchtiefe* für jeden Knoten bestimmt. Die Restsuchtiefe wird im Pseudocode durch die Variable $restdepth$ bezeichnet, dort erkennt man auch, dass sie automatisch für jede Ebene des Suchbaumes um eins reduziert wird. Die Restsuchtiefe wird zu Beginn mit zwei initialisiert, dies soll die Evolution vereinfachen, denn dadurch werden auch Schachindividuen, die das Tiefenmodul nicht verwenden, mindestens einen Suchbaum der Tiefe zwei aufbauen können. Dies wurde aus zwei Gründen gemacht, der erste liegt in der internetbasierten Evolution begründet (siehe hierzu Abschnitt 5.1.2.5). Der zweite Grund bezieht sich auf das Stellungsmodul und das Zugsortierungsmodul, die sich so auch bei einem nicht aktiven Tiefenmodul in der Evolution weiterentwickeln können, da das Individuum mindestens die Suchtiefe zwei erreicht. Dies ist besonders zu Beginn der Evolution hilfreich, da so die Diversität des Demes nicht so schnell erodiert.

Das Tiefenmodul hat die Aufgabe dem Schachindividuum die Möglichkeit zu geben einen variablen Suchhorizont zu erzeugen. Normalerweise haben Schachprogramme einen festen Suchhorizont [97]. Das heißt, ein Programm darf maximal eine vorgegebene Anzahl von Ebenen im Suchbaum expandieren; wenn die maximale Tiefe erreicht wird, wird die Suche beendet und alle Knoten wie Blätter behandelt. Hierdurch ergeben sich Probleme, die auch als Horizonteffekt bekannt sind [59, 13].

Da das Tiefenmodul durch einen evolutionären Prozess entwickelt wird, gibt es zwei Parameter um die Rechenzeit eines Individuums zu beschränken. Die erste Schranke ist die Suchtiefe, die Maximaltiefe für die Individuen ist zehn. Diese sind ungefähr 10^{14} Knoten im Suchbaum, der expandierte Suchbaums der Tiefe zehn, mit einen guten Alpha-Beta Algorithmus, hätte immer noch 512 Millionen Knoten. Dies ist für einen evolutionären Prozeß zu viel, denn bis die Fitness eines Individuums feststeht, müssen mehrere Spiele durchgeführt werden, hierbei würde die Auswertung eines Spiels aber im Stundenbereich liegen. Daher wurde noch eine zweite Beschränkung eingefügt, sie beschränkt die Anzahl der Knoten im Suchbaum auf 100.000. Ein Individuum darf im Durchschnitt 100.000 Knoten pro Zug verwenden, das heißt, wenn es im x-ten Zug 50.000 Knoten verwendet hat, so darf es im nächsten Zug 150.000 Knoten verwenden. Dadurch kann die vollständige Fitnessauswertung eines Individuums innerhalb einiger Stunden durchgeführt werden.

Die Aufgabe des Tiefenmoduls ist es, anhand der gegebenen Stellung zu entscheiden, ob der mögliche Teilbaum genauer untersucht werden soll oder nicht. Diese Aufgabe kann man nicht durch einen evolutionären Algorithmus lösen, daher wurde an dieser Stelle ein GP-Programm verwendet. Als Struktur für das GP-Programm wurde die Linear-Graph Struktur verwendet (siehe 4.2). Das Funktionsset für dieses Modul wurde sehr einfach gehalten, die schachspezifischen Funktionen sind in das Terminalset eingegangen, eine genaue Auflistung ist im Anhang 8.2.

Das Tiefenmodul definiert die Suchtiefe für einen Knoten nicht direkt, sondern verändert wie schon gesagt eine Variable (*Resttiefe*), die angibt, wieweit die Suche für diesen Ast noch weitergeführt werden soll. Um die Variable, welche die Suchtiefe enthält, verändern zu können stehen dem GP-Individuum zwei Terminalfunktionen zur Verfügung. Die Funktion *incHorizon* erhöht den Wert der Resttiefe um eins und die Funktion *decHorizon* verringert den Wert um eins. Die Funktionen können im Individuum mehrfach ausgeführt werden und sie können auch beide nacheinander ausgeführt werden. Letzteres hat den Effekt, dass diese Befehlsfolge ein sematisches Intron ist, da der Wert der Variable Resttiefe sowohl mit als auch ohne diese Befehlsfolge die gleiche ist. Eine Einschränkung dieser Funktionen gibt es, die Resttiefe kann durch die Funktion *decHorizon* nicht kleiner null werden. Durch die Funktion *incHorizon* kann der Wert von der Resttiefe aber auch nicht zu groß werden, diese Einschränkung wurde gemacht, damit ein Individuum gezwungen wird nicht nur in der Tiefe zu suchen, sondern auch in der Breite. Werden diese beiden Funktionen im Tiefenmodul nicht aufgerufen, wird der Wert der Variable Resttiefe automatisch um eins reduziert. Der Wert der Variable Resttiefe wird zu Beginn der Auswertung des Tiefenmoduls mit zwei initialisiert. Dadurch wird erreicht, dass das Gesamtindividuum mindestens einen Suchbaum der Tiefe zwei aufspannt, selbst wenn das Tiefenmodul die Variable Resttiefe nicht verändert.

Wie schon oben erwähnt, besteht das GP-Programms aus der Linear-Graph Struktur (siehe 4.2). Die Branchingfunktionen in der Graphstruktur wurden durch schachspezifische Funktionen ersetzt und sie überwachen nicht das Ausgaberegister wie bei den Tests mit den Benchmarks. Dies würde für das Tiefenmodul auch wenig Sinn machen, da es kein Ausgaberegister im engeren Sinn gibt, denn die Ausgabe des Moduls ist die Veränderung oder Nichtveränderung der Resttiefe. Die Variable wird aber nicht durch ein Ausgaberegister verändert, sondern durch den Seiteneffekt zweier Terminalfunktionen.

Eine weitere Änderung für diese Anwendung war ein vermehrter Einsatz von *if-then* oder *if-then-else* Funktionen. Normalerwiese werden *if-then* Funktionen in linearen Individuen wie normale Funktionen behandelt, hier wurde aber eine erhöhte Wahrscheinlichkeit für diese Funktionen verwendet. Dies führt somit auch zum Einsatz spezieller Terminalfunktionen, die für die Bedingung in den *if-then* Funktionen entwickelt wurden. Im Abschnitt 8.2 sind die verschiedenen Funktionen und ihre Funktion genauer beschrieben, grundsätzlich handelt es sich aber um Funktionen mit Schachwissen.

Zur Mutation wurde im Unterschied zu den im Abschnitt 4.2.4 beschriebenen Verfahren noch eine weitere Methode eingesetzt. Dabei wurde ein Crossover mit einem Zufallsindividuum durchgeführt. Jedoch ist diese Methode vergleichbar mit der Mutation einer ganzen Sequenz von Knoten im Individuum. Daher wurde diese Methode auch nicht in diesem Kapitel beschreiben. Ein Nachteil dieser Methode ist, dass ein sehr großer Anteil des Individuums verändert wird und sie somit besonders in späteren Generationen hauptsächlich destruktiver Natur ist. Es wurde jedoch wegen der kleinen Populationen genutzt, die während der Evolution verwendet wurden. Somit soll einem zu schnellen Absinken der Diversität entgegengewirkt werden (hierzu siehe Abschnitt 5.1.2.5).

5.1.2.5 Evolution

Die Evolution des qoopy-Systems ist parallelisiert, da durch den großen Rechenzeitbedarf während der Evolution eine Auswertung mit nur einem Rechnern zu lange dauern würde. Die Parallelisierung ist jedoch nicht auf eine Auslagerung der Fitnessberechnung auf weitere Computer beschränkt, wie sie im ChessGP-System durchgeführt wurde. Im qoopy-System wird sie durch eine Aufteilung in Teilpopulationen erreicht.

Für das *qoopy-System* wurde der Ansatz des *verteilten Rechnens* verwendet, um die Parallelisierung zu erreichen (siehe [39, 30]). Im Fall des qoopy-Systems kann man das verteilte Rechnen als ein zentralisiertes P2P-System betrachten, bei dem ein zentraler Server *Aufgaben* an registrierte *Clients* verteilt und die Ergebnisse der dort erfolgten Berechnung zu einem späteren Zeitpunkt wieder erfasst und weiterverwendet. Der Server speichert unter anderem eine Liste der besten Individuen der gesamten Evolution, damit sie später von uns genauer analysiert werden können. Durch diesen Ansatz ist es möglich die Aufgabe auf verschiedenste Rechner im Internet zu verteilen und somit sehr viel Rechenkraft zur Verfügung gestellt zu bekommen. Ein sehr bekannter Ansatz für das verteilte Rechnen ist das *SETI@home* Projekt, bei dem Radiosignale nach Mustern außerirdischen Ursprungs durchsucht werden. Um den Teilnehmern an dem qoopy-System einen Anreiz zu bieten bei der Evolution mitzumachen, wurde wie schon oben gesagt nicht nur eine Fitnessberechnung an die *Clients* weitergegeben, sondern eine gesamte Teilpopulation. Die Tielnehmer konnten verschiedene Parameter der Evolution *ihrer* Teilpopulation verändern, so sollte eine Bindung zu dem Ergebnis ihrer Evolution aufgebaut werden. Die Teilnehmer der internetweiten Evolution konnten neben der Populationsgröße auch die Größe der Individuen und verschiedene Einstellungen zur Evolution wie Crossover, Mutation, Migrationsraten und andere Parameter einstellen.

Die besten Individuen der Populationen wurden dann zum Server geschickt und an andere Subpopulationen weitergegeben, um so eine Migration der besten Gene durch die Gesamtpopulation zu ermöglichen. Gleichzeitig wurden beim Server Information über die besten Individuen jedes Clients und der zur Verfügung gestellten Rechenzeit erfasst und auf Internetseiten für alle transparent dargestellt.

Dieses Vorgehen erreichte, wie von uns erhofft, eine starke Bindung der Teilnehmer an das Projekt und eine große und stabile Anzahl von Teilnehmern (mehr als 1600). Ein großer Nachteil war jedoch der geringe Einfluss, den wir auf die Evolution hatten. So konnten wir nicht mehrere Läufe durchführen, um so die Auswirkung verschiedener Parameter zu erkennen.

Die Defaultgröße einer Teilpopulation beträgt 20 Individuen, der Teilnehmer hat zwei Möglichkeiten eine Population zu initialisieren. Er konnte alle Individuen zufällig initialisieren oder Individuen aus anderen Populationen zu Initialisierung nutzen. Anschließend beginnt die Berechnung der Fitness der Individuen, die Fitness ist dabei in fünfzehn disjunkte Fitnessklassen eingeteilt, allerdings ist als Fitness jeder Wert von $0,0$ bis $15,0$ möglich. In Abschnitt 5.1.2.6 wird die Fitness der Individuen genauer beschrieben. Einem neuen Individuum wird die Fitness $1,0$ zugewiesen, der beste Fitnesswert im System ist $15,0$. Um die exakte Fitness eines Individuums zu bestimmen müssen eine gewisse Anzahl von Spielen gegen Schachalgorithmen bekannter Stärke durchgeführt werden. Dadurch benötigt die Fitnessberechnung für ein Individuum sehr lange, damit es aber schon während der Fitnessberechnung zur Evolution neuer Individuen kommen kann, werden die Individuen in drei Reifeklassen eingeteilt.

- Individuen werden als *normal* bezeichnet, wenn die Bewertung der Individuen noch nicht abgeschlossen ist. Individuen dieser Klasse können nicht ersetzt werden, aber auch nicht bei der Mutation und Rekombination herangezogen werden.

- Individuen werden als *reif* bezeichnet, wenn sie bei der Mutation und Rekombination als Eltern herangezogen werden dürfen. Sie können aber nicht überschrieben werden, da die Fitness noch nicht endgültig bewertet ist. Wann ein Individuum als reif bezeichnet wird, kann vom Benutzer eingestellt werden, als Defaultwert wurde die Fitness $5,0$ verwendet.

- Die letzte Klasse ist *fertig*, ab dieser Stufe ist die Fitnessbewertung des Individuums abgeschlossen und die Fitness ist bekannt.

Bis zum Zeitpunkt, da die Fitness eines Individuum als *fertig* bezeichnet wird, kann man den Fitnesswert als untere Schranke sehen, d.h. die Fitness des Individuums hat mindestens diesen Wert, kann aber auch höher sein. Eine weitere Reifeklasse eines Indivduums ist die Klasse *Spielschwach*, diese Klasse hat einer Sonderstellung, da sie nur für neue Individuen zutreffen kann. Ein Individuum wird als *Spielschwach* klassifizeirt, wenn für ein Individuum während des ersten Spiels weniger als 50% der Züge als *gut* betrachtet werden. Wenn dies geschieht, wird die Partie sofort abgebrochen und das Individuum aus der Population entfernt. Dies wird gemacht um die eingesetzten Ressourcen auf gute Individuen zu konzentrieren.

Als Selektion wird eine Tunierselektion zwischen *reifen* Individuen durchgeführt, anschließend kommt es dann zur Rekombination, Mutation oder Erzeugung neuer Individuen. Die Wahrscheinlichkeiten für die Auswahl der Methoden kann durch den Benutzer gewählt und auch während der Evolution verändert werden. Die Erzeugung neuer Individuen wurde als eine weitere Möglichkeit der Evolution hinzugefügt, da man durch die sehr kleinen Populationen einer zu geringen Diversität entgegenwirken wollte.

Die Migration von Individuen ist ein weiterer wichtiger Punkt für die internetweite Evolution, hierbei gibt es zwei Möglichkeiten der Migration, die automatische Migration von Individuen zwischen Populationen und die durch den Teilnehmer initiierte Migration, die durch eine Anfrage an den Server gestartet wird. Beiden Methoden ist gemeinsam, dass lediglich Kopien der Populationsmitglieder versendet werden, dadurch bleibt die Population des Senders unverändert. Da die Evolution der Clients auch dann weiter läuft, wenn der Client nicht mit dem Internet verbunden ist, werden die Individuen, die als Migranten ausgewählt wurden, in einer Queue gespeichert. Die Queue hat eine Länge von zehn, wenn sie vollständig gefüllt ist, wird immer das älteste Individuum aus der Queue gelöscht, wenn ein neues Individuum als Migrant in die Queue gestellt wird. Somit ist die Queue immer mit aktuellen Individuen gefüllt, wenn ein Versenden der Individuen möglich ist. Dem Anwender ist es nicht möglich das Versenden von Individuen zu unterbinden, jedoch kann er die Migration neuer Individuen in die Population unterbinden, um so eine isolierte Evolution durchzuführen. Da der Anwender das Versenden von Individuen nicht unterbinden kann, ist es den anderen Teilpopulation weiter möglich an den Ergebnissen, die von diesem Client erzeugt werden, zu partizipieren.

Die Evolution des Gesamtsystems hat ein offenes Ende, sie wird nicht nach einer festen Anzahl von Generationen abgebrochen.

5.1.2.6 Fitnessberechnung

Der Fitnesswert eines Individuums ist eine reelle Zahl zwischen $0,0$ und $15,0$, wobei höhere Werte mit besseren Individuen korrespondieren. Jede ganze Zahl zwischen 0 und 15 steht für eine disjunkte Fitnessklasse und die Nachkommastellen bestimmen die Güte des Individuums in der Fitnessklasse, wobei auch hier ein größerer Wert eine bessere Güte bezeichnet.

Die Fitnessklasse wird durch die Algorithmen bekannter Stärke gebildet, diese Algorithmen liegen in den Fitnessklassen 2, 4, 6, 8, 10, 12 und 14. Alle ungeraden Fitnessklassen entsprechen einer Stärke, die zwischen den Spielstärken der festen Algorithmen liegen. Die Fitnessklassen der festen Algorithmen werden durch die maximale Suchtiefe des Algorithmuses gebildet. So wird die Fitnessklasse 2 durch den Algorithmus mit der Suchtiefe 1 gebildet und die Fitnessklasse 4 durch den Algorithmus mit der Suchtiefe 2. Die Fitnessklasse ist also Suchtiefe des festen Algorithmus mal zwei. Der Schachalgorithmus zur Fitnessberechnung wird in Abschnitt 5.1.2.7 genauer beschrieben.

Das Ergebnis eines Spiels wird durch eine reelle Zahl zwischen -1 und 1 beschrieben, wobei -1 anzeigt, dass das Individuum verloren hat und der Standard Algorithmus gewonnen hat. Ein Remis wird durch eine 0 beschrieben. Da ein Spiel nach maximal 50 Halbzügen abgebrochen wird, kann es vorkommen, dass der Sieger der Partie noch nicht feststeht. In diesem Fall wird die Stellung mittels einer Materialbewertung bewertet und dieser Wert als Ergebnis der Partie zurückgegeben. Der Wert der Materialbewertung liegt immer zwischen $[-1.0, \dots, 1.0]$ und gibt das Verhältnis des Materialvorteils des Individuums zum maximalen Materialwert ohne den König an, wobei ein Materialvorteil für den weißen Spieler durch positive Werte dargestellt wird und für den schwarzen Spieler mit negativen Werten.

Die Fitness eines Individuums wird mit eins initialisiert und dann wie folgt berechnet:

$$fitness = \frac{\sum_{j \in \mathcal{C}}^{m} \sum_{i=1}^{n} \text{class}_j^{\text{of result}_i} + \text{result}_i}{n * m}$$

Die Klassen $\mathcal{C}$ teilen sich wiederum in zwei Klassen auf, die erste Klasse enthält alle gewonnenen Spiele und die zweite Klasse die Remis und verlorenen Spiele. Ein Spiel wird für weiß als gewonnen betrachtet, wenn die Bewertung der Partie einen Wert größer als $0,3$ ergibt und als Remis, wenn der Wert zwischen $0,3$ und $-0,3$ liegt, alle anderen Bewertungen werden als verlorene Spiele gedeutet. Der Wert $0,3$ entspricht einen Materialvorteil von ungefähr 3 Bauern oder einer schweren Figur[4]. Dies ist ein sehr hoher Materialvorteil, da bei guten Spielern schon ein Materialvorteil von einem Bauern oft zur Aufgabe der Partie reicht. Die Klassen eines Individuums, die zur Fitnessberechnung herangezogen werden, werden durch die folgenden zwei Regeln bestimmt:

- Wenn ein Individuum alle Spiele bis zur Klasse k gewonnen hat, werden die Ergebnisse der Klasse $k-1$ ignoriert.

[4]Schwere Figuren entsprechen hier einem Läufer, Pferd und Turm.

- Wenn ein Individuum alle Spiele ab der Klasse k verliert, werden diese Ergebnisse ignoriert.

Um die Berechnung der Fitness zu verdeutlichen, hier zwei Beispiele. Das erste Beispiel zeigt die Berechnung der Fitness in einer Klasse, wenn ein Individuum gegen einen Gegner der Klasse 6 zweimal verliert und einmal gewinnt $(-1, -1, 1)$, dann ergeben sich daraus für das Individuum folgende Werte $(5, 0, 5, 0, 7, 0)$, diese ergeben dann den Fitnesswert $5,667$.

Wenn man das Beispiel nun auf mehrere Klasse erweitert, ergibt sich folgendes Bild. Ein Individuum i gewinnt alle Spiele der Klassen 2, 4 und 8, weiterhin erspielt es Gewinne, Remis und Niederlagen in den Klassen 6 und 10. In den Klassen 12 und 14 verliert es alle Spiele, dann enthält $\mathcal{C}_i$ die Klassen 6, 8 und 10. Die erste Regel definiert eine untere Grenze, welche Klasse noch zur Fitnessberechnung genutzt wird, in diesem Fall ist es die Klasse 4, die zweite Regel ist eine obere Grenze und definiert in diesem Beispiel die Klasse 12 als Grenze. Anschließend kann die Fitness des Individuums mit der oben vorgestellten Formel berechnet werden.

Ein weiteres Merkmal der Fitnessberechnung im qoopy-System ist die Unterteilung der Berechnung in vier Phasen, wobei jede Phase andere Fitnessklassen enthält und bestimmt, wie oft die Individuen ausgewertet werden. Diese Vorgehen wurde gewählt, da die Berechnung der Fitness oder allgemeiner die Auswertung eines Schachspiels mit jeder Klasse immer rechenintensiver wird und somit mehr und mehr Zeit verbraucht.

In der ersten Phase spielt ein Individuum zwei Spiele gegen Programme der Klasse *zwei*. Die Individuen dürfen kein Spiel in dieser Klasse verlieren, da sie sonst direkt ersetzt werden. Durch diese Regel können schon in einer sehr frühen Phase schwache Individuen herausgefiltert werden. Dadurch kann das System die Rechenzeit für aufwendigere Fitnessberechnungen der höheren Klassen einsparen. In der zweiten Phase spielt das Individuum gegen Programme der Klassen 4, 6, 8 und 10. Wenn das Individuum am Ende dieser Phase mindestens eine Fitness von $4,5$ hat, wird die Fitnessberechung für das Individuum in der dritten Phase fortgesetzt, sonst wird die Fitnessberechung beendet und der erreichte Wert wird als Fitness des Individuums betrachtet. In der Phase drei spielen die Individuen 1-2 Spiele gegen Programme der Klassen 2 bis 10. Starke Individuen, die schon in der zweiten Phase Spiele gegen Programme der Klassen 6 bis 10 gewonnen haben, spielen nur gegen Programme der Klassen 6 bis 10. Wenn in dieser Phase alle Spiele gewonnen werden, wird die vierte Phase der Fitnessberechung begonnen, sonst wird die aktuelle Fitness als endgültige Fitness betrachtet. Die vierte Phase der Fitnessberechnung erreichen nur die stärksten Programme, dies ist auch sehr wichtig, da die Spiele gegen die Programme der Klasse 12 und 14 sehr rechenzeitintensiv sind (siehe hierzu 2.2). Die maximale Anzahl der Spiele in dieser Phase wird durch ein Punktesystem bestimmt. Jedes Individuum hat 6 (5) Punkte für Spiele in der Klasse 12 (14) zur Verfügung; wenn die Punktzahl erreicht wird oder je zwei Spiele in jeder Klasse gewonnen werden, wird die Fitnessberechung beendet. Denn ein Individuum erhält für jedes Remis einen Punkt und für jedes verlorene Spiel zwei Punkte.

Es werden immer mehrere Spiele in jeder Klasse gespielt, da die Individuen durch verschiedene Funktionen nicht hundertprozentig deterministisch agieren. Daher kann ein Individuum Spiele gegen das gleiche Programm einer Klasse sowohl gewinnen als auch verlieren. Die Variation des Spielverhaltens ist jedoch nicht so dramatisch, dass ein Individuum alle Spiele gegen Programme der Klassen 2 - 8 verlieren kann und Spiele der höheren Klassen gewinnt. Der Nicht-Determinismus kommt durch eine kleine Variation der Materialbewertung im Stellungsmodul zustande (siehe Abschnitt 5.1.2.2).

Die Fitnessberechung wird nach den Spielen gegen Programme der Klasse 14 eingestellt, da schon sehr effiziente Schachprogramme der Klasse 14, also Suchtiefe sieben, ungefähr

2 Millionen Knoten im Suchbaum expandieren. Programme der nächsten Klasse würden 13 Millionen Knoten expandieren, den GP-Individuen ist es aber nur erlaubt maximal 100.000 Knoten pro Zug zu expandieren, somit ist es unwahrscheinlich, dass das System ein GP-Programm entwickeln kann, welches ein Programm der Klasse 16 schlagen könnte. Denn bei den Individuen handelt es sich auch um Baumsuchverfahren, somit werden Programme der gleichen Klasse von dem System entwickelt und es ist dem System nur möglich eine Optimierung von Baumsuchverfahren durchzuführen. Daher gehen wir davon aus, dass es dem System nicht möglich ist ein Verfahren zu entwickeln, welches um den Faktor 130 besser ist als die bekannten Verfahren.

5.1.2.7 Der Gegner des Individuums

Wie schon im Kapitel 5.1.2.6 besprochen, werden zur Fitnessberechnung Programme verschiedener Stärken benötigt, die zur Bestimmung der Fitness genutzt werden. Hierzu werden Schachprogramme genutzt, die den Suchbaum bis zu einer festgelegten Tiefe aufspannen dürfen. Für die Fitnessbestimmung haben wir die Suchtiefen 1 bis 7 festgelegt, wobei diese mit den Fitnessklassen 2 bis 14 korrespondieren.

Die Schachprogramme zur Fitnessevaluation nutzen für die Materialbewertung und die Zugsortierung die gleichen Funktionen wie die Individuen, jedoch wurden die Gewichte für die einzelnen Parameter vorher festgelegt. Wir haben uns dabei an Werte aus der Literatur und von guten Schachspielern orientiert. Ein wichtiger Unterschied zu den Individuen besteht noch darin, dass die Programme keine Zufallswerte nutzen.

Da diese Programme sehr oft aufgerufen werden, sollten sie sehr effizient sein, d. h. eine sehr geringe Anzahl von Knoten im Suchbaum expandieren. Daher wurde der *F-Negascout* Algorithmus verwendet, der mit einem *iterative deepening* Ansatz kombiniert wurde (siehe [102]). Der *F-Negascout* Algorithmus ist ein sehr effizienter Algorithmus, der auf dem Alpha-Beta Verfahren basiert (siehe Abschnitt 2.2.4). Jedoch wurde anstelle der Nullfenstersuche mit einem Fenster der Größe $(\alpha, \alpha + 0,00001)$ eine Fenstergröße genutzt, die 20% des Wertes eines Bauern entspricht, somit erhält man ein Fenster $(\alpha, \alpha + 20)$. In Simulationen hat sich diese Einstellung gegenüber der Nullfenstertechnik als überlegen erwiesen. Denn je größer das Suchfenster ist, desto geringer ist die Wahrscheinlichkeit, dass eine Wiederholungssuche erforderlich ist, da der ermittelte Wert öfter innerhalb des Suchfensters liegt. Die Wiederholungssuche arbeitet mit einem einseitig beschränkten Suchfenster und ist demnach meist sehr aufwendig. Der Nachteil des größeren Suchfensters ist, dass die normale Suche etwas langsamer abläuft als mit einem Nullfenster. Jedoch überwiegen hier die Vorteile die Nachteile klar.

Die Zugsortierung wird mittels der Technik des *Iterative Deepening* erreicht (vgl. Abschnitt 2.2.6.1). Zusätzlich werden noch die *killer moves* genutzt um eine weitere Einsparung von Zügen bei der Berechnung zu erreichen (vgl. Abschnittsub:KillerMove).

Die Materialbewertung nutzt die in Abschnitt 5.1.2.2 beschriebene Funktion. Die Werte für die einzelnen Parameter der Funktion sind für den Algorithmus vorgegeben und haben sich aus der Literatur, Erfahrungen von Schachspielern und Tests ergeben.

1. Bauer [100]
2. Springer [340]
3. Läufer [340]
4. Turm [500]
5. Dame [900]

6. Bestrafung für einen Läufer in der Anfangsposition [15]
7. Bonus für einen Bauern, der das Zentrum des Schachbrettes erreicht hat [15]
8. Bestrafung, wenn zwei Bauern auf der gleichen Linie stehen [30]
9. Bonus für einen Bauern, der auf beiden Nachbarlinien keinen gegnerischen Bauern hat [20]
10. Bestrafung für einen Bauern der keinen Bauern auf den Nachbarlinien hat, der näher an der ersten Reihe des Gegners ist [15]
11. Bonus für einen Freibauern, der durch einen Turm beschützt wird [20]
12. Bonus für einen Bauern, wenn er in der Nähe der ersten Reihe des Gegners ist (Promotionsmöglichkeit) [300]
13. Bonus, wenn beide Läufer vorhanden sind [20]
14. Bestrafung für einen Läufer, der sich in der Startposition aufhält [15]
15. Bestrafung für Läufer in einer blockierten Stellung. Eine blockierte Stellung ist in diesem Zusammenhang eine Stellung, in der drei eigene Bauern sich im Bereich des erweiterten Zentrums befinden und Felder besetzen, die die gleiche Farbe haben wie die Felder, auf denen sich der Läufer bewegt [20].
16. Bonus für den Springer, wenn es sich um einen blockierte Stellung handelt. Für einen Springer handelt es sich um eine blockierte Stellung, wenn sich mehr als sechs Bauern im erweiterten Zentrum aufhalten. Durch die Fortbewegungsweise des Springers wird er aufgewertet [20].
17. Bonus für den Springer, wenn er mehr als sechs Felder erreichen kann [15]
18. Bonus für das Pferd in einer geschlossenen Position [20]
19. Bestrafung, wenn auf jeder Seite des Pferdes ein generischer Bauer steht [25]
20. Bonus für den Turm, falls er von dem anderen Turm gedeckt wird [15]
21. Bonus für den Turm, wenn er auf einer halboffenen Linie steht [15]
22. Bonus für den Turm, wenn er auf einer offenen Linie steht [15]
23. Bonus für den Turm für andere Vorteile [10]
24. Bestrafung für den König, wenn er die erste Reihe während des Eröffnungs- oder Mittelspiels verlässt [15]
25. Bonus, nachdem eine Rochade ausgeführt wurde [20]
26. Bestrafung für die Rochade, wenn es eine Schwäche in der Bauernstruktur vor dem König gibt [15]
27. Bestrafung, wenn die Möglichkeit einer Rochade vergeben wurde [20]
28. Zufallswert, der zum Wert der Stellung addiert oder subtrahiert wird [20]

5.2 Das GeneticChess System

Das *GenticChess System* ist wie die Vorgängersysteme ein spezialisiertes GP System mit einem sehr engen Fokus, der auf die Aufgabenstellung ausgerichtet ist. Aus diesem Grund wurden verschiedenste Techniken entwickelt, aus anderen Bereichen übernommen und angepasst, um *gut spielende* Schachindividuen zu erzeugen. Auch hier wird der Grundgedanke dieser Arbeit weiterverfolgt, der versucht evolutionäre Verfahren soweit zu erweitern und unterstützen, dass sie auch komplexe Probleme in einer *annehmbaren Zeit* entwickeln und lösen können, was mit den *Standardverfahren* nicht möglich ist. Das System verwendet die *State-Space* Struktur für die Individuen, die Struktur wurde in Abschnitt 4.3 vorgestellt. Diese Struktur zielt auf die Evolution von Problemen, die als Suchproblem dargestellt werden können, im Fall des Schachproblems müssen dem Individuum aber noch zusätzliche Funktionen und Datenstrukturen zur Verfügung gestellt werden, damit sie das Problem lösen können. In der Literatur und Praxis sind schon Verfahren und Funktionen bekannt, die das Problem oder Teilprobleme lösen können. Die Güte dieser Verfahren variiert stark und ist von den eingesetzten Verfahren und ihrer Kombination abhängig. Diese Kombination sollen nun durch die Evolution erprobt werden.

5.2.1 Erweiterungen für das GeneticChess System

Im folgenden werden nun die Erweiterungen des GP Systems und des Individuums besprochen. Hierbei steht eine starke Spezialisierung auf das Schachproblem im Vordergrund. Um das Problem adäquat zu lösen, benötigt man einige Datenstrukturen, die normalerweise ein GP System nicht verwendet. Hierbei sind neben schachtypisichen Datenstrukturen, wie das Schachbrett, ein Zug oder ein Spielbaum, auch Elemente eingeführt worden, die eine Anpassung des GP Systems an die Funktionen und Module darstellten, hier sind die Datenstruktur der Register und das Individuen ROM zu erwähnen.

5.2.1.1 Datenstrukturen

Eine weitere Notwendigkeit ist die Einführung komplexerer schachspezifischer Datenstrukturen, die durch verschiedene Funktionen und Terminale manipuliert werden. Die wichtigsten Datenstrukturen sind hierbei der Suchbaum, ein Schachzug und das Schachbrett. Der Suchbaum ist ein Bestandteil der State-Space Struktur und wird während der Auswertung durch den Interpreter aufgebaut, siehe hierzu auch Abschnitt 4.3.1. Der Suchbaum besteht dann aus den Schachbrettern als Knoten und den möglichen Zügen zwischen den Stellungen als Kanten.

Das Schachbrett ist eine Bitboard Implementierung, das heißt, ein Brett wird durch mehrere 64-Bitzahlen im Speicher dargestellt. Dies ermöglicht es sehr schnell, Züge durchzuführen und wieder zurückzunehmen, da diese durch xor-Verknüpfungen erfolgen. Da man durch eine 64-Bitzahl nur die Positionen feststellen kann, an denen sich die Figuren auf dem Schachbrett befinden, benötigt man mehrere 64-Bitzahlen um ein ganzes Schachbrett abzubilden [1, 100]. Insgeamt benötigt man neun 64-Bitzahlen, eine mit allen Figuren, eine nur mit den schwarzen Figuren, eine nur mit den weißen Figuren und eine für jeden Figurtyp. Durch und-Verknüpfungen der verschiedenen Zahlen kann man die Position einer Figur ermitteln. Diese Implementierung ist sehr schnell und ermöglicht es viele Stellungen pro Sekunde zu betrachten, besonders vorteilhaft ist das Verfahren bei der Verwendung von 64-Bit Prozessoren.

Der Zug ist eine weitere wichtige Datenstruktur im Schachspiel, er besteht hier aus dem Startfeld, dem Zielfeld und einem Gütewert. Das Startfeld und Zielfeld wird durch eine Zahl zwichen 0 und 63 dargestellt und der Gütewerte ist eine reelle Zahl. Der Gütewert ist grundsätzlich nicht für die Datenstruktur notwendig, da es aber im System zu einer Zugbewertung kommt, ist es sinnvoll diesen Wert direkt im Zug zu speichern um keine zusätzliche Struktur zu nutzen.

5.2.1.2 Individuen ROM

Der *Individuen ROM* ist eine Erweiterung des Individuums, er besteht aus einem Array mit reellen Zahlen. Das Individuum kann auf diese Werte nur lesend und nicht schreibend zugreifen. Damit die Werte sich während der Evolution dennoch verändern können, werden sie während der Crossover oder der Mutation des Individuums mit verändert. Für das ROM gibt es keine explizite Fitnessfunktion, die Fitness des ROM's ergibt sich implizit aus der Fitness des Gesamtindividuums. Die Werte des ROM's werden als Funktionsparameter interpretiert und sind somit für die Fitness des Individuums für besondere Bedeutung. Hierbei handelt es sich um Funktionen, die für die Materialbewertung eingesetzt werden, siehe Abschnitt 5.2.2.1. So werden den Figuren Werte zugeordnet, mit deren Hilfe eine Bewertung der Stellung durchgeführt wird. Dadurch ist die Vorgehensweise die Fitness des ROM's an die Fitness des Individuums zu koppeln zweckmäßig, denn ohne diese Funktionen wäre es dem Individuum nicht möglich die Aufgabe zu erfüllen. Die Annahme ist, dass schlechte Parameter auch zu schlechten Individuen führen.

5.2.2 Das Schachindividuum

Die *State-Space* GP-Struktur ist der zentrale Motor des GeneticChess-Systems, die durch verschiedene schachspezifische Elemente erweitert wurde, damit das Individuum Schachprogramme evolvieren kann. Diese Erweiterungen sind die spezialisierten Datenstrukturen wie das Schachbrett, der Zug und der Individuen ROM, wie sie oben beschrieben wurden. Weitere Spezialisierungen für das Problem sind die Schachfunktionen für die einzelnen Module der State-Space Struktur.

Die *Modularisierung* der GP Struktur ist ein wichtiger Punkt des GeneticChess Systems. Die Modularisierung soll dem System die Möglichkeit geben das Zielproblem zu lösen, indem es Lösungen für Teilprobleme sucht. Um die Suche nach einer Lösung zu unterstützen wird die Aufteilung in die Teilprobleme nicht durch die Evolution entwickelt, sondern von dem System vorgeben. Hierdurch wird dem System einerseits die Möglichkeit genommen eigene Lösungswege zu finden, jedoch wird den Individuen auch die Möglichkeit gegeben schneller Lösungen zu erreichen, wie wir es aus den Erfahrungen des GeneticChess und qoopy-Systems erkannt haben. Da Schach selbst ein komplexes Problem ist, kann man davon ausgehen, dass man durch den Einsatz von einfachen GP Individuen nicht in annehmbarer Zeit zu einem Ergebnis kommen kann oder vielleicht auch nie zu einem annehmbaren Ergebnis kommt. Für das Schachproblem sind die Teilprobleme und die Interaktion der Teilprobleme klar definiert, so dass die Aufgabenstellung der Module und deren Interaktion in einem Gesamtindividuum klar definiert sind. Diese Module sind untereinander durch eine hierarchische Beziehung verbunden, das heißt, dass ein Modul einer höheren Hierarchieebene die Ergebnisse der Module einer niedrigeren Hierarchieebene benötigt.

Ein hierarchisches Modul unterscheidet sich von *ADF's* (automatisch definierte Funktionen), wie sie Koza in [74] eingeführt hat. Diese Module sind Teilfunktionen, die sich während der Evolution herausbilden und über deren Funktionalität zu Beginn der Evolution keine Informationen vorhanden sind. Die Funktionalität dieser ADF's wird nur durch ihre Funktions- und Terminalmenge vorgegeben. Meistens haben sie aber die gleichen Funktions- und Terminalmengen wie das Hauptindividuum. Daher wird die Funktion eines ADF's erst in dem Evolutionsprozess herausgebildet und unterscheidet sich stark zwischen Individuen verschiedener Evolutionsläufe. Eine weitere Methode Module in GP Systeme zu verwenden wurden von Angeline in [4] vorgestellt. Dabei werden während der Crossoveroperation Teilbäume gekapselt, vor weiterer Veränderung geschützt und als *Libraries* bezeichnet. Auch hier werden die Module innerhalb der Evolution erzeugt und können von den gegebenen Ressourcen nur eine Teilfunktion dieser Aufgabe sein.

Im GeneticChess System wird die Funktionalität eines Moduls vorher festgelegt, so ist ein Modul nicht eine zufällig entstandene Funktion sondern ein gezielt entwickeltes Modul eines komplexeren Gesamtindividuums oder Gesamtprogramms. Die Module unterscheiden sich nicht nur in ihrer Aufgabenstellung sondern auch in ihren Funktions- und Terminalmengen. Das GeneticChess System soll einen Alpha-Beta Algorithmus evolvieren, somit braucht das Individuum drei Module:

- das Materialbewertungs Modul,
- das Zugbewertungs Modul und
- das Tiefen Modul.

Als GP-Struktur wird die State-Space Struktur verwendet, die in Kapitel 4.3 vorgestellt wurde.

5.2.2.1 Materialbewertungs Modul

Das *Materialbewertungs Modul* hat die Aufgabe, die Analyse einer Schachposition durchzuführen. Dies ist im Schachspiel eine sehr wichtige Aufgabe, da während der Suche eine Bewertung gebraucht wird, um jede betrachtete Stellung im Spiel zu bewerten. Dies ist notwendig, da die Berechnung eines Zuges nicht bis zum Remis oder Matt des Spielzuges durchgeführt werden kann. Die Suche muss mitten im Spiel abgebrochen werden und diese Situation muss dann bewertet werden. Die Bewertung einer Stellung beruht somit auf unvollständigen Informationen, denn eigentlich ist die Bewertung nicht von der statischen Situation abhängig, sondern von dem dynamischen Verhalten des Spiels der nächsten Züge. Da die Materialbewertung aber sehr oft in den Algorithmen aufgerufen wird, muss sie auch schnell sein und kann somit nur die statische Situation bewerten. Die einfachste Variante ist die Differenz der Summe der Materialwerte beider Seiten. In der Literatur findet man aber verschiedene Verfahren, die wichtigsten davon werden in Kapitel 2.2.1 genauer erklärt.

Eine einfache Materialbewertung bietet jedoch sehr viel Raum für Verbesserungen, so können verschiedene positionelle Kriterien berücksichtigt werden um eine Verbesserung der Bewertung zu erzeugen. Die einfache Materialbewertung benutzt nur die reinen Materialwerte der Figuren für die Bewertung. Dem GP Modul wird hier die Möglichkeit gegeben die Werte für die einzelnen Figuren selbst anzupassen. Zu den reinen Materialwerten kann das Modul aber auch positionelle Werte in die Materialbewertung einbeziehen. Dem Modul stehen hierzu Funktionen zur Verfügung, die folgenden Kriterien in die Materialbewertung mit einbeziehen können:

Zentrumsbauer Ein Bonus, wenn Bauern die Zentrumsfelder der entsprechenden Seite besetzen. Die Zentrumsfelder sind d3-6 und e3-6. Zusätzlich hat weiß noch die Felder c4-5 und schwarz f4-5. Die Idee der Zentrumsfelder ist, dass Figuren im Brettzentrum größeren Einfluss ausüben als am Rand.

Doppelbauer Ein Malus, wenn zwei gleichfarbige Bauern dieselbe Linie besetzen.

Freibauer Ein Bonus für Bauern, die an einer Umwandlung nicht mehr direkt von gegnerischen Bauern gehindert werden können.

Rückständiger Bauer Ein Malus für Bauern, die sich nicht mehr in der Grundstellung befinden und keinen gleichfarbigen Bauern auf den benachbarten Linien haben, die näher an der Grundstellung sind.

Springermobilität Ein Bonus, der die Mobilität des Springers berücksichtigt. Denn je nach Position kann ein Springer 2,3,4,6, oder 8 Felder erreichen. Das Merkmal, wieviele Felder ein Springer erreichen kann, erhöht den Bonus für den Springer.

Springerbonus Dieser Bonus wird gewährt, wenn sich im erweiterten Zentrum bestehend aus 16 Feldern, mehr als 6 Bauern aufhalten, hat der Springer aufgrund seiner Fortbewegung im Gegensatz zu den Läufern einen Vorteil, der durch diesen Wert berücksichtigt wird.

Blockierter Läufer Ein Malus, wenn im erweiterten Zentrum (Quadrat aus 16 Feldern in der Mitte des Brettes) mindestens 3 eigene Bauern mit der gleichen Feldfarbe wie der Läufer stehen. Diese Bauern beeinträchtigen die Mobilität des Läufers, gegnerische Bauern zählen nicht, da sie geschlagen werden können.

Läuferpaar Ein Bonus, der vergeben wird, solange beide Läufer vorhanden sind. Denn nur beide können alle Felder des Schachbrettes erreichen.

Läufer Startposition Ein Malus für einen Läufer, der in der Startposition verbleibt. Für den Springer wird dieses Kriterium über die Mobilität bewirkt und für den Turm über die Rochademöglichkeit.

Turm auf halboffener Linie Ein Bonus, wenn der Turm auf einer Linie steht, auf der sich kein eigener, sondern nur ein gegnerischer Bauer befindet.

Turm auf offener Linie Ein Bonus wenn der Turm auf einer Linie steht, auf der sich kein eigener und kein gegnerischer Bauer befindet.

Freibauer und Turm Ein Bonus für einen Freibauern, wenn er durch einen Turm geschützt wird, der auf der gleichen Linie steht und damit die Promotionschancen erhöht.

Turm Ein Bonus, falls sich Türme gegenseitig decken oder auf Reihe 1 oder 2 stehen.

Rochade verpasst Ein Malus, falls in einer Stellung eine lange oder kurze Rochade möglich war, diese aber nicht ausgeführt wurde.

Rochade durchgeführt Ein Bonus, falls eine Rochade durchgeführt wurde.

Bauernstruktur Ein Malus für eine Schwäche in der Bauernstruktur nach einer Rochade. Das heißt, die schützenden Bauern bleiben in der Grundstellung. Dieser Malus wird im Endspiel nicht mehr verwendet.

König Grundlinie Ein Malus, wenn der König die Grundlinie verlässt, dies gilt nicht für das Endspiel.

Zusätzlich werden noch verschiedene positionelle Kriterien nur im Endspiel benutzt, diese sind:

Promotionschancen Ein Bonus, der die Nähe des Bauern zur gegnerischen Grundlinie bewertet. Denn je näher ein Bauer ist, desto besser ist die Chance für eine Promotion. Dies ist besonders im Endspiel wichtig, da es in Endspielpositionen oft zu Promotionen kommt.

Bauer Läufer Feldfarbe Ein Malus, wenn ein Bauer auf einem Feld steht, welches die gleiche Farbe wie der Läufer hat. Dieser Wert wird dann interessant, wenn der Gegner im Endspiel nur noch einen Läufer hat und somit der Bauer auf den entsprechenden Felder nicht geschlagen werden kann.

Springer Bauer Ein Malus, falls sich gegnerische Bauern auf der Linie a oder b befinden oder auf der Linie g oder h. Wenn der Springer beide Bauern bedrohen möchte, braucht er viele Züge um die Seite zu wechseln. Dieses Kriterium gilt nicht für den Läufer.

In vielen Schachprogrammen werden diese oder ähnliche Merkmale verwendet um eine Stellung zu bewerten. Dies geschieht aber immer statisch, das heißt, den Merkmalen wird ein bestimmter Wert zugeordnet und mittels einer Prozedur zu einem Wert für die Stellung verarbeitet. Diese Werte werden entweder durch die Programmierer vorgeben oder auch durch Methoden der CI gelernt (siehe [123, 121] und Kapitel 2.2.1).

Das Bewertungsmodul soll nun die Möglichkeit erhalten nicht nur diese Werte zu adaptieren, sondern auch die Kombination dieser Werte zu einer Bewertung zu komponieren, ein ähnlicher Versuch wurde in [22] durchgeführt (siehe auch Kapitel 5.1.1). Dort gab es eine feste Fitnessfunktion, so dass die Individuen nur eine festgelegte Funktion lernen konnten. Durch den Ansatz der impliziten Fitnessfunktion wird dem Modul die Möglichkeit gegeben die Bewertungsfunktion frei zu erzeugen. Die Fitnessfunktion ist implizit, da als Grundlage der Fitness nur der Sieg oder die Niederlage des Gesamtindividuums betrachtet wird.

Das Modul nutzt die Linear-Graph Struktur um eine Blattbewertung zu berechnen. Die Funktionen und Terminale, die verwendet wurden, sind in Abschnitt 8.3.1 genauer beschreiben. Wichtig an dem Funktionsset ist, dass es an den Standardverfahren zur Materialbewertung angeleht ist, aber auch Funktionen verwendet werden, die darüber hinausgehen. So werden hier wie auch in den anderen Modulen Funktionen verwendet, die dem Algorithmus die Möglichkeit geben potentielle Entwicklungen in die Bewertung mit einfließen zu lassen. Somit ermöglicht man sowohl dem Modul als auch dem Individuum etwas über den aktuellen Suchhorizont hinaus zu sehen um die Bewertung der aktuellen Position durchzuführen.

5.2.2.2 Zugbewertungsmodul

Die Aufgabe des *Zugbewertungsmoduls* ist es alle möglichen Züge einer Stellung zu bewerten und so eine relative Sortierung der Züge untereinander erreichen zu können. Zugsortierungen werden im Besonderen in Verfahren angewendet, die auf den Alpha-Beta Verfahren beruhen. Denn dies basiert auf einer Suchfenstertechnik, bei der stets die Werte der besten Alternativen des weißen und schwarzen Spielers im sogenannten Suchfenster (α, β) gespeichert werden. Je kleiner dieses Fenster ist, desto größer ist die Chance, dass Teile des Suchbaums abgeschnitten werden können. Eine gute Vorsortierung der Züge im Suchbaum ermöglicht es dem Alpha-Beta Verfahren die guten Alternativen früh zu finden und somit das Suchfenster schnell zu verkleinern. Die Sortierung der Züge erfolgt nach der Bewertung jedes Zugs durch einen Quicksortalgorithmus. Anschließend werden die Züge in der neuen Reihenfolge expandiert. Wenn das Modul eine gute Zugsortierung evolviert hat, ist es also möglich die Anzahl der Knoten, die bei der Suche expandiert werden, stark einzuschränken. So expandieren selbst die spielstärksten Schachprogramme $5 * \sqrt(n)$ Knoten, wobei n die Anzahl der Knoten für eine feste Suchtiefe angibt, siehe Tabelle 2.2 in Kapitel 2.2.5. Diese Werte liegen aber immer noch 20-30% über der Anzahl von Knoten den ein optimalen Suchbaum benötigen würde.

Gute Zugsortierungen berechnen im Allgemeinen eine gewichtete Summe über verschiedene Merkmale des Zuges, ähnlich wie es im qoopy-System durchgeführt wurde, siehe hierzu Abschnitt 5.1.2.3. Da die Zugsortierung ein sehr wichtiges Modul ist, um die Rechenkraft eines Algorithmus zu verbessern, haben wir uns dazu entschlossen nicht eine festgelegte gewichtete Summe über eine Kombination von Merkmalen des Zuges und der aktuellen Stellung zu berechnen, um die Parameter zu optimieren, sondern durch den evolutionären Prozess einen ganzen Algorithmus für diese Aufgabe zu evolvieren. Der Algorithmus hat dabei mehr Möglichkeiten einen Wert für einen Zug zu bestimmen, als dies durch eine gewichtete Summe möglich ist. Dies wird besonders durch die zur Verfügung gestellten Funktions- und Terminalmengen ermöglicht, sie werden im Abschnitt 8.3.2 genauer beschreiben. Sie basieren auf Techniken zur Zugbewertung, wie sie in Kapitel 2.2.3 vorgestellt wurden. Einige dieser Funktionen gehen jedoch darüber hinaus, eine Gruppe dieser Funktionen bewertet dabei verschiedene Aspekte der aktuellen Stellung und die andere Gruppe von Funktionen die gleichen Aspekte unter der Annahme, dass der Zug schon durchgeführt wurde. So ist es dem Zugbewertungsmodul möglich festzustellen, ob ein Zug kurzfristige positive oder negative Eigenschaften hat. Keine dieser Funktionen führt eine vollständige Materialbewertung der neuen Stellung durch, sondern betrachtet nur Teilaspekte davon oder Aspekte, die nie für eine Materialbewertung herangezogen werden. So kann die Differenz der angegriffenen/gedeckten Figuren durch einen Zug ermittelt werden oder es kann ermittelt werden, wie hoch der maximale Gewinn oder Verlust durch den Gegner nach diesem Zug ist. Dies soll es dem Modul ermöglichen die Güte eines Zuges besser zu bestimmen als durch eine gewichtet Summe von Merkmalen. Das Modul nutzt die Linear-Graph Struktur, um dem Algorithmus eine einfache Möglichkeit zu geben auf verschiedene Eingaben mit verschiedenen Berechnungen zu reagieren und somit eine größere Flexibilität zu gewährleisten, als dies mit eine linearen oder Baumstruktur möglich wäre.

Mit diesen Möglichkeiten muss das Modul nun einen Algorithmus entwickeln, der einen Gütewert für jeden Zug bestimmen kann. Die Randbedingungen sind dabei, dass gute Züge einen höheren Wert bekommen als schlechte Züge und die Bewertung muss unabhängig von der Seite mit dem Zugrecht berechnet werden. Das heißt, die Bewertung des Zuges muss unabhängig von der Farbe der Figur sein, die zieht. Die zweite Bedingung ist zwar einfach, jedoch kann durch die Wahl einer falschen Funktions- und Terminalmenge diese Nebenbedingung durch ein Individuum nur schwer oder gar nicht zu erreichen sein. Daher sind sie so gewählt, dass diese Nebenbedigung immer erfüllt ist. Somit hat die Evolution nur die Aufgabe ein Gütemaß zu bestimmen. Die Funktions- und Terminalmenge für dieses Modul besteht aus sehr vielen schachspezifischen Methoden, eine genaue Auflistung kann im Anhang, Abschnitt 8.3.2 gefunden werden.

5.2.2.3 Tiefenmodul

Das *Tiefenmodul* entscheidet für jede Position, ob diese expandiert werden soll oder nicht. Das Tiefenmodul unterscheidet sich in der Aufgabenstellung stark von dem Tiefenmodul des qoopy-Systems, wie es in Kapitel 5.1.2.4 beschrieben wird. Im qoopy-System wird durch das Tiefenmodul die Entscheidung nicht direkt getroffen. Im GenticChess hat das Modul neben der direkten Entscheidung noch eine weitere Aufgabe, es bestimmt die Funktion, welche die Züge erzeugt, die im nächsten Schritt expandiert werden sollen. Dem Modul stehen vier verschiedene Funktionen zur Verfügung, um die Struktur des Suchbaumes zu beeinflussen. Die Funktionen sind:

- Alle Züge: Es werden alle erlaubten Züge erzeugt.
- Angriffszüge: Es werden nur Züge erzeugt, die eine Figur schlagen oder angreifen.
- Verteidigungszüge: Es werden nur Züge erzeugt, die die Deckung der eigenen Figuren erhöhen.
- Nullzüge: Es wird kein Zug durchgeführt, sondern das Zugrecht wird an den Gegner abgegeben, siehe 2.2.6.6.

Um dem Modul diese Möglichkeit zu geben, wurde das Tiefenmodul der State-Space Struktur entsprechend implementiert, so dass die Terminalknoten der Linear-Graph Struktur eine dieser Funktionen enthalten. Diese Elemente der Struktur werden Programmflussabschnitt genannt, weiteres findet man dazu in Kapitel 4.3.1.

Der Programmflussabschnitt besteht aus zwei Funktionen, die miteinander kombiniert werden, einerseits hat man eine binäre Entscheidungsfunktion, die Ergebnisse des Tiefenentscheidungsmoduls für die Entscheidung nutzen, ob die Problembewertung oder die Blattbewertung aufgerufen wird. Zusätzlich enthält der Abschnitt die Funktion, die die möglichen Züge im Suchbaum für den nächsten Schritt erzeugen. Durch den Programmflussabschnitt ermöglicht man den Individuen einen sehr variablen Aufbau des Suchbaums und somit die Möglichkeit andere Wege im Suchraum zu finden.

Das Tiefenmodul hat somit auch die Möglichkeit einen variablen Suchhorizont zu erzeugen. Dies wirkt dem Horizonteffekt entgegen, da nun die Suche nicht nach einer festen Suchtiefe beendet wird, sondern situationsabhängig auf Grund der gegebenen Stellung. Gleichzeitig geben wir dem Tiefenmodul die Möglichkeit heuristisch bei der Wahl der nächsten Züge vorzugehen, dies bewirkt eine heuristische Bewertung des gewählten Zuges an der Wurzel, jedoch ist die Bewertung jedes Algorithmus im Schach heuristisch, da die Bewertung nie bis zum Ende der Partie durchgespielt werden kann. Durch dieses Vorgehen kann jedoch der Ausgrad des Baumes um viele Ebenen reduziert werden, was zu einer genaueren Analyse der beobachteten Züge führt. Die Aufgabe des Moduls ist es somit auch ein gutes Verhältnis zwischen der Auswahl der Züge und der Tiefe der betrachteten Variante zu finden. Im optimalen Fall soll das Modul einen ungleichmäßigen Suchbaum erzeugen, der jedoch alle wichtigen Varianten enthält um eine gute Entscheidung treffen zu können.

Da das Tiefenmodul einen entscheidenden Einfluß auf die Rechenzeit des Individuums hat, ist die Suchtiefe nicht unbeschränkt, so wird die Expansion des Baumes maximal bis zur Tiefe zehn zugelassen. Diese sind ungefähr 10^{14} Knoten im Suchbaum, dies ist für einen evolutionären Prozess zu langsam, denn bis die Fitness eines Individuums feststeht, müssen mehrere Spiele durchgeführt werden, hierbei würde die Auswertung eines Spiels aber Tage benötigen. Daher wurde noch eine zweite Beschränkung eingefügt, die nun die Anzahl der Knoten im Suchbaum beschränkt. Ein Individuum darf eine maximale Anzahl Knoten pro Zug verwenden, dadurch kann die vollständige Fitnessauswertung eines Individuums innerhalb einiger Stunden durchgeführt werden. Der Wert für die maximale Anzahl der Knoten wird als Parameter dem System übergeben, dieser Wert variierte zwischen 10.000 und 250.000 Knoten. Wenn das Knotenlimit vom Individuum erreicht wird, wird das Tiefenmodul nicht mehr ausgeführt, sondern die Rekursion wird für alle noch folgenden Züge beendet. Da nun aber noch viele Bewertungen nicht durchgeführt sein können, da das Module immer sehr tief gesucht hat, erlauben wir dem Individuum, für alle ausstehenden Stellungen noch die Materialbewertung durchzuführen. Dadurch können von einem Indviduum etwas mehr Knoten ausgewertet werden als durch das Knotenlimit vorgesehen. Wir haben uns für diese Vorgehen aber entschieden, da wir hoffen gute Zugsortierungsmodule und Materailbewertungmodule nicht durch ein schlechtes Tiefenmodule zu beginn der Evolution zuverlieren.

Die Aufgabe des Tiefenmoduls ist es, anhand der gegebenen Stellung zu entscheiden, ob der mögliche Teilbaum genauer untersucht werden soll oder nicht. Die Funktions- und Terminalmenge für diese Module besteht aus sehr vielen schachspezifischen Methoden, eine genaue Auflistung kann im Anhang Abschnitt 8.3.3 gefunden werden.

5.2.3 Fitness Funktion

Die Fitness eines Schachindividuums wird dadurch gemessen, dass es Spiele gegen Schachprogramme fester Suchtiefe durchführen muss. Wenn das Individuum gewinnt, spielt es gegen ein Gegnerprogramm mit einer Suchtiefe, die um eins größer ist. Die Fitnessberechnung hat zwei Abbruchkriterien, das erste Abbruchkriterium ist die Anzahl der verlorenen Spiele. Während der Evolution wird die Fitnessberechnung nach einem verlorenen Spielen beendet, diese Grenze kann durch einen Parameter bestimmt werden. Ein verlorenes Spiel ist zwar sehr restriktiv, so kann aber die Fitnessbewertung für ein Individuum sehr schnell beendet werden. Es hat sich gezeigt, dass es, wenn man die Grenze der verlorenen Spiele erhöht, zu keinem Gewinn in der Evolution für die verwendeten Parameter kommt. Die Anzahl der verlorenen Spiele ist auch ein Parameter für den Selektionsdruck, somit haben wir hier einen sehr hohen Selektionsdruck gewählt, der Grund hierfür ist der hohe Rechenaufwand, wenn ein geringerer Selektionsdruck verwendet wird. Sehr viele Individuen, die mehr als zwei Spiele verlieren, verlieren auch Spiele gegen Gegnerprogramme mit einer größeren Rechentiefe, wodurch sehr viel Rechenzeit aufgewendet wird. Die Tests haben für die verwendete Evolutionszeit zu keinen Vorteilen geführt. Denn viele Individuen, die gegen leichte Gegnerprogramme verlieren, verlieren auch gegen stärkere Programme und die Individuen die auch gegen Gegnerprogramme mit größeren Rechentiefen gewinnen, sind auf der anderen Seite zu unbeständig und dies ist nicht das Ziel der Evolution. Das System soll Individuen generieren, die gute Ergebnisse mit kleinen Spielbäumen erreichen, daher ist es nicht sinnvoll Individuen zu erhalten, die unbeständig sind oder zuviel Rechenzeit benötigen.

Ein Spiel dauert maximal 50 Halbzüge, danach wird es abgebrochen, falls nicht ein Programm vorher gewonnen hat. Als Ergebnis eines Spiels wird dann der reine Materialwert des Spiels ohne Bewertung von Stellungselementen genutzt. Der Wert bezieht sich immer auf das Individuum und ist auf den maximalen Materialwert ohne die Könige normiert. Das heißt, ein Ergebnis von eins für ein Individuum bedeutet, dass es keine Figuren verloren hat und das Gegnerprogramm nur noch den König besitzt. Ein Bauer hat somit einen Einfluss von $\frac{100}{4060} \approx 0,0246$ auf den Ergebniswert. Ein Spiel wird nun als gewonnen betrachtet, wenn das Individuum mindestens einen Wert von $0,1/Tiefe$ hat. $Tiefe$ ist immer die Tiefe und somit die Spielstärke des Gegnerprogramms. Für ein Individuum wird es immer schwieriger zu gewinnen und daher wird die Schwelle für den Gewinn immer weiter gesenkt. Ab Gegnerprogramme der Tiefe fünf wird die Schwelle auf einen Wert von $0,02$ gesenkt, was ungefähr einem Bauern entspricht, d.h. ein Individuum muss während eines Spiels mindestens einen Materialvorteil von einem Bauern erspielen, um weiter an der Fitnessbewertung teilzunehmen.

Die Fitness selbst setzt sich aus vier Werten zusammen, zwei dieser Werte bewerten die Ergebnisse der Individuen gegen die verschiedenen Gegnerprogramme. Diese Werte werden durch die Funktionen 5.13 und 5.14 berechnet. Die Funktionen summieren die Bewertung der Spiele für die jeweilige Farbe, dabei wird eine Gewichtung durchgeführt die auch davon abhängig ist, ob ein Spiel gewonnen oder verloren wurde. Gewonnene Spiele werden stärker gewichtet als verlorene Spiele und Spiele gegen Gegnerprogramme mit größerer Suchtiefe werden auch stärker gewichtet. Die Gewichtung erfolgt über die Gauss'sche Reihe[5], deren Eingabeparameter die Spielstärke des Gegnerprogramms ist.

[5]Im Jahr 1786 entwickelte Karl Friedrich Gauß eine Funktion, wie die Summe der natürlichen Zahlen

Durch dieses Vorgehen erreicht man, dass Spiele gegen spielstärkere Gegnerprogramme stärker bewertet werden und somit der Verlust eines Spieles gegen einen schwächeren Gegner nicht so ins Gewicht fällt. In der Fitnessfunktion wird der Ergebniswert als positiv bewertet, wenn er größer Null ist und nicht mit einem Schwellwert wie beim Abbruchkriterium. Das Ergebnis des Schwellwertes geht aber in die Werte $BothWin$ und $LastWinDepth$ ein. $BothWin$ ist die Tiefe, in der das Individuum sowohl das schwarze als auch das weiße Gegnerprogramm besiegt hat, und $LastWinDepth$ ist die Tiefe, in der ein Individuum das letzte mal gegen ein Gegnerprogramm gewonnen hat.

$$White(i) = \sum_{d=0}^{MaxDepth} \begin{cases} gauss(d+1) * resWhite(d) > 0 & resWhite(d) > 0 \\ gauss(d) * resWhite(d) & resWhite(d) < 0 \end{cases} \tag{5.13}$$

$$Black(i) = \sum_{d=0}^{MaxDepth} \begin{cases} gauss(d+1) * resBlack(d) > 0 & resBlack(d) > 0 \\ gauss(d) * resBlack(d) & resBlack(d) < 0 \end{cases} \tag{5.14}$$

Der Wert $LastWinDepth$ wird mit 1.000 multipliziert und der Wert $BothWin$ mit 100. Die Fitnessfunktion hat somit drei Ebenen, auf denen die Fitness steigen kann. Die Ebene mit den geringsten Einfluss ist der gewichtete Materialgewinn der Individuen, die zweite und dritte Ebene ist jeweils eine Treppenfunktion und wird über die Werte von $BothWin$ und $LastWinDepth$ gesteuert. Da diese Werte den höchsten Anstieg in der Fitness bewirken, werden somit Individuen bevorzugt die gegen starke Gegnerprogramme gewinnen können und nicht Individuen, die mit einem möglichst hohen Materialvorteil gewinnen. Aus diesem Grund wird auch der Wert $LastWinDepth$ stärker gewichtet, denn dadurch sollen Programme bevorzugt werden die gegen spielstarke Gegner gewinnen können. Diese Werte wurden in den ersten Fitnessfunktionen nicht berücksichtigt und es führte dazu, dass viele Evolutionen an den Gegnerprogrammen ab der Tiefe drei scheiterten und nur den Materialgewinn gegen die leichte Gegnerprogramme erhöhten.

$$\begin{aligned} BothWin &= BothWin * 100; \\ dFit &= ((BothWin) + dWhiteVal + dBlackVal + (LastWinDepth * 1.000, 0); \end{aligned} \tag{5.15}$$

Durch die Verbindung des gewichteten Materialgewinns und der Stufenfunktion in der Fitnessfunktion 5.15 ermöglicht man es der Evolution, Programme zu erzeugen, die gegen spielstarke Programme gewinnen können aber nicht den maximalen Materialgewinn erzielen müssen um gegen den Gegner der nächsten Stufe zu spielen.

Die Rechenzeit ist ein weiterer wichtiger Punkt bei der Auswahl der Individuen, so wird, neben der Fitness auch die Anzahl der betrachteten Stellungen, bei der Selektion der Individuen mit berücksichtigt. Ein Ziel der Evolution ist es Programme zu entwickeln, die möglichst kleine Suchbäume erzeugen, daher wird beim Vergleich der Fitness zweier Individuen nicht nur der Fitnesswert betrachtet, sondern auch die durchschnittliche Anzahl der betrachteten Stellungen pro Zug. Dieser Wert wird aber nur dann berücksichtigt, wenn die Fitnesswerte, die durch die Funktion 5.15 berechnet wurden gleich sind. Dadurch erreicht man in den Phase, in denen es zu keine Verbesserung der Fitness kommt, eine Verkleinerung des Suchraums.

von 1 bis n einfach berechnet werden kann.

Zu dieser Art der Fitnessberechnung gibt es für Schach nur eine Alternativen. Ein Individuum kann mit Hilfe von Schachstellungen trainiert werden. Diese Art der Fitnessberechnung wurde für einige Module des *ChessGP-Systems* verwendet (siehe 5.1.1), diese Methode hat aber den Nachteil, dass nicht alle Spielsituationen für ein Programm bereitstellt. So gab es bei der Erstellung der Fitnesscases für das ChessGP-System das Problem, dass man kaum Beispiele für Endspiel Stellungen hatte und diese erst erzeugen musste. Bei der Analyse der Ergebnisse haben wir auch gesehen, dass es sehr oft zu nicht erwünschten Verhalten der Individuen gekommen ist, da es ihnen nicht gelungen ist mit Hilfe der gegebenen Fitnesscases die Problemstellung zu abstrahieren (siehe hierzu Abschnitt 6.1.1). Wegen dieser Probleme haben wir in allen weiteren Evolutionen immer die zeitaufwändigere Variante genutzt, die vollständige Spiele zur Berechnung der Fitness nutzt.

5.2.4 Evolution

Das GeneticChess System nutzt ein Standardverfahren zur Evolution der Schachindividuen. Es wird ein Pool mit eine zweier Turnierselektion verwendet, die auf einem Rechner läuft. Es wird keine Internet-Evolution wie in qoopy verwendet, da hier mehrere Läufe mit verschiedenen Parametern durchgeführt werden sollten und dies bei dem internetbasierten qoopy System schwierig ist. Da die Geschwindigkeit des GeneticChess Systems gegenüber qoopy auch um einen Faktor von circa 50 besser ist, war auch eine Parallelisierung für die durchgeführten Läufe nicht notwendig. Es hat sich zwar später gezeigt, dass eine Parallelisierung für die Evolution der stärksten Individuen vorteilhaft gewesen wäre, jedoch war auch ohne Parallelisierung der Fitnessbewertung möglich, sehr gute Ergebnisse zu erzielen.

Da die Evolution von Schachindividuen sehr rechenzeitaufwendig ist, werden für die Läufe maximale Fitnesswerte angegeben, die erzielt werden sollen. Wenn die Zielfitness durch die Evolution erreicht wird, wird die Evolution abgebrochen. Dies verhindert zwar eine weitere Verbesserung der Individuen, wenn das Abbruchkriterium, die maximale Anzahl der Generationen, noch nicht erreicht wurde, jedoch wäre dies bei den gewählten Einstellungen für die Individuen auch sehr unwahrscheinlich. Dies ermöglicht die Einsparung erheblicher Rechenzeiten, da die Fitnessberechung für das Individuum auch sehr aufwendig ist, wenn die Zielfitness erreicht wurde. Die Erfahrung mit evolutionären Verfahren zeigt auch, dass nach dem Auftreten eines guten Individuum sehr viele Nachfahren entstehen, die ähnlich gut sind und somit in unserem Fall die Zeit der Evolution erheblich erhöhen würden.

5.2.5 Das ChessGP Gegnerprogramm

Das *Gegnerprogramm* zur Fitnessberechnung entspricht mit leichten Modifikationen dem Gegnerprogramm, wie es im qoopy-System verwendet wird, siehe hierzu Abschnitt 5.1.2.7. Eine Änderung zum Algorithmus im qoopy-System ist das Weglassen des Zufallswerts bei der Materialbewertung. Dieser bewirkte, dass zu einer Stellung ein zufälliger Wert in einem festgelegten Intervall von 0 bis 20 zum Materialwert der Stellung hinzugezählt oder abgezogen werden kann. Damit wurde beabsichtigt, das Spiel variabler zu gestalten, so dass die Gegnerprogramme nicht immer gleich agieren. Da qoopy auf einer internetbasierten Evolution basierte, war ein variableres Spiel bei der Auswertung wünschenswert. Dies führte jedoch dazu, dass der Algorithmus besser, aber auch schlechter arbeiten kann und somit die Bewertung eines Individuums auch vom Zufall abhing. Dies wurde durch die Fitnessbewertung gemildert, die ein Individuum mehrmals gegen den Algorithmus mit gleicher Stärke spielen ließ. In den Nachbewertungen hatte sich jedoch gezeigt, dass dies nicht ausreichend war. Durch den Verzicht auf diesen Wert soll die Bewertung der Individuuen einheitlicher sein, dies kann jedoch nicht ganz erreicht werden, da die Bewertung als solche dynamisch ist und auch stark vom Verhalten des Individuums abhängt.

Die weiteren Veränderungen betreffen nur Implementationsdetails, die keine Auswirkungen auf das Verhalten des Algorithmus haben. Die stärkste Änderung ergibt sich durch die Verwendung von C++ als Programmiersprache. Das qoopy-System ist in Java implementiert, was für die Zielsetzung einer Internetevolution von Vorteil war, jedoch auch einen stärkeren Verlust bei der Rechengeschwindigkeit bewirkte. Dies lag auch an der Bitboard Programmierung, die in Java nicht so effizient zu programmieren ist wie in C++. Jedoch wurden das Verhalten und der Aufbau des Gegnerprogramms nicht verändert, da wir mit diesem Programm im qoopy System gute Erfahrungen gemacht haben.

Kapitel 6

Ergebnisse und Analyse

Im folgenden Kapitel werden nun die Ergebnisse und Analysen der drei GP-Systeme vorgestellt. Wir folgen hierbei der Reihenfolge, wie sie durch das Kapitel 5 vorgegeben wird. Da zwischen der Entwicklung und Analyse der einzelnen Systeme mehrere Jahre liegen und die Systeme zusätzlich sehr unterschiedlich sind, wurden hier verschiedene Schwerpunkte bei der Analyse gewählt oder durch die Systeme ermöglicht.

6.1 Ergebnisse des ChessGP Systems

Das ChessGP-System war das erste System, das versucht hat schachspielende GP-Individuen zu evolvieren. Da die Aufgabe komplex ist, haben wir die Evolution des Schachindividuums in verschiedene einzelne Evolutionsvorgänge aufgeteilt. Diese Evolutionsprozesse spiegeln die drei Bereiche wieder, die der Aufteilung des Individuums in die drei Ebenen entspricht. Im folgenden werden nun die Ergebnisse der einzelnen Ebenen vorgestellt und mit der letzten Ebene auch das Ziel der Evolution, die schachspielenden Individuen, vorgestellt.

6.1.1 Materialbewertungsebene

Die Individuen für die Materialbewertung wurden durch lineare Strukturen gebildet. Die Funktionen, die von den Individuen zur Lösung der Aufgabe genutzt wurden, sind im Abschnitt 8.1.1 aufgeführt und näher erläutert, die Parameter, die in allen Tests nicht verändert wurden, sind in der Tabelle 6.1 zusammengefasst.

Die ersten Evolutionsläufe der Materialbewertungsebene wurden mit einer Populationsgröße von 100 Individuen durchgeführt. Jedoch traten über den gesamten Evolutionszeitraum kaum Verbesserungen auf, wie man dies im direkten Vergleich der Plots 6.1 und 6.2 für die Clevere Materialbewertung sehr gut sehen kann. Dieser Vergleich ist beispielhaft für das Verhalten aller Evolutionsläufe mit 100 Individuen für die verschiedenen Aufgaben. Daher wurden alle Läufe mit 1.000 Individuen durchgeführt. Auf den folgenden Kurven ist jeweils die Fitness des besten Individuums abgebildet. Hierbei ist zu beachten, dass die Fitnesswerte der verschiedenen Teilprobleme der Materialebene nicht normiert sind und immer die Quadratsumme der Fehlerabweichungen über alle Fitnesscases berechnet wird. Eine Ausnahme bilden hier die Bewertungen *Angriff* und *Verteidigung*, hier wird die Wurzel der Quadratsumme der Fehlerabweichungen dargestellt. Dies war sinnvoll, da sonst die Entwicklung der Fitness in der Endphase der Evolution schlecht dargestellt worden wäre. Die genaue Beschreibung der Fitnesswerte kann in Kapitel 5.1.1.3 genauer nachgelesen werden.

Parameter	Wert	Beschreibung
Evaluationen	1.000.000	Anzahl der Fitnessauswertungen pro Lauf
Population	1.000	Größe der Population für einen Lauf
Crossover	0,8	Wahrscheinlichkeit, dass zwei Individuen an einem Crossover teilnehmen
Mutation	0,3	Wahrscheinlichkeit, dass ein Individuum mutiert wird
Selektion	Turnierselektion 2*2	Turnierselektion aus zwei Zweier-Turnieren
Initialisierung	ramped half and half	Initialisierungsart der Population
Minimale Individuengröße	10	Anzahl der Knoten, die ein Individuum minimal haben durfte
Maximale Individuengröße	50	Anzahl der Knoten, die ein Individuum maximal haben durfte
Konstanten	[-1000,..,1000]	Wertebereich der Konstanten

Tabelle 6.1: Diese Tabelle enthält alle Parameter, die für alle Tests der Materialbewertung genutzt wurden.

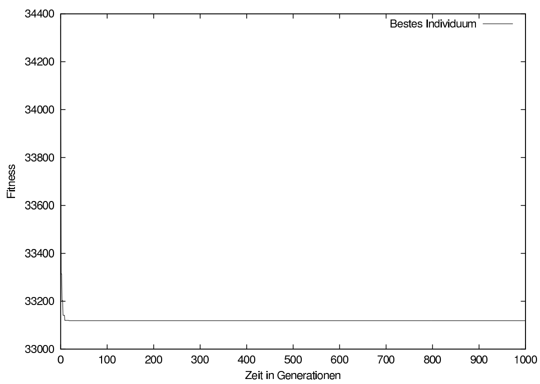

Abbildung 6.1: Materialbewertung Clever mit einer Population von 100 Individuen.

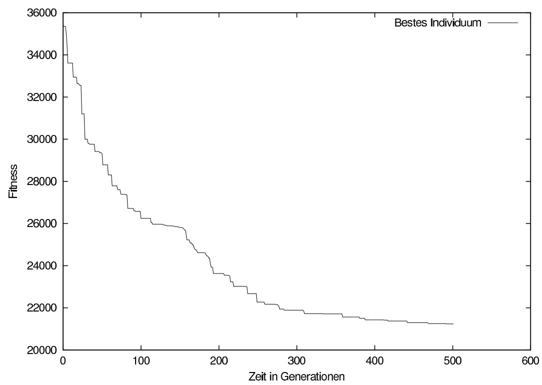

Abbildung 6.2: Materialbewertung Clever mit einer Population von 1.000 Individuen.

Da die Fitnesscases aus 40.000 transformierten Schachstellungen bestanden, die aus 65.000 Schachstellungen ausgewählt wurden, konnten nicht alle Fitnesscases zur Bestimmung der Güte eines Individuums herangezogen werden. Daher wurde eine *Random subset selection* durchgeführt, dabei wird in jeder Generation eine Teilmenge aus alle Fitnesscases ausgewählt und diese Teilmenge auf allen Individuen angewendet [38]. Wir haben auch das *Stochastic sampling* getestet, indem für jedes Individuum eine neue Teilmenge ausgewählt wurde [93]. Dies führte aber in unserem Fall zu sehr schlechten Ergebnissen, so dass wir uns für das *Random subset selection*-Verfahren entschie-

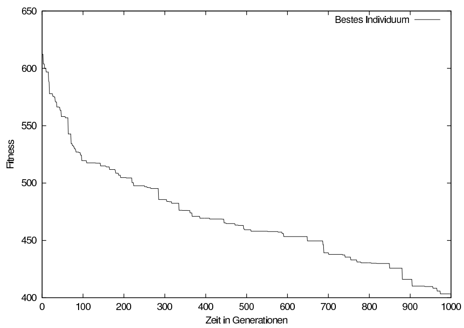

Abbildung 6.3: Bewertung Verteidigung mit einer Population von 1.000 Individuen

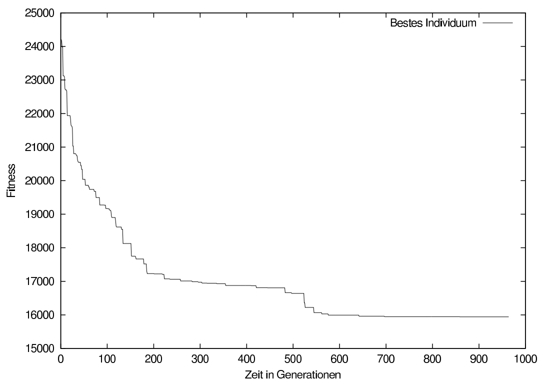

Abbildung 6.4: Bewertung Material mit einer Population von 1000 Individuen.

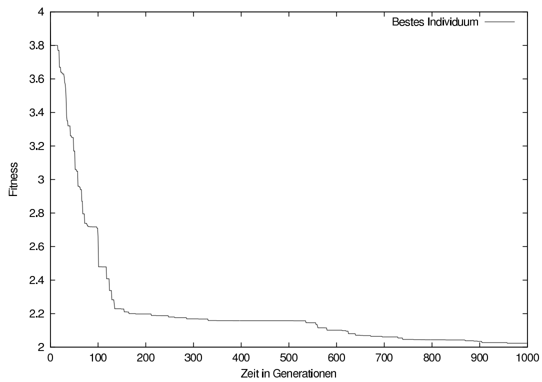

Abbildung 6.5: Materialbewertung Aggressive mit einer Population von 1000 Individuen.

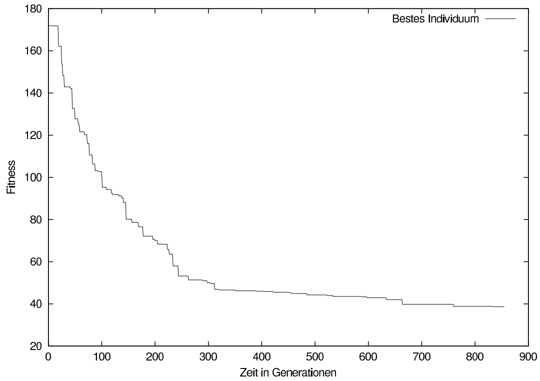

Abbildung 6.6: Bewertung Defensive mit einer Population von 1000 Individuen

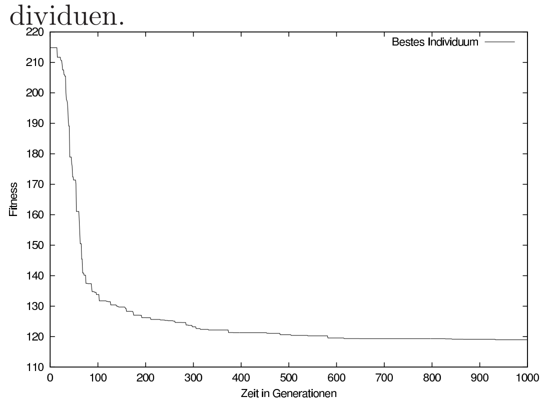

Abbildung 6.7: Bewertung Angriff mit einer Population von 1000 Individuen

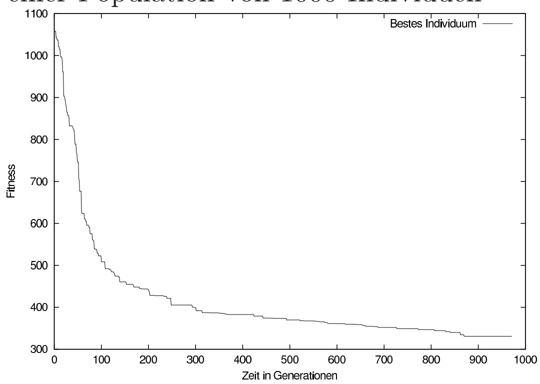

Abbildung 6.8: Bewertung Endspiel mit einer Population von 1000 Individuen

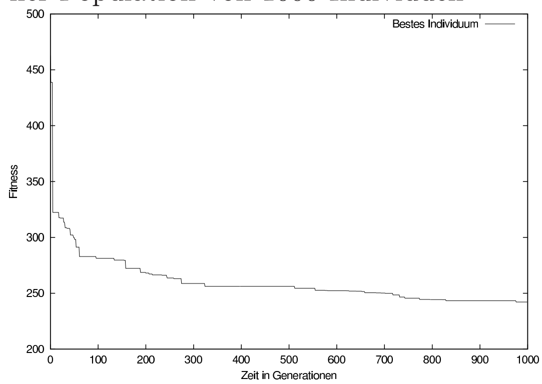

Abbildung 6.9: Bewertung Komplex mit einer Population von 1000 Individuen

den haben. Nach verschiedenen Tests haben wir die Größe der Teilmenge auf 100 Fitnesscases festgelegt. Dies war ein Kompromiss zwischen der Güte der Individuen und dem Rechenaufwand während der Evolution. Für eine kurze Zusammenfassung weiterer Verfahren siehe auch [75]. Die restlichen 25.000 Schachstellungen, die nicht als Fitnesscases verwendet wurden, wurden anschließend zur Analyse der Individuen genutzt, hierzu später mehr.

Die Plots 6.4, 6.5, 6.6, 6.7, 6.8 und 6.9 zeigen ein typisches Verhalten für die Entwicklung der Fitness während der Evolution. Die Verbesserung der Fitness ist in den ersten

Generationen dramatisch und mündet dann in eine Phase mit nur noch kleinen Fitnessverbesserungen. Aufgrund der hohen Laufzeit für eine Evolution wurden auch einige Evolutionsläufe vorzeitig beendet, wenn keine signifikanten Verbesserungen mehr zu erwarten waren. Eine Ausnahme ist die Fitnessentwicklung der Verteidigungsbewertung in Plot 6.3, hier scheint die Verbesserung über den gesamten Zeitraum verteilt zu sein. Dies deutet darauf hin, dass noch weitere Verbesserungen durch eine längere Evolution möglich gewesen wären. Die Evolution wurde jedoch trotzdem abgebrochen, da erstens alle Individuen einer Ebene des Gesamtindividuums die gleichen Parameter haben sollten und zweitens wurde die Evolution der Teilindividuen während der Evolution des Gesamtindividuums wieder aufgenommen. Dadurch sollte den Teilindividuuen eine Anpassung an das Gesamtindividuum ermöglicht werden. Aus diesen Gründen wäre es vorteilhaft gewesen, wenn auch die Evolutionsläufe der anderen Materialbewertungen ebenfalls ein größeres Entwicklungspotenzial gezeigt hätten. Alle Evolutionsläufe wurden mit dem vollständigen Funktionssatz für diese Ebene durchgeführt, alle Funktionen sind im Kapitel 8.1.1 beschrieben.

Da wir die Ergebnisse nicht befriedigend fanden, wurden anschließend Evolutionsläufe mit eingeschränkten Funktionssätzen durchgeführt, die auf die einzelnen Problemstellungen zugeschnitten waren. Hierzu wurde eine Menge von Funktionen verwendet, die allen Bewertungen zu Verfügung gestellt wurden und eine Teilmenge, die auf die jeweilige Aufgabe zugeschnitten war. Allen Bewertungsfunktionen gemeinsam waren die Operationen: *add, addEqual, sub, subEqual, mult, multEqual, div, divEqual, absolute* und *ifThen*. Die eingeschränkten Funktionssätze für die einzelnen Bewertungen sahen wie folgt aus.

- Material Operationen: boardLoop, whichPiece
  Terminale: pieceValue
- Clever Operationen: boardLoop, whichPiece, setRow, setLine
  Terminale: pieceValue, legalMovesNumber
- Angriff Operationen: boardLoop, rowLoop, whichPiece, setRow, swap
  Terminale: pieceValue, attackedByNumber, attackedByValue
- Verteidigung Operationen: boardLoop, rowLoop, whichPiece, setRow, swap
  Terminale: pieceValue, attackedByNumber, attackedByValue
- Aggressive Operationen: boardLoop, rowLoop, whichPiece, setRow
  Terminale: pieceValue, attackedByNumber, attackedByValue
- Defensive Operationen: boardLoop, rowLoop, whichPiece, setRow
  Terminale: pieceValue, attackedByNumber, attackedByValue
- Komplex Operationen: boardLoop, whichPiece
  Terminale: pieceValue, legalMovesNumber, attackedByNumber
- Endspiel Operationen: boardLoop, whichPiece
  Terminale: pieceValue

Durch eine Einschränkung des Funktionssatzes sollte es der Evolution ermöglicht werden, einfacher Lösungen zu finden, da nun nur Operationen zur Verfügung stehen, die eine Lösung des Problems ermöglichen. Die Evolution der Materialbewertung erfüllte die Aufgabe, so konnte hier ein Individuum erzeugt werden, welches die optimale Lösung bezogen auf die Fitnessfunktion liefert. Die Bewertung machte sowohl auf der Trainingsals auch auf der Testmenge keine Fehler. Die Evolution der Aggressiven und Komplexen Materialbewertung erreichten zwar nicht die optimale Lösung, aber Verbesserungen gegenüber den Läufen mit den nicht eingeschränkten Funktionsmengen, siehe hierzu die Plots 6.5 und 6.9.

Die Plots 6.6, 6.7 und 6.8 zeigen die Ergebnisse für die anderen Materialbewertungen, hier wurde keine Verbesserung der Ergebnisse erkennbar. Zu beachten ist hier aber auch die kurze Laufzeit, die einerseits auf externe Rechnerstörungen zurückzuführen war, aber auch durch Abbruch der Läufe zustande kam. Aufgrund der benötigten Rechenleistung und den geringen Erfolgsaussichten, die auch durch Vortests untermauert wurden, haben wir hier darauf verzichtet, die Läufe bis zur letzten Generation durchzuführen.

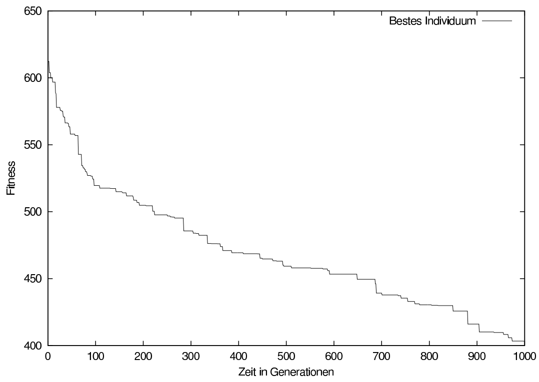

Abbildung 6.10: Materialbewertung Verteidigung mit einer Population von 1000 Individuen

Zum Abschluss der Evolution wurde das Verhalten der Individuen auf sämtlichen Schachstellungen in der Datenbank untersucht. Zur Visualisierung wurden der Fehler des Individuums bezüglich des Wertes in der Datenbank auf der Y-Achse eines Plots angegeben. Dabei ergaben sich vier verschiedene Verhaltensweisen der Individuen auf allen Datensätzen. Das Individuum der einfachen Materialbewertung erzeugte ein optimales Verhalten, indem es für jede Schachstellung den Fehlerwert 0 erzeugte. Das Individuum mit der Aufgabe einer Aggressiven Materialbewertung, Plot 6.20, zeigt auf allen Schachstellungen ähnlich gute Ergebnisse. Obwohl dieses Individuum extreme Ausreißer gegenüber dem erwarteten Wert hat, zeigt sich aber, dass das Individuum abstrahieren und ein allgemeines Konzept für die Aufgabe entwickeln konnte. Das Individuum der Cleveren Materialbewertung, Abbildung 6.18, zeigt dagegen ein Verhalten, welches man zwar erwartet, jedoch eigentlich nicht erhalten möchte. Denn hier erzeugt das Individuum auf den unbekannten Schachstellungen (ab Stellung 40000) einen höheren Fehler, als auf den Stellungen, die als Fitnesscases genutzt wurden. Dieses Verhalten ist zwar nicht wünschenswert, aber ein schlechteres Ergebnis auf einer Validierungsmenge ist für evolutionäre Verfahren normal. Ein ähnliches Verhalten ergab sich auch bei der Defensiven Materialbewertung, Plot 6.19, nur mit dem Unterschied, dass der Fehler auf den unbekannten Datensätzen kleiner war, diese Verhalten wurden auch von der Komplexen Materialbewertung gezeigt. Die Materialbewertungen Angriff, Endspiel und Verteidigung zeigten das extremste Verhalten auf den unbekannten Datensätzen. Die Abbildung 6.17 zeigt das Verhalten der Materialbewertung Verteidigung auf allen Datensätzen, hier ist deutlich die Verschlechterung auf den unbekannten Daten zu erkennen. Der Evolution ist es für diese Bewertungen nicht gelungen ein generelles Modell für die Aufgabenstellung zu entwickeln.

6.1.2 Planungsebene

Die Individuen der Planungsebene sollen für eine gegebene Stellung eine Zugliste generieren. Die Aufgabe bestand einerseits darin eine Zugliste zu generieren und andererseits darin eine möglichst kleine Zugliste zu erzeugen. Die Funktionen, die von den Individuen zur Lösung der Aufgabe genutzt wurden, sind im Abschnitt 8.1.2 beschrieben. Bei der Evolution hat sich jedoch gezeigt, dass die gestellte Aufgabe für die Individuen zu schwierig war. Obwohl der Verlauf der Fitnesskurven dem anderer Evolutionsläufe entsprach, zeigte sich bei der Analyse der Individuen, dass alle Individuen extrem kurze Zuglisten berechneten. Wie im Kapitel 5.1.1.4 beschrieben ist, wurde die Länge der Zugliste mit einem Strafterm belegt, siehe hierzu die Gleichungen 5.11 und 5.12. Obwohl der Strafterm sowohl für zu kurze, als auch zu lange Listen gilt, war es für die Individuen besser oder einfacher kurze Listen zu erzeugen. Die besten Individuen der verschiedenen Läufe

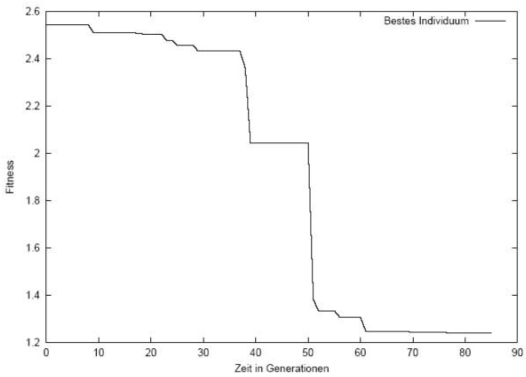

Abbildung 6.11: Materialbewertung Aggressive mit einer Population von 1000 Individuen und eingeschränkter Funktionsmenge

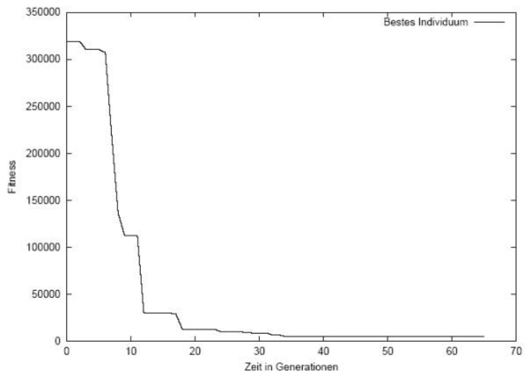

Abbildung 6.12: Materialbewertung Clever mit einer Population von 1000 Individuen und eingeschränkter Funktionsmenge

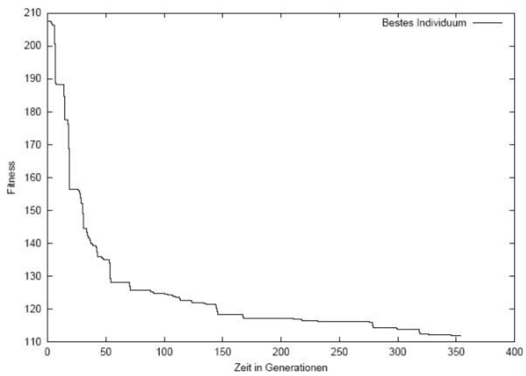

Abbildung 6.13: Materialbewertung Defensive mit einer Population von 1000 Individuen und eingeschränkter Funktionsmenge

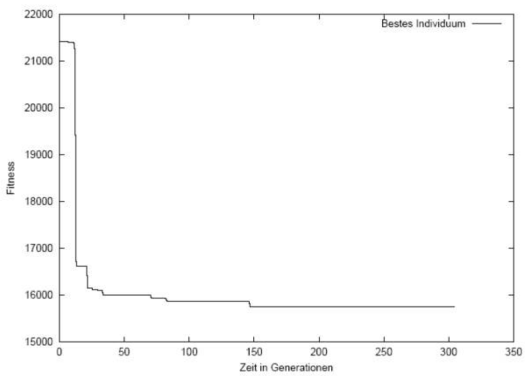

Abbildung 6.14: Materialbewertung Angriff mit einer Population von 1000 Individuen und eingeschränkter Funktionsmenge

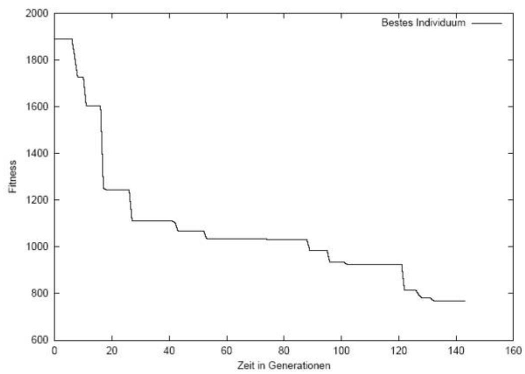

Abbildung 6.15: Materialbewertung Endspiel mit einer Population von 1000 Individuen und eingeschränkter Funktionsmenge

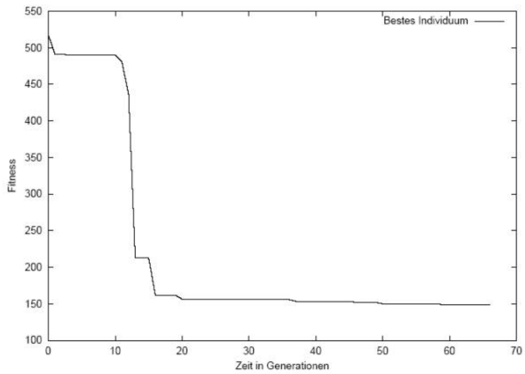

Abbildung 6.16: Materialbewertung Komplex mit einer Population von 1000 Individuen und eingeschränkter Funktionsmenge

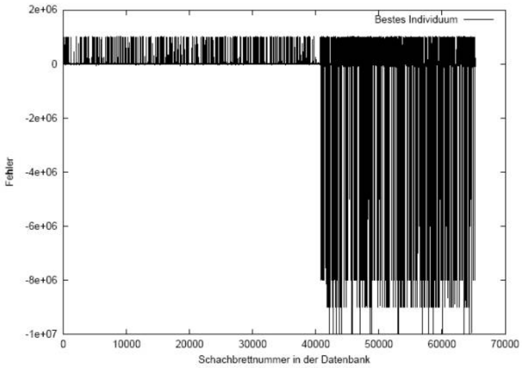

Abbildung 6.17: Fehler des besten Individuums der Materialbewertung Verteidigung auf allen Schachstellungen der Datenbank. Der durchschnittliche Fehler beträgt 213685.

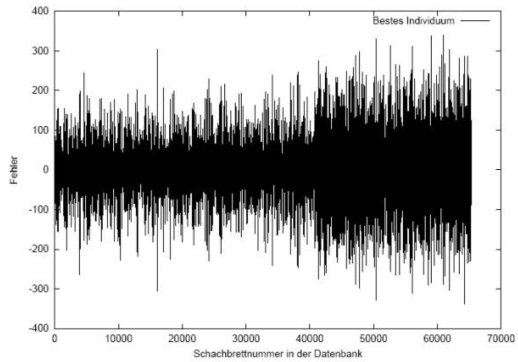

Abbildung 6.18: Fehler des besten Individuums der Materialbewertung Clever auf allen Schachstellungen der Datenbank. Der durchschnittliche Fehler beträgt 50,8.

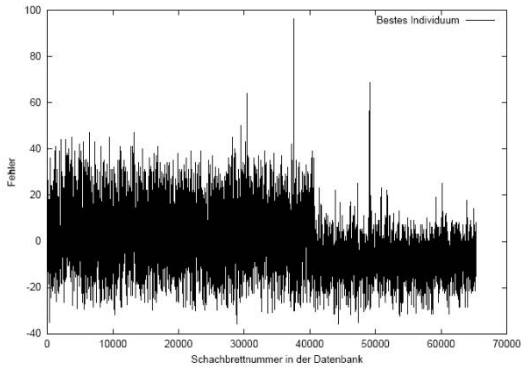

Abbildung 6.19: Fehler des besten Individuums der Materialbewertung Defensiv auf allen Schachstellungen der Datenbank. Der durchschnittliche Fehler beträgt 8,26.

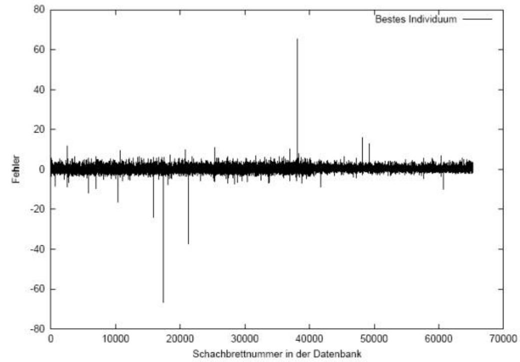

Abbildung 6.20: Fehler des besten Individuums der Materialbewertung Aggressiv auf allen Schachstellungen der Datenbank. Der durchschnittliche Fehler beträgt 0,88.

gaben selten Zuglisten zurück, die mehr als fünf Züge beinhalteten. Eine genaue Analyse der Zuglisten zeigte, dass die Individuen entweder einfache Heuristiken entwickelt hatten oder aufgrund von Design- und Implementierungsentscheidungen zu einem gewissen Verhalten tendierten. So lieferte zum Beispiel das Individuum der Desperado Züge nur Züge, die auf der a-Linie des Schachbrettes lagen. Die a-Linie wurde intern durch die Zahl eins dargestellt und alle Register wurden immer mit der Zahl eins initialisiert. Das Individuum für die Gefährlichen Züge lieferte immer eine Liste von Zügen, deren Zielfeld auf der gleichen Linie lag.

Im Hinblick auf die Wichtigkeit der Ebene für die Evolution der Varianten mussten wir feststellen, dass alle evolvierten Module häufig Zuglisten mit weniger als fünf Zügen aufstellten und eine so kleine Auswahl aufgrund der Güte der einzelnen Züge nicht ausreichend war um ein spielstarkes Individuum zu erzeugen. Daher haben wir uns dazu entschlossen den Strafterm für die Längenbeschränkung aus der Fitnessberechnung zu entfernen, um somit größere Zuglisten zu erhalten. Wir erwarteten durch diese Veränderung der Fitness, dass die Individuen im schlimmsten Fall Listen mit allen möglichen Zügen in verschiedenen Permutationen erzeugten. Dies geschah dann auch in allen durchgeführten Läufen. Eine Analyse der verschiedenen Zuglisten zeigte dabei keine sinnvolle Umordung der Listen bezogen auf das Evolutionsziel. Es waren Zuglisten mit allen möglichen Zügen einer Stellung, die in einer willkürlichen Reihenfolge sortiert wurden. Da diese Ebene für das Gesamtindividuum eine große Bedeutung hat, sich durch die Evolution aber keine Individuen gebildet hatten, die die Aufgabe sinnvoll lösten, modellierten wir zwei Individuen *von Hand*. Wir nutzten dafür die der Evolution zur Verfügung gestellte Syntax und den Sprachumfang des Interpreters mit allen erlaubten Operationen. Dies ermöglichte uns Individuen zu erzeugen, die die gestellte Aufgabe hinreichend gut lösten. Erstaunlich ist hierbei, dass die Evolution es nicht geschafft hat entsprechende Individuen zu erzeugen, obwohl sie die gleichen Möglichkeiten hatte wie wir. Es ist möglich, dass dies durch größere Populationen und längere Laufzeiten möglich gewesen wäre, jedoch waren die von uns gewählten Parameter bezogen auf die Laufzeit der Evolution und die Ressourcen, die uns zur Verfügung gestanden haben, maximal gewählt.

Da wir uns bei der Erzeugung der Individuen auf die Ausdruckskraft der Sprache beschränkten, die auch für die Evolution galt, zeigte sich, dass es auch uns nicht möglich war Individuen für die Problemstellung *Tauschzüge* und *Materialzüge* zu entwickeln. Für beide Aufgabenstellungen wäre es notwendig gewesen probehalber Züge auszuführen, dies konnte aber von einem Individuum nicht durchgeführt werden. Nach weiteren Versuchen wurden dann die Individuen für *Beste Züge* und *Gefährliche Züge* als Individuen von Hand implementiert.

6.1.3 Das Schachindividuum

Ein Gesamtindividuum besteht aus vier Modulen, der Variantenebene und jeweils acht Individuen der Material- und Planungsebene. Der Aufbau der Variantenebene ist in Kapitel 5.1.1.5 genauer beschrieben. Nach den Problemen der Planungsebene konnten dem Gesamtindividuum jedoch nur zwei verschiedene Module zur Verfügung gestellt werden. Dies schränkt zwar die Variabilität des Individuums ein, hinderte jedoch ein Individuum nicht, sinnvoll Schach zu spielen.

Im Unterschied zu den ersten beiden Ebenen des Individuums wurden nun zur Berechnung der Fitness Schachspiele durchgeführt. Da ein Schachspiel zur Bewertung nicht ausreichend ist, mussten die Individuen mindestens fünf Schachspiele durchgeführt haben, bevor sie durch den Selektionsmechanismus ausgewählt werden konnten, um anschließend rekombiniert oder mutiert zu werden. Spätestens nach zehn Spielen wurde ein Individuum zur Rekombination oder Mutation ausgewählt. Individuen, die fünf Pflichtspiele noch nicht durchgeführt hatten, wurden als *junge Individuen* bezeichnet. Wenn nun ein Schachspiel durchgeführt wurde, wurde mit einer Wahrscheinlichkeit von 90% ein Individuum aus der Menge der *jungen Individuen* gewählt. Mit einer Wahrscheinlichkeit von 10% wurde ein beliebiges Individuum aus dem Pool oder eines der zehn Schachprogrammen mit verschiedenen Spielstärken ausgewählt. Das zweite Individuum wurde dann entsprechend der Fitness (Elo-Zahl) des Individuums ausgewählt, dazu wurde der Durchschnitt aus der Fitness des Individuums und der durchschnittlichen Fitness des Pools berechnet. Dann wurde dieser Wert genutzt um ein Individuum im Pool zu finden, das etwa die gleiche Elo-Zahl hat. Anschließend wurde die Farbe ausgewählt mit der das Individuen das Spiel bestreitet.

Die Population enthielt 100 Individuen und in jeder Generation wurden 100 Spiele gespielt, durch die Parallelsierung des Systems wurden jeweils 20-30 Spiele gleichzeitig ausgeführt [1]. Ein Spiel wurde nach spätestens 80 Halbzügen abgebrochen, der Sieger der Partie wurde dann durch die Materialüberlegenheit bestimmt. Jedes Individuum durfte pro Zug im Suchbaum maximal 20000 Knoten expandieren und bis zur Tiefe 6 vordringen. In jedem fünften Spiel wurde versucht, ein schlechtes Individuum durch ein neues

[1] Die Anzahl der Spiele ist von der Anzahl freier Rechner abhängig.

Individuum zu ersetzen. Ein Individuum durfte aber erst nach sieben Spielen ersetzt werden und das neue Individuum durfte nur durch Individuen erzeugt werden, die nicht mehr zu den jungen Individuen zählten, also mehr als fünf Spiele durchgeführt hatten. Um nun ein Individuum zu ersetzen wird ein Turnier mit zehn Individuen durchgeführt. Dann wurde das neue Individuum mit je gleicher Wahrscheinlichkeit durch eine Mutation des besten Individuums des Turniers oder durch eine Rekombination der besten beiden Individuen des Turniers erzeugt.

Die Abbildung 6.21 zeigt nun den Verlauf der Elowerte über 160 Generationen. Wenn man diese Kurve betrachtet, scheint ab ungefähr der 75.ten Generation ein starker Einbruch zu passieren, von dem sich die Evolution nicht mehr vollständig erholt. Was man dort sieht, ist jedoch ein Artefakt der Elobewertung, denn jedesmal, wenn ein Individuum gewinnt, wird die Elo-Zahl besser, wie von der Elobewertung definiert. Erst wenn dieses Individuum gegen ein anderes verliert, sinkt seine Bewertung wieder, das heißt, in der Phase des Anstiegs gibt es kein Individuum im Pool, das besser ist, somit steigt seine Elobewertung immer weiter. Wenn nun ein neues Individuum besser ist, sinkt die Bewertung des alten Individuums immer dann, wenn es verliert. Gleichzeitig steigt die Bewertung des neuen Individuums. Die Phasen der fallenden Bewertung zeigen somit immer das Erscheinen eines neuen, besseren Individuums. Da die Elobewertung nur eine relative Güte der Individuen im Pool angibt, heißt das aber auch, dass ein Individuum der Generation 30 mit einer Elobewertung von 700 nicht besser sein muss als ein Individuum der Generation 150 mit einer Bewertung von 650.

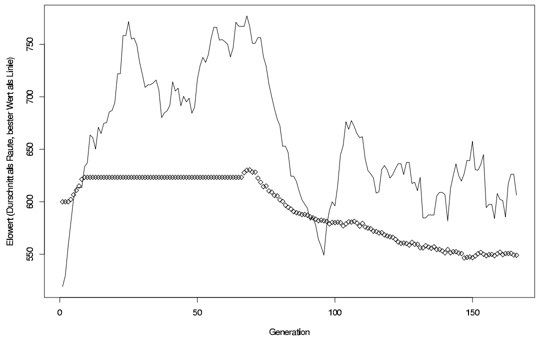

Abbildung 6.21: Die Entwicklung der Elowerte über die Zeit.

Für eine genaue Beurteilung der Spielstärke des besten Individuums muss man die Ergebnisse aus Spielen gegen die selbstgeschriebenen Schachprogramme untersuchen. Die Tabelle 6.2 listet die Ergebnisse des besten Individuums der Evolution gegen fünf verschiedene Gegner auf. Der einfachste Gegner ist ein Programm, das zufällig einen legalen Zug angibt. Die nächste Klasse sind zufällige Individuen, denn durch die Struktur der Individuen und ihrer Funktionen sind sie besser als zufällige Züge. Abschließend kommen die Klassen, die durch Minimax-Programme der Suchtiefen 1-3 erzeugt werden. Für diese Tabelle wurden 100 Spiele pro Gegner durchgeführt. Während der Evolution wurden zur Analyse des besten Individuums die gleichen Spiele durchgeführt, es wurden jedoch aus Zeitgründen nur fünf Spiele pro Gegner durchgeführt. In den Plots 6.22 und 6.23 sind die Ergebnisse für ein Minimax-Programm der Suchtiefe eins und zwei zu sehen, diese zeigen am besten die Entwicklung innerhalb der Population. So kann man in beiden Plots erkennen, dass die besten Programme während der Generation 100 und 150 in der Population existieren. Dies kann man gut an der geringen Varianz der Ergebnisse in der Abbildung 6.22 erkennen und an den Ergebnissen in der Abbildung 6.23. Dabei würde ein Ergebnis von 1,0 bedeuten, dass das Individuum fünf Mal gegen das Minimax-Programm gewonnen hat. Man kann aber in beiden Abbildungen kaum eine Ähnlichkeit zum Plot der Elowerte in Abbildung 6.21 erkennen.

Die Tabelle 6.2 zeigt nun eine genauere Analyse der Spielstärke des besten Individuums. Daraus kann man erkennen, dass das Individuum ungefähr die Spielstärke eines Schachprogramms der Suchtiefe zwei hat. Das Individuum hätte zwar 20.000 Knoten im Suchbaum expandieren können, was ausgereicht hätte ein Programm der Tiefe drei zu schlagen, denn ein effizienter Suchalgorithmus benötigt für diese Suchtiefe etwa 10.000

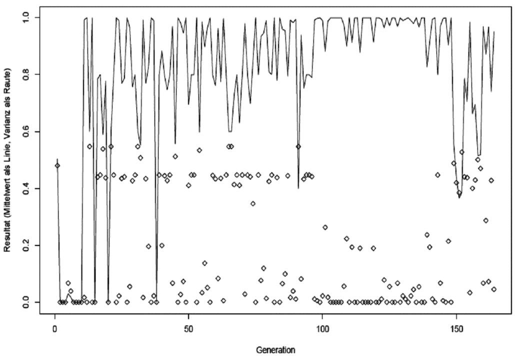

Abbildung 6.22: Der Plot zeigt das durchschnittliche Ergebnis von fünf Spielen des besten Individuums gegen einen MiniMax-Algorithmus der Tiefe 1. Die Rauten geben die Varianz der Ergebnisse an. Die Kurve gibt an, ob das Individuum gewonnen (1,0) oder verloren (0,0) hat.

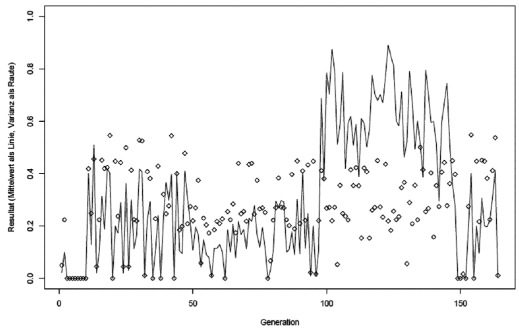

Abbildung 6.23: Der Plot zeigt das durchschnittliche Ergebnis von fünf Spielen des besten Individuums gegen einen MiniMax-Algorithmus der Tiefe 2. Die Rauten geben die Varianz der Ergebnisse an. Die Kurve gibt an ob, das Individuum gewonnen (1,0) oder verloren (0,0) hat.

Gegner	Zufall	zufälliges Individuum	Minimax (1)	Minimax (2)	Minimax (3)
Sieg (Individuum)	100	87	95	38	0
Überlegen	0	0	3	18	0
Remis	0	13	0	13	5
Unterlegen	0	0	2	13	8
Niederlage	0	0	0	18	87

Tabelle 6.2: Diese Tabelle enthält die Spielausgänge der Spiele des besten Individuums gegen verschiedene Gegner. Der Zufallsgegner ist ein Schachprogramm, das zufällige, aber legale Züge liefert. Die Minimax(x) Gegner sind Minimax-Algorithmen mit den Suchtiefen eins bis drei.

Knoten. Ein normaler Minimax-Algorithmus benötigt jedoch 38.000 Knoten. Das beste Individuum hat jedoch im Durchschnitt nur 531 Knoten pro Zug expandiert, was dem Niveau eines Alpha-Beta Suchprogramms der Suchtiefe zwei entspricht. Somit hat die Evolution zwar einen effizienten Algorithmus entwickelt, jedoch hat sie nicht alle Möglichkeiten genutzt, die ihr zur Verfügung standen. Hier muss man nun anmerken, dass wir nur einen Pool mit 100 Individuen genutzt haben, die Evolution nach 160 Generationen abgebrochen haben und nur eine Evolution durchführen konnten. Jedoch waren wir nicht in der Lage weitere Läufe mit dieser Größenordnung oder sogar mit mehr Individuen und einer längeren Evolution durchzuführen, da selbst diese Evolution bei einer parallelen Berechnung auf 28 Rechnern mehr als 14 Wochen benötigte. Da uns auch Testläufe mit kleineren Populationen und mit vorevolvierten Programmen keine Verbesserung zeigten, gehen wir davon aus, dass die erzielten Ergebnisse den Möglichkeiten des Systems entsprechen.

6.1.4 Fazit ChessGP

ChessGP ist ein Negativergebnis, das aber als Grundlage für die weiteren Systeme wichtig war. Das System konnte Schachindividuen evolvieren, jedoch war die Komplexität des Systems zu groß und die Aufgabenstellung einiger Module zu schwer gewählt. So erreichte das beste Individuum nur eine Spielstärke, die einem Alpha-Beta Algorithmus der Suchtiefe zwei entspricht. Somit hat dieses System wichtige Ergebnisse und Impulse für die weitere Entwicklung geliefert und durfte daher in dieser Diskussion nicht fehlen.

Ein positives Ergebnis sehen wir in der Hierachisierung des Gesamtindividuums, sie ist eine Möglichkeit komplexe Aufgaben durch ein GP-Individuum evolvieren zu lassen. Durch die Verwendung verschiedener Ebenen mit unterschiedlichen Funktionssets konnten gezielt Teilprobleme gelöst werden, die dann zu einem Gesamtsystem zusammengefasst werden konnten. Dies ermöglichte eine Reduktion der Komplexität der Gesamtaufgabe. Das Problem der Komplexitätsreduktion wurde aber nicht vollständig gelöst, wie wir am Verhalten der Variantenebene sehen konnten. Letztlich war es nicht möglich Individuen zu evolvieren, die das zuerst formulierte Ziel erreichten. Die Gründe sind hier in dem Ziel und den Möglichkeiten der Individuen zu sehen. So sollten möglichst kurze Zuglisten mit verschiedenen Ausrichtungen erzeugt werden, die von den Individuen ohne die Möglichkeit einen Suchbaum aufzubauen erzeugt werden sollten. Motiviert wurde dies durch psychologische Untersuchungen menschlicher Schachexperten, die Züge nicht durch die Abarbeitung eines Suchbaumes ermitteln, sondern indem sie die vorhanden Situation einschätzen. Dieses Verhalten sollte nachgeahmt werden, die Evolutionsläufe haben aber gezeigt, dass das System mit den vorhandenen Mitteln nicht in der Lage war die Aufgabe zu lösen. Auch eine Vereinfachung der Aufgabenstellung brachte nur wenig Erfolg, so dass wir an dieser Stelle *selbst* erzeugte Individuen nutzen mussten um ein spielendes Individuum zu entwickeln. Da wir hier nur Operationen aus dem Sprachumfang genutzt haben, spricht nichts gegen die prinzipielle Möglichkeit, dass solche Individuen durch eine Evolution erzeugt werden können.

Wichtig war auch die Erfahrung, dass eine Elo-Bewertung als Fitness für die Individuen nicht sinnvoll ist, da sie ein relatives Maß ist. Dies in Verbindung damit, dass Individuen hauptsächlich gegeneinander gespielt haben um die Fitness zu berechnen, führte dazu, dass die Elowerte kein objektives Maß der Spielstärke darstellten. Denn wenn ein Individuum besser war als alle anderen, stieg seine Fitness (Elowert) immer weiter, solange kein Individuum erzeugt wurde, was dieses Individuum schlagen konnte. An der parallel durchgeführten Bewertung mittels verschieden starker Programme konnte man auch erkennen, dass in der Population gute Individuen verloren gehen können. Dies ist aber für eine Evolution von Schachprogrammen aufgrund der hohen Rechenzeiten nicht praktikabel.

Grundsätzlich muss man zum ChessGP-System sagen, dass sich viele Ideen des Systems als zu komplex erwiesen haben. So waren die drei einzelnen Evolutionsprozesse zwar möglich, jedoch schränkt man die Möglichkeiten der Ebenen, miteinander zu interagieren, ein, da sie sich nur in den durch die Fitnessfunktion gegebenen Grenzen bewegen konnten. Auch einzelne Aufgabenstellungen, wie die der Variantenebene, waren zu komplex um adequat gelöst zu werden. Eine stärkere Anlehnung an Suchalgorithmen war somit ein Ergebnis des ChessGP-Systems, ebenso eine Vereinfachung der Evolution selbst.

6.2 qoopy

Das qoopy-System hat verschiedene Probleme, die durch das ChessGP-System aufgezeigt wurden, aufgegriffen und versucht diese zu beseitigen. Ein wichtiges Problem des ChessGP-System war das Problem der Rechenzeit für eine Evolution. So konnte trotz einer Parallelisierung nur eine kurze Evolution mit wenigen Individuen durchgeführt werden. Daher wurde das qoopy-System als eine internetbasierte Evolution durchgeführt, um hier die Vorteile des Peer-to-Peer Computing [2] zu nutzen, siehe Abschnitt 5.1.2.5. Dadurch konnte die Evolution mit sehr viel mehr Individuen und Generationen durchgeführt werden, als es bei dem ChessGP-System möglich war. Der Vorteil der großen Rechenkraft wurde durch den Nachteil erkauft, dass man weniger Einfluss auf die Evolution hatte, da wir den einzelnen Nutzern die Möglichkeit gegeben haben verschiedene Parameter der Evolution zu variieren, um so eine stärkere Bindung der Nutzer an das System zu erzielen. Dies war auch erfolgreich, so konnten über einen Zeitraum von mehr als drei Monaten über 1.300 Nutzer am Projekt gezählt werden. Zusätzlich wurden verschiedene Statistiken der einzelnen Nutzer im Internet veröffentlicht, um jedem Nutzer zu zeigen, wie gut *seine Individuen* sind und wie viel Rechenzeit er zur Verfügung gestellt hat. Die besten Individuen wurden in einer Top-100 Liste erfaßt.

Aufgrund der internetweiten Evolution wurden die besten Individuen zum Abschluss noch einer Nachbewertung unterzogen, um eine genauere Bewertung der Individuen zu erzielen. Die Nachbewertung wurde nicht im Internet durchgeführt, so dass mehr Spiele und auch Spiele gegen stärkere Gegner durchgeführt werden konnten als während der internetweiten Evolution.

6.2.1 Die Nachbewertung

Die Fitness der Individuen wird durch die Ergebnisse der Partien gegen Programme fester Suchtiefen bestimmt. Als Gegner standen den Individuen die Gegner $G_1, G_2, \ldots, G_7$ zur Verfügung, wobei der Index die Suchtiefe des Programms angibt, siehe auch Abschnitt 5.1.2.7. Das Spielergebnis ist ein Wert im Intervall von $[0, 1]$, wobei 0 einer Niederlage des Individuums entspricht, 0.5 einem Remis und 1.0 dem Gewinn der Partie für das Individuum. Da Partien zwischen dem Individuum und dem Gegnerprogramm sehr lange dauern können, gelten die gleichen Abbruchkriterien wie bei der Fitnessberechnung, so wurde nach maximal 50 Halbzügen eine Partie beendet. Dann erfolgte eine Materialbewertung der letzten Stellung im Intervall von $[0, 1]$ und diese wurde als Ergebnis des Spiels gewertet. Da die Individuen aufgrund der Blattbewertung nicht deterministisch spielten, ist war der Partieverlauf und somit das Ergebnis nicht sicher vorhersagbar. Somit musste jedes Individuum gegen jeden Gegner G_i mehrere Partien spielen um eine Aussage über die Güte der Individuen zu erhalten. Da die Nachbewertung auf einem eigenen Rechner am Institut durchgeführt wurde, haben wir in interessanten oder kritischen Fällen maximal 100 Partien pro Gegner durchgeführt, im Normalfall 20 Partien. Eine Nachbewertung mit je 20 Partien pro Gegner dauerte 1-2 Tage. Die besten 200 Individuen der Evolution sollten nachbewertet werden, da dies jedoch zu zeitaufwendig wäre, wurde eine Vorauswahl getroffen. Dabei wurde eine Nachbewertung mit weniger Partien durchgeführt und überprüft, ob es zu einer starken Überbewertung der Fitness während der Evolution gekommen war. Es hat sich gezeigt, dass die Fitness der Individuen um eine bis drei Fitnessklassen während der Evolution besser bewertet wurde als durch die Nachbewertung. Daher wurden am Ende 82 Individuen aus der Liste der 200 besten Individuen nachbewertet.

Die Tabelle 6.3 führt die Fitness und die durchschnittlichen Spielergebnisse der besten 10 Individuen nach der Nachbewertung auf. Gleichzeitig wird das durchschnittliche

Fitness	G_3	G_4	G_5	G_6
10,4816	1,0 100	0,8807 50	0,7528 50	0,10 50
10,3981	1,0 100	0,9803 50	0,5486 50	0,1652 50
10,3943	1,0 100	0,9218 100	0,6155 100	0,1809 93
10,3803	1,0 100	0,8898 100	0,6274 100	0,2065 100
10,3647	1,0 100	0,9318 50	0,5921 50	0,1717 50
10,3572	1,0 100	1,0 20	0,4285 20	0,2072 20
10,3479	1,0 100	0,8841 20	0,7205 20	0,1397 20
10,3425	1,0 100	0,9431 20	0,4437 20	0,2503 20
10,3345	1,0 100	0,95 20	0,582 20	0,425 20
10,331	1,0 100	0,95 20	0,6649 20	0,0748 20

Tabelle 6.3: Diese Tabelle zeigt die Fitness der besten Individuen, die durch die Nachbewertung ermittelt wurden sowie das durchschnittliche Spielergebnis der Individuen gegen die verschiedenen Gegnerprogramme G_i und die Anzahl der Spiele gegen das entsprechende Gegnerprogramm. Das durchschnittliche Spielergebnis liegt zwischen 0 und 1, wobei 0 den Verlust der Partie bedeutet und 1 den Gewinn.

Spielergebnis der Individuen gegen die verschiedenen Programme fester Suchtiefe mit der Anzahl der Spiele angegeben. Man kann an der Tabelle sehr gut sehen, dass Programme der Suchtiefe fünf (G_5) noch geschlagen werden können, aber Programme der Suchtiefe sechs (G_6) nicht mehr.

Wichtig ist hierbei, dass die Individuen im Durchschnitt 60.000 Knoten expandieren um diese Ergebnisse zu erreichen. Ein einfaches Alpha-Beta Schachprogramm muss bei einer Suchtiefe von fünf Halbzügen 897.070 Knoten pro Zug expandieren und ein F-Negascout Algorithums benötigt auch noch 120.000 Knoten bei dieser Suchtiefe. Diese Programme würden einen Fitnesswert von 10 erreichen, die Individuen erreichen eine etwas bessere Fitness und nutzen dabei weniger Resourcen als die bekannten Schachprogramme. Im Vergleich zu einem F-Negascout Algorithmus ist die Ersparnis 50% und im Vergleich zu einem Alpha-Beta Algorithmus sogar 94%.

Die qoopy Individuen beruhen, wie im Kapitel 5.1.2.1 beschrieben, auf einen Alpha-Beta Algorithmus und können diese an drei Punkten optimieren. Der Evolution ist es also gelungen einen bekannten Algorithmus zu verbessern, denn wie man in der Tabelle erkennen kann, liegt das durchschnittliche Spielergebnis nicht bei $0,5$, was einem Remi entsprechen würde und somit die gleiche Spielstärke anzeigen würde, sondern in acht von zehn Fällen deutlich über $0,5$, das beste Ergebnis war sogar $0,75$. Das Spielergebnis bezieht sich immer auf das Individuum und ist auf den maximalen Materialwert ohne die Könige normiert. Das heißt, ein Ergebnis von eins für ein Individuum bedeutet, dass es keine Figuren verloren hat, das Gegnerprogramm nur noch den König besitzt oder das Gegnerprogramm schachmatt ist. Eine Figur hat einen Einfluss von $\frac{\text{Figurwert}}{4060}$ für einen

Bauern, mit einem Figurwert von 100 ergibt das $\frac{100}{4060} \approx 0,0246$. Der Wert $0,75$ liegt um $0,25$ über dem Wert eines Remis, das heißt, das Individuum hat einen Materialvorteil vom $0,25$, dies sind ungefähr $\frac{1000}{4060}$ und entspricht bei den für qoopy gewählten Figurwerten einer Dame und einem Bauern oder zwei Türmen, andere Figurwerte können in Abschnitt 5.1.2.7 nachgeschlagen werden.

6.2.2 Analyse

Ein Individuum des qoopy-Systems besteht wie ein ChessGP Individuum aus drei Ebenen, jedoch bestehen die Ebenen im letzteren Individuum aus mehreren Modulen. Im qoopy Individuum wird jede Ebene durch ein Modul dargestellt und die Evolution eines qoopy Individuums wird in einem Schritt durchgeführt. Dadurch erhält man, anders als im ChessGP-System, nur einen Fitnesswert, der die Gesamtleistung des Individuums beschreibt. Somit kann eine Bewertung der einzelnen Module erst durch eine anschließende Analyse erfolgen. Da es sich aber bei den Individuen um ein Gesamtsystem handelt, kann auf der anderen Seite eine allgemein gültige Bewertung der einzelnen Module nicht durchgeführt werden. Denn für sich alleine betrachtet kann ein Modul ein schlechteres Ergebnis als das Modul eines anderen Individuums haben, aber in der Kombination mit den anderen Modulen doch zu besseren Ergebnissen führen als das *bessere* Modul.

6.2.2.1 Tiefenmodul

Das Tiefenmodul hat im Gesamtindividuum die Aufgabe die maximale Suchtiefe für jeden Pfad im Suchbaum zu bestimmen, dadurch wird ein großer Einfluss auf die Spielstärke des Individuums und die Anzahl der expandierten Knoten im Suchbaum ausgeübt. Es hat sich gezeigt, dass zufällig initialisierte Individuen sehr spielschwach sind. Dies liegt hauptsächlich an dem Tiefenmodul, denn die zufälligen Module beschränkten die Tiefe entweder kaum oder sehr stark. Wenn ein Tiefenmodul die Suchtiefe extrem beschränkt, werden durch den Rahmenalgorithmus mindestens zwei Halbzüge durchgeführt. Kommt es zu keiner Beschränkung, werden maximal zehn Halbzüge durchgeführt. Im ersten Fall erhält man maximal ein Individuum, welches die Fitnessklasse vier erreicht und im zweiten Fall kann nur ein sehr kleiner Teil des Suchbaums expandiert werden. Da die Suche erst nach 10 Halbzügen abgebrochen wird und pro Zug maximal 100.000 Knoten expandiert werden (siehe hierzu Abschnitt 5.1.2.1), wird ein extrem linkslastiger Suchbaum erzeugt. Dies ist zwar immer das Ergebnis einer Alpha-Beta Suche, aber in Verbindung mit einem zufälligen Zugsortierungs- und Stellungmodul erreichen diese Individuen noch schlechtere Ergebnisse als ein Individuum mit einem restriktiven Tiefenmodul. Denn in diesem Fall könnten von dem Individuum vielleicht nur die schlechten Züge expandiert werden, aber für ein gutes Resultat braucht die Alpha-Beta Suche eine gute Zugsortierung

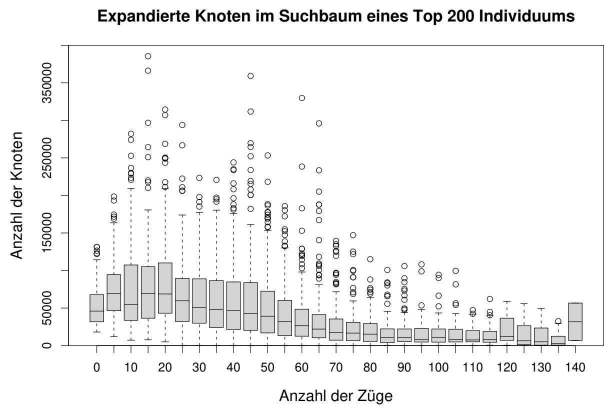

Abbildung 6.24: Ein Boxplot Diagramm der durchschnittlich verbrauchten Knoten pro Zug über eine Spieldauer von 140 Halbzügen. Das Ergebnis wurde über 50 Spiele mit einem evolvierten Individuum ermittelt.

und Stellungsbewertung. Da dies für zufällige Individuen im allgemeinen nicht gilt, erreichen Individuen mit einem restriktiven Tiefenmodul bessere Ergebnisse, da diese die gesamte Breite des Suchbaums bis zur Tiefe zwei explorieren und so trotz einer schlechten Zugsortierung auch gute Züge expandieren können.

Während der Evolution bildete sich schnell ein Modul, das die Suchtiefe auf fünf Halbzüge begrenzte. Wenn man sich vor Augen führt, dass dem Individuum 100.000 Knoten pro Zug zur Verfügung stehen, ist dies eine gute Suchtiefe. Denn ein guter Alpha-Beta Algorithmus benötigt 50.000 Knoten bei einer Suchtiefe von fünf und 320.000 Knoten bei einer Suchtiefe von sechs. Somit könnte ein Individuum selbst mit schlechter Zugsortierung noch einigermaßen gut spielen. Erst sehr spät entwickelten sich Individuen, die in den ersten Zügen nur zwei oder drei Halbzüge expandierten. Anschließend expandierten die Individuen den Suchbaum wieder bis zur Tiefe fünf. Die durch das Individuum eingesparten Knoten konnten so in der späteren Phase des Spiels genutzt werden. Jedoch kam es während der gesamten Evolution nicht dazu, dass ein Tiefenmodul entwickelt wurde, das mehr als fünf Halbzüge tief suchte. Die Abbildung 6.24 zeigt das oben beschriebene Verhalten, welches die Reduktion der Suchtiefe zum Beginn einer Partie durchführt. Man kann gut erkennen, dass es ab dem fünften Halbzug zu einem Anstieg des Knotenverbrauchs führt. Nach dem 20. Halbzug kommt es wieder zu einer Reduktion der expandierten Knoten, die mehr oder weniger bis zum 50. Halbzug konstant bleibt. Anschließend kommt es kontinuierlich zu einer weiteren Reduktion des expandierten Knotens, dies wird jedoch nicht durch das Individuum verursacht, sondern ergibt sich durch die geringe Anzahl von Figuren auf dem Spielfeld zum Ende der Partie und die konstant gehaltene Suchtiefe des Individuums.

6.2.2.2 Zugsortierungsmodul

Das Zugsortierungsmodul hat neben dem Tiefenmodul den größten Einfluss auf die Anzahl der expandierten Knoten und die Spielstärke der Individuen. Denn das Alpha-Beta Verfahren (siehe Kapitel 2.2.5) betrachtet für jede Stellung des Spielbaums, in dem die Suche fortgesetzt werden soll, alle möglichen Stellungen, die von dieser Stellung aus durchgeführt werden können. Eine gute Vorsortierung der Zugliste kann die Suche erheblich beschleunigen, denn wenn dem Algorithmus vorzeitige gute Alternativen bekannt sind, verkleineren sie das (α, β) Suchfenster oder ermöglichen den Abbruch der Suche für diesen Weg im Suchbaum durch einen Alpha- oder Beta-Schnitt. Wenn man den Worst Case annimmt und ein Zugsortierungsmodul die Züge immer in der falschen Reihenfolge ordnet, dann muss ein Alpha-Beta Algorithmus alle Züge expandieren, da keine Verkleinerung des Suchfensters oder ein Schnitt im Suchbaum möglich sind, siehe Abschnitt 2.2.3. Dann würde der Algorithmus alle Knoten expandieren und es würde zu keiner Einsparung kommen. Aber selbst die besten Schachprogramme expandieren 20-30% mehr Knoten, als in einem optimalen Suchbaum expandiert würden. Eine Verbesserung der Zugsortierung führt somit zu einer starken Reduktion der Größe des Suchbaums.

Im Rahmen der Analyse des Zugsortierungsmoduls wurden spielstarke Individuen aus der Top 200 Liste mit zufälligen Individuen und dem Algorithmus der Gegnerprogramme bezüglich des Knotenverbrauchs bei einer konstanten Suchtiefe verglichen. Durch diesen Ansatz wird die Güte der evolvierten Zugsortierungsmodule, bezüglich der Reduktion des Suchraums zu den beiden Vergleichsalgorithmen, untersucht. Hierzu wurden die Zugsortierungsmodule in einen Alpha-Beta Algorithmus eingesetzt, so dass die Ergebnisse unabhängig von dem Rest des Individuums waren. Das Gegnerprogramm ist jedoch weiterhin das Programm, welches auch während der Evolution verwendet wurde (siehe Abschnitt 5.1.2.7). Die Stellungsbewertung des Apha-Beta Algorithmus entspricht der des Gegnerprogramms. Die Programme wurden auf

4.500 Brettstellungen getestet, die aus 50 Partien stammen, die der Gegneralgorithmus gegen sich selbst gespielt hat. Damit wird jede Zugsortierungsstrategie auf identischen Situationen getestet und die Ergebnisse sind somit direkt vergleichbar.

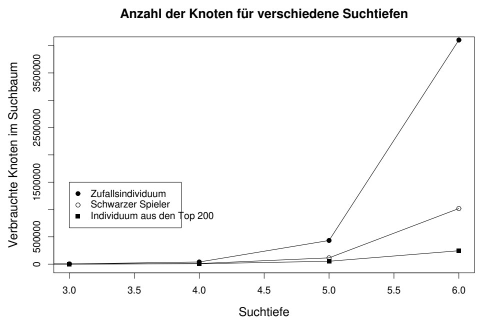

Abbildung 6.25: Durchschnittliche Anzahl verbrauchter Knoten im Suchbaum für verschiedene Suchtiefen. Den größten Unterschied beim Knotenverbrauch erkennt man für die Suchtiefe 6. Die Daten wurden über 50 Spiele gemittelt.

Zur Analyse wurden Module aus drei Individuen der Top 200 Liste ausgewählt und drei zufällige Module erzeugt. Bei der Analyse hat sich gezeigt, dass sich während der Evolution Zugsortierungsmodule entwickelt haben, die bei einer Suchtiefe von fünf Halbzügen einer zufälligen Sortierung bezüglich des Knotenverbrauchs um den Faktor 7,8 und bei einer Suchtiefe von sechs Halbzügen um einen Faktor von 16,7 überlegen sind. Gleichzeitig ziegte sich, dass die evolvierten Module den Gegnerprogrammen bei einer Suchtiefe von fünf um den Faktor 1,8 und bei einer Suchtiefe von sechs sogar um 7,8 überlegen waren. Dies ist besonders erstaunlich, da die Gegnerprogramme im Gegensatz zu dem Alpha-Beta Algorithmus, der als Testumgebung für die zufälligen und evolvierten Module diente, ein Scout-Algorithmus mit der F-Verbesserung kombiniert mit den Techniken des Iterative Deepening und den Killermoves nutzten, was gegenüber einem einfachen Alpha-Beta Algorithmus eine erhebliche Einsparung von Knoten im Suchbaum zur Folge hat. Trotzdem konnten die durch die Evolution erzeugten Module die Gegnerprogramme bezogen auf den Knotenverbrauch im Suchbaum schlagen. Die Abbildung 6.25 zeigt den Knotenverbrauch der drei Gruppen für drei, vier, fünf und sechs Halbzüge. Deutlich kann man erkennen, dass die Unterschiede in der Zugsortierung erst ab einer Suchtiefe von fünf richtig erkennbar werden. Dies liegt einerseits an der Skalierung der y-Achse, jedoch lagen die Ergebnisse aller drei Klassen sehr eng zusammen, so dass man nicht sagen konnte, ein Modul ist deutlich besser oder schlechter als ein anderes. Dies liegt daran, dass die zufälligen Module durch die Einschränkungen der Parameter, die vorgegeben sind, schon sinnvolle Lösungen für die Problemstellung sind. Jedoch konnte die Evolution die Parametereinstellung so weit verbessern, dass es dem Modul sogar möglich war das Gegnerprogramm in Bezug auf die Anzahl der expandierten Knoten zu unterbieten.

6.2.2.3 Stellungsmodul

Das Stellungsmodul hat die Aufgabe materielle und positionelle Merkmale einer Stellung zu bewerten. Die Stellungsmodule der Individuen sind von der Struktur her identisch zu denen der Gegnerprogramme, nur die Werte der einzelnen Merkmale des Algorithmus unterscheiden sich. Eine genaue Beschreibung des Stellungsmoduls und seiner Merkmale findet man in Abschnitt 5.1.2.2. Um das Modul zu analysieren sind wir ähnlich vorgegangen wie für das Zugsortierungsmodul, hierzu wurde das Modul in ein Alpha-Beta Programm integriert. Dann wurden für die Suchtiefen eins bis fünf 500 Partien gegen das Gegnerprogramm aus der Evolution mit der entsprechenden Suchtiefe gespielt. Es wurden drei Module aus Individuen der Top 200 Liste ausgewählt und fünf zufällig erzeugte Module verwendet. Die zufälligen Module wurden wie bei der Initialisierung der Evolution erzeugt, also durch die Berechnung einer normal verteilten Zufallsvariable im zulässigen Intervall für das entsprechende Merkmal.

Das Balkendiagramm 6.26 zeigt, wieviele Partien die verschiedenen Module gegen das Gegnerprogramm, bei einer Suchtiefe von fünf, gewonnen haben. Man sieht, dass die zufälligen Module im Bereich von 44 und 61 % liegen und die evolvierten im Bereich von 58 und 70 %. Dies zeigt zuerst einmal, dass alle getesten evolvierten Stellungsmodule dem Standardmodul überlegen waren. Denn wären sie gleich stark, müssten die Ergebnisse in einem kleinen Bereich um 50 % liegen, denn in Experimenten mit dem Stellungsmodul des Gegnerprogramms lagen die Ergebnisse im Bereich von 48 bis 53 %. Da alle Programme die gleiche Suchtiefe benutzen und somit eine ähnliche Anzahl von Knoten expandieren, liegt der Vorteil der evolvierten Stellungsmodule in einer besseren Bewertung der positionellen Kriterien. Dieser Vorteil wird aber erst mit den größeren Suchtiefen sichtbar. Denn für diese Analyse wurden auch die Suchtiefen eins bis vier untersucht, jedoch zeigten sich hier nicht so große Unterschiede zwischen den Ergebnissen wie in dem Experiment mit Suchtiefe fünf. Grundsätzlich waren die evolvierten Module immer etwas besser als die zufälligen, jedoch lag der maximale Unterschied bei diesen Versuchen bei 16 % und im Fall der Suchtiefe fünf 26 %.

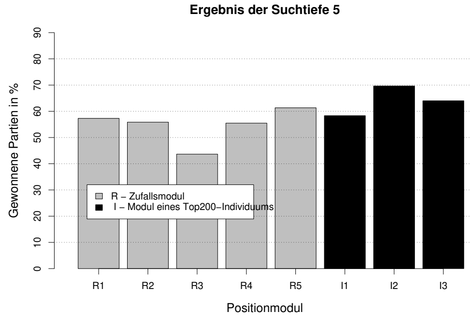

Abbildung 6.26: Das Balkendiagramm zeigt die Ergebnisse der Suchtiefe fünf für die fünf zufälligen Module und die drei evolvierten Stellungsmodule. Sie wurden in 500 Partien getestet. Das beste evolvierte Modul konnte 70 % der Partieen gegen das Standardmodul gewinnen, aber auch das beste zufällige Individuum konnte 61 % der Partien für sich entscheiden.

Erstaunlich an diesen Ergebnissen sind jedoch auch die großen Unterschiede in den evolvierten Modulen, denn die Module wurden aus Individuen entnommen, die eine ähnliche und sehr gute Fitness hatten. Daraus folgt, dass man über die Güte eines Moduls, bezogen auf diese Testumgebung, eigentlich keine zuverlässige Aussage treffen kann. Das Ergebnis legt die Vermutung nahe, dass die Aufgabe des Moduls von der Umgebung abhängig ist, in der es arbeitet. Dadurch ist es nicht möglich eine objektive Güte der Module untereinander zu erhalten.

6.2.3 Fazit qoopy

Das qoopy-System zeigte im Gegensatz zum ChessGP-System sehr gute Ergebnisse, so konnte das System spielstarke Schachprogramme evolvieren, die im Vergleich mit bekannten Algorithmen sogar sehr gut abschneiden. Der Ansatz, einen Alpha-Beta Algorithmus zu nehmen und ihn durch evolutionäre Verfahren zu verbessern, ist somit insgesamt betrachtet ein großer Erfolg. Das beste Individuum aus der Evolution erreichte eine Fitness von 10,48 und ist somit besser als ein Schachprogarmm mit einer festen Suchtiefe von fünf, dies entspricht einem Elowert von 1.400, siehe Abbildung 6.35. Dies entspricht einem Vereinsspieler der letzten bis vorletzten Klasse. Bezogen auf bekannte Schachprogramme sind die Individuen zwar immer noch sehr schwach, jedoch muss man hier den Rechenaufwand der Programme bezogen auf ihre Spielstärke vergleichen. Dabei zeigte sich während der Nachbewertung in 60 Spielen, dass ein Individuum im Durchschnitt 58.410 Stellungen pro Zug expandiert. Ein einfacher Alpha-Beta Algorithmus expandiert bei einer festen Suchtiefe von fünf im Durchschnitt 897.070 Stellungen pro Zug. Der von uns als Gegnerprogramm verwendete F-Negascout Algorithmus expandiert 120.000 Stellungen pro Zug bei dieser Suchtiefe. Das heißt, die Evolution hat einen Alpha-Beta Algorithmus so optimiert, dass er nur 6 % der Ressourcen eines einfachen Alpha-Beta Algorithmus benötigt und bezogen zum F-Negascout Algorithmus nur 50 % der Stellungen betrachten muss, um diesen zu besiegen. Durch den einfachen Aufbau der Individuen konnte eine Analyse der Individuen durchgeführt werden. Diese zeigte, dass die Zugsortierung die größte Auswirkung auf die Spielstärke der Individuen hat. Gleichzeitig müssten bessere Tiefen- und Stellungsmodule die Spielstärke der Individuen noch weiter steigern, jedoch war das im Rahmen von qoopy nicht so einfach, da wir uns durch die Internetevolution zwar den Vorteil der großen Rechenkraft gesichert haben. Gleichzeitig haben wir aber auch einige Nachteile dadurch eingekauft, wie eine sehr geringe Kontrolle über die einzelnen Evolutionsläufe, *Manipulationen* einiger User bei der Evolution und ein Geschwindigkeitsnachteil der Java Implementierung.

Das Problem der geringen Kontrolle über die Gesamtevolution war zwar am Anfang von uns gewollt, damit eine höhere Bindung der User zu *ihren Individuen* entstand und sie somit mehr Rechenzeit zur Verfügung stellten. Jedoch war dies aus der jetzigen Sicht betrachtet ein Fehler. So konnte nur ein Evolutionslauf durchgeführt werden, der zwar mit verschiedenen Einstellungen in den verschiedenen Demes durchgeführt wurde, jedoch war es nun nicht mehr möglich eine Verbindung zwischen der Güte einer Evolution zu ihren Einstellungen herzustellen. Es war auch kaum möglich Evolutionsläufe mit neuen Individuentypen durchzuführen, da die User lieber die *alte Evolution* weiter laufen ließen. Der Versuch eine zweite Evolution mit veränderten Individuen durchzuführen, scheiterte an der geringen Resonanz. Die Bindung der User zu der Evolution führte auch dazu, dass einige User die Fitnessbewertung manipuliert haben, indem sie das Feature *Spielabbruch* genutzt hatten. Durch den Spielabbruch konnte eine Auswertung zu jedem Zeitpunkt abgebrochen werden, dies war notwendig, da wir den Rechner nicht immer blockieren konnten. Wurde die Auswertung dann später wieder gestartet, wurde die Auswertung normalerweise beim letzten Zug der Partie wieder gestartet. Jedoch konnte man auch von Neuem beginnen, indem man das Individuum vorher gespeichert hatte. Dies nutzten einige User aus und brachen Spiele ab, wenn ihre guten Individuen Gefahr liefen ein Spiel zu verlieren und starteten es dann neu. So konnte man bessere Fitnessbewertungen erreichen, da die Individuen nicht ganz deterministisch spielten. Als dies im Forum bekannt wurde, haben wir die Nachbewertung der besten Individuen eingeführt, was zu einer Reduktion vieler Fitnesswerte führte.

Bei der Evolution wäre es daher besser gewesen, so vorzugehen, dass die User nicht eingreifen können und nur Rechenzeit zur Verfügung stellen, wie dies bei Seti-At-Home gemacht wird. Dadurch hätte man die ganze Kontrolle über die Evolution erhalten. Jedoch wäre der Implementationsaufwand für das System viel größer, da man dann nicht ganze Spiele auf einem Client durchführen könnte, denn ein Spiel kann mehrere Stunden dauern. Hier müsste man eine Lösung finden, welche die Bewertung der Individuen in kleinere Pakete aufteilt, damit eine Berechnung auf einem Client maximal ein paar Minuten dauert, so dass der Verlust einer Berechnung nicht groß ist.

6.3 GeneticChess System

In diesem Kapitel werden die Ergebnisse des GeneticChess Schachsystems vorgestellt, es ist das letzte System, welches entwickelt wurde, und versucht somit alle Ergebnisse der Vorgängersysteme in sich zu vereinen. Genau wie das qoopy System besteht es aus den drei Modulen Materialbewertung, Zugbewertung und Tiefenmodul. Jedoch sind diese Module in ihrem Aufbau und in ihrem Verhalten komplexer, dadurch ist eine Analyse der einzelnen Module nicht so einfach möglich, wie es für das qoopy System war. So konnte im qoopy System für das Tiefenmodul nach einer Analyse das Verhalten des Moduls erklärt werden, dies ist hier durch die Menge verschiedener Verhaltensweisen nicht möglich. Man kann zum Beispiel sehen, dass einige Individuen die maximale Anzahl von Knoten immer ausnutzen, andere jedoch in jedem Spiel variabel reagieren. Auch die Material- und Zugbewertung kann nicht wie im qoopy System in ein Standard Alpha-Beta Programm eingesetzt und getestet werden, da ihr Verhalten im GeneticChess System nicht so eng begrenzt ist. Tests haben gezeigt, dass Module als schlecht bewertet wurden, obwohl sie aus sehr guten Individuen stammen. Daher kann man an dieser Stelle keine objektive Aussage über die Leistungsstärke der Module als solches machen. Dies konnte schon im qoopy System gezeigt werden. Dies zeigt aber auch, dass sich die Module während der Evolution gegenseitig beeinflusst haben und somit besser als Gesamtindividuum bewertet werden sollten.

Zur Evolution der Individuen wurden Läufe mit einer Knotenbeschränkung von 100.000 und 250.000 Knoten durchgeführt. Zu Testzwecken wurden auch Läufe mit nur 10.000 Knoten durchgeführt, diese Ergebnisse werden hier jedoch nicht vorgestellt, da sie hauptsächlich genutzt wurden, um Parameter und Funktionsmengen zu testen. Die meisten Läufe wurden mit 100.000 Knoten durchgeführt, da eine Evolution mit einer Knotenbegrenzung von 250.000 bis zu einer Woche dauern konnte. Für diese Läufe war das Ziel, ein Individuum zu evolvieren, welches ein Programm der Tiefe acht schlägt. Bei den Evolutionsläufen mit 100.000 Knoten dagegen war das Ziel, ein Programm der Tiefe sechs zu schlagen, daher ist die durchschnittliche Zeit für eine Evolution vier bis sechs Stunden.

Für alle Läufe waren die Parameter für die Anzahl der Generationen, die Größe der Demes und weitere Parameter unverändert. Diese Parameter sind durch die Testläufe mit 10.000 Knoten evaluiert worden und in der Tabelle 6.4 zusammengefasst. Was einem bei den Parametern sofort auffällt, ist die kleine Population von nur 20 Individuen, was für GP-Läufe ungewöhnlich ist. In Abschnitt 4.4.2 konnte aber schon gezeigt werden, dass diese Poolgröße für die Linear-Graph Struktur ausreichend ist um gute Ergebnisse zu erzeugen. Dies hatte ja auch zur Wahl dieser Struktur innerhalb der State-Space Struktur der Schachindividuen geführt. Ein zweiter Parameter, der auffällig ist, ist die geringe Anzahl von Evaluationen. So werden maximal 4.000 Evaluationen durchgeführt, danach wird die Evolution abgebrochen. Die Evolution kann aber auch abgebrochen werden, wenn ein Individuum eine vorgegebene Güte erreicht. Für die Evolutionsläufe mit 100.000 Knoten war diese der Gewinn einer Partie gegen Programme der Suchtiefe sechs und bei 250.000 Knoten der Gewinn einer Partie gegen Programme der Suchtiefe acht.

Für die Läufe mit 100.000 Knoten wurden die Parameter für den Programmflussabschnitt, siehe auch Kapitel 4.3.1, variiert. Der Programmflussabschnitt bestimmt, ob die Materialbewertung oder die Zugbewertung aufgerufen wird und damit eine Rekursion durchgeführt wird. Vor dem Aufruf der Zugbewertung müssen jedoch die möglichen Züge für die Stellung generiert werden. Neben der Entscheidungsfunkton bestimmt der Programmflussabschnitt noch, welche Züge generiert werden. Hierzu stehen vier Funktionen zur Verfügung, eine Funktion generiert alle legalen Züge für die aktuelle Stellung. Die zweite Funktion generiert aggressive Züge, die dritte defensive Züge und die letzte Funktion ermöglicht dem Individuum einen Nullzug durchzuführen. Um den Einfluss der verschiedenen Funktionen auf die Fähigkeiten des Individuums zur Entwicklung eines Algorithmus zu bestimmen, wurden Evolutionsläufe mit folgenden Funktionskombinationen durchgeführt:

- 1. alle legalen Züge,
- 2. alle legalen und aggressive Züge,
- 3. alle legalen, aggressive und defensive Züge,
- 4. alle legalen, aggressive, defensive und Nullzüge.

Parameter	Wert	Beschreibung
Evaluationen	4000	Maximale Anzahl der Fitnessauswertungen pro Lauf
Population	20	Die Größe der Population für einen Lauf
Crossover	0,2	Wahrscheinlichkeit, dass zwei Individuen an einem Crossover teilnehmen
Mutation	1,0	Wahrscheinlichkeit, dass ein Individuum mutiert wird
Selektion	Turnierselektion 2*2	Turnierselektion aus zwei Zweier-Turnieren
Minimale Individuengröße	20	Anzahl der Knoten, die ein Individuum minimal haben durfte
Maximale Individuengröße	60	Anzahl der Knoten, die ein Individuum maximal haben durfte
Maximale Individuen Tiefe	10	Die maximale Tiefe der Graphen in den einzelnen Modulen

Tabelle 6.4: Diese Tabelle enthält alle Parameter, die für alle Tests der Materialbewertung genutzt wurden.

Die letzteren Funktionsmengen ermöglichen der Evolution Programme zu entwickeln, die immer komplexere Suchbäume erzeugen. Alle Funktionsmengen enthalten die Funktion *alle legalen Züge*, dies ist notwendig, da sonst Programme entstehen würden, die nicht die Möglichkeit hätten den Suchraum ausreichend aufzuspannen. Die Philosophie bei der Auswahl der Funktionsmengen ist es, der Evolution immer weitere Möglichkeiten zu geben, ohne ihr die Möglichkeit der vorherigen Stufe zu nehmen. Für jede dieser Funktionsmengen wurden Läufe mit der 100.000 Knoten Grenze durchgeführt, die Ergebnisse werden im folgenden dargestellt.

Wie man anhand der Fitnessfunktion in Kapitel 5.2.3 sehen kann, war die Evolution darauf ausgelegt, Programme zu entwickeln, die sowohl *weiß* als auch *schwarz* spielen sollten. Jedoch hat sich bei den Testläufen sehr schnell gezeigt, dass die Evolution in der vorgegebenen Zeit keine Individuen entwickeln konnte, die diese Aufgabe mit einer zufrieden stellenden Güte lösen konnten. In den Läufen mit 10.000 Knoten konnte keine Evolution Individuen generieren, die ein Programm der Suchtiefe eins mit beiden Farben schlagen konnte. Obwohl Programme, die nur weiß oder schwarz spielten, in der Lage waren Programme der Suchtiefe drei zu schlagen. Daher wurden die Läufe so gestartet, dass die Individuen nur weiß oder nur schwarz spielten.

Im Anhang im Abschnitt 8.3 sind alle Funktionen aufgelistet, die für das GeneticChess-System entwickelt wurden. Jedoch wurden für die Tests des GeneticChess-Systems nur eingeschränkte Operations-, Terminal- und Branchingmengen benutzt. Es wurden auch Tests mit dem kompletten Satz von Funktionen durchgeführt, jedoch sind die Ergebnise relativ schlecht, näheres dazu später. Diese Funktionsmengen wurden anhand verschiedener Testläufe entwickelt, dabei haben sich diese Funktionsmengen als gut erwiesen. Problematisch ist hier, dass aus Gründen der Kombinatorik keine vollständigen Tests durchgeführt werden konnten, um die optimalen Funktionsmengen zu finden.

Für das Tiefenmodul wurden folgende Funktionssätze verwendet: Die Operationsmenge bestand aus den Funktionen: $+, -, *, /$, ifCurrentDepthIsBigger, ifCurrentDepthIsSmaller, isBiggerThanAlpha, isSmallerThanBeta, isIndKingUnderAttack, isIndKingUnderCover, isIndQueenUnderAttack, isIndQueenUnderCover, isOppoKingUnderAttack, isOppoKingUnderCover, isOppoQueenUnderAttack und isOppoQueenUnderCover.

Die Terminalmenge setzt sich aus den folgenden Funktionen zusammen: numOfExpandedMoves, numOfMoves, numOfKingMoves, numOfQueenMoves, numOfPawnMoves, numOfBishopMoves, numOfKnightMoves, numOfRookMoves, bestMoveValue, worstMoveValue, currentMoveValue, numOfThreatsOfInd, numOfThreatsOfOppo, numOfCoversOfInd, numOfCoversOfOppo, endgameIndex, getBoardSimpleVal, getBoardAdvancedVal, numOfAttacksBefore, numOfDefensedBefore, attackDefenseDiffBefore, numOfAttacksAfter, numOfDefensedAfter, attackDefenseDiffAfter, captureValueOfMove, captureFigureOfMove, looseValueOfMove, maxLooseValueAfterMove, maxLooseValueBeforeMove, maxCaptureValueAfterMove, maxCaptureValueBeforeMove, moveFigureValue und getHistoryOfMove.

Die Branchingmenge besteht aus den Funktionen: R0<, R0>, currentDepthIsBigger, currentDepthIsSmaller, biggerThanAlpha, smallerThanBeta, ifOppoOfQueens, ifOppoTwoBishops, ifIndTwoKnights, ifIndMoreKnights, ifIndMoreBishops, ifIndMove, ifIndTwoBishops und ifIndOfQueens.

Für das Zugsortierungsmodul wurden folgende Funktionssätze verwendet:

Die Operationsmenge besteht aus den mathematischen Operationen $+, -, *, /$.

Die Terminalmenge setzt sich aus den folgenden Funktionen zusammen: umOfAttacksBefore, numOfDefensedBefore, attackDefenseDiffBefore, numOfAttacksAfter, numOfDefensedAfter, attackDefenseDiffAfter, captureValueOfMove, captureFigureOfMove, looseValueOfMove, maxLooseValueAfterMove, maxLooseValueBeforeMove, maxCaptureValueAfterMove, maxCaptureValueBeforeMove, moveFigureValue, getHistoryOfMove, numOfThreatsOfInd, numOfThreatsOfOppo, numOfCoversOfInd, numOfCoversOfOppo, getAllOppoValue, getAllIndValue, endgameIndex, getBoardAdvancedVal, getBoardSimpleVal, getSumOfIndPawns, getSumOfIndKnigths, getSumOfIndBishops, getSumOfIndRooks, getSumOfIndQueens, getSumOfIndKing getSumOfOppoPawns, getSumOfOppoKnigths, getSumOfOppoBishops, getSumOfOppoRooks, getSumOfOppoQueens, getSumOfOppoKing, getNumOfIndPawns, getNumOfIndKnigths, getNumOfIndBishops, getNumOfIndRooks, getNumOfIndQueens, getNumOfIndKing, getNumOfOppoPawns, getNumOfOppoKnigths, getNumOfOppoBishops, getNumOfOppoRooks, getNumOfOppoQueens und getNumOfOppoKing.

Für das Stellungsbewertungsmodul wurden folgenden Funktionssätze verwendet:

Die Operationsmenge besteht aus den mathematischen Operationen $+, -, *, /$.

Die Terminalmenge setzt sich aus den folgenden Funktionen zusammen: getNumOfIndPawns, getNumOfIndKnigths, getNumOfIndBishops, getNumOfIndRooks, getNumOfIndQueens, getNumOfIndKing, getPawnValue, getKnigthValue, getBishopValue, getRookValue, getQueenValue, getKingValue, getNumOfOppoPawns, getNumOfOppoKnigths, getNumOfOppoBishops, getNumOfOppoRooks, getNumOfOppoQueens, getNumOfOppoKing, numOfIndFiguresInOppoCenter, numOfOppoFiguresInOppoCenter, numOfIndFiguresInIndCenter, numOfOppoFiguresInIndCenter, advancedPawnValueInd, advancedKnightValueInd, advancedBishopValueInd, advancedRookValueInd, advancedPawnValueOppo, advancedKnightValueOppo, advancedBishopValueOppo und advancedRookValueOppo.

Die Branchingmenge des Zugsortierungs- und Stellungsbewertungsmoduls bestand aus den Funktionen $R0 <$ und $R0 >$.

6.3.1 Ergebnisse der Evolution

Die größte Anzahl von Evolutionsläufen wurde mit einem Knotenlimit von 100.000 durchgeführt, dies ermöglichte es, eine große Menge von Evolutionsläufen durchzuführen mit den Kapizitäten, die zur Verfügung standen. Es wurden jeweils 100 Evolutionsläufe, in denen das Individuum die weiße oder die schwarze Seite spielte, und acht Evolutionsläufe mit einem Knotenlimit von 250.000 durchgeführt, hiervon konnten nur so wenige Evolutionsläufe durchgeführt werden, da eine Evolution bis zu zwei Wochen auf einem Rechner dauern konnte. Plot 6.27 zeigt die durchschnittliche Fitness der Evolutionsläufe, hier sind deutlich die Unterschiede im Ergebnis der verschiedenen Aufgaben und Parameter zu sehen. So konnte die Evolution der weißen Individuen eine Fitness von 4.800 erreichen, mit gleichen Parametern konnte die Evolutionsläufe für schwarze Individuen nur eine Fitness von 2700 erreichen. Bei einem Knotenlimit von 250.000 erreichten die Individuen eine durchschnittliche Fitness von 6.700. Das Verhalten der Evolution für die weißen Individuen mit einem Knotenlimit von 100.000 zeigt einen typischen Verlauf der Fitnesskurve, mit einem starken Anstieg am Beginn der Evolution und einen geringeren oder fast keinem Anstieg zum Ende der Evolution. Daher kann man davon ausgehen, dass für diese Einstellungen keine großen Verbesserungen durch längere Evolutionszeiträume möglich sind. Der Fitnessverlauf für die schwarzen Individuen zeigt ein anderes Verhalten, hier kann man deutlich erkennen, dass der Anstieg der Fitness gleichmäßig über den gesamten Evolutionszeitraum geschieht. Das deutet darauf hin, dass bei einem längeren Evolutionzeitraum noch bessere Ergebnisse zuerreichen wären. Es zeigt uns aber auch, dass es für die Individuen schwerer ist, die schwarze Seite zu spielen als die weiße Seite. Dies hat sich auch bei den Versuchen gezeigt, bei dem die Individuen beide Seiten während der Evolution spielen mussten. Bei keinem dieser Versuche ist es der Evolution gelungen ein Gegnerprogramm der Tiefe zwei zu schlagen. Neben den Problemen mit der Farbe Schwarz, hatten die Individuen massive Probleme Algorithmen zu entwickeln, die unabhängig von der Farbe sind, für die gezogen werden musste. Wobei hier während der Evolution immer die Spiele gegen Programme der Farbe Weiß ein Problem für die Individuen darstellte. Wir haben keine weitere Analyse durchgeführt, warum es den Individuen schwerfällt Programme für die Farbe Schwarz zu entwicklen.

Die Evolutionsläufe mit dem Knotenlimit von 250.000, die rote Kurve in der Abbildung 6.27, zeigen ein stufigeres Verhalten als die anderen Evolutionsläufe, dies liegt jedoch an der geringen Anzahl der Evolutionsläufe, die durchgeführt wurden. Das Ziel war dabei, ein Individuum mit einer Fitness von 8000 zu erzeugen, hierzu jedoch mehr in Kapitel 6.3.1.1.

Die Tabelle 6.5 zeigt die Fitnessverteilung für die Evolution der weißen und schwarzen Individuen aus Abbildung 6.27 und die Evolution mit maximalem Funktionssatz 6.28. Man erkennt an der Tabelle viel besser, warum die Evolutionsläufe der schwarzen Individuen im Durchschnitt soviel schlechter ausfallen als die der weißen Individuen. Der wichtigste Unterschied der Evolutionsläufe liegt in den Fitnessbereichen von 1000-1999 und ab 6000. Nur 1,9 % der weißen Evolutionsläufe beenden den Lauf im Fitnessbereich von 1000-1999, aber 38,5 % der schwarzen Evolutionsläufe. Dagegen erreichen 68,9 % der weißen Evolutionsläufe eine Fitness von 6000 und mehr, aber nur 26,9 % der schwarzen Evolutionsläufe. Evolutionsläufe der schwarzen Individuen können die gleiche maximale Fitness erreichen wie die der weißen Individuen, jedoch ist die Erfolgswahrscheinlichkeit viel kleiner als die der weißen Individuen, so können mehr als doppelt so viele weiße Individuen diese Fitness erreichen. Für den Fitnessbereich von 1000-1999 beträgt der Unterschied sogar das 20 fache. Sehr viele Evolutionsläufe scheitern schon bei dem Versuch ein Programm der Suchtiefe 2 zu schlagen. Für die weißen Evolutionsläufe gibt es erst ab Programmen der Suchtiefe 3 (Fitness 2000-2999) einen größeren Widerstand, erstaunlich ist hier, dass der Widerstand für beide Fälle ungefähr gleich ist. Für die Evolutionsläufe

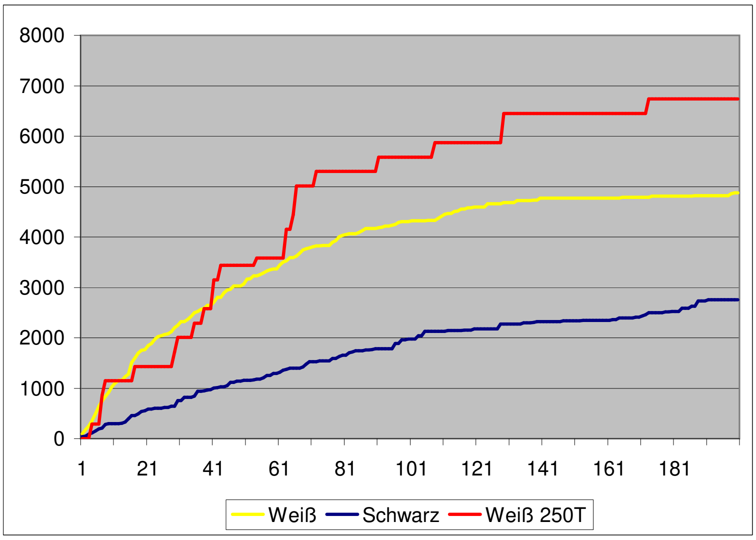

Abbildung 6.27: Der Plot zeigt die durchschnittliche Fitness der Individuen mit einem Knotenlimit von 100.000 für die Seite Weiß und Schwarz, sowie mit einem Knotenlimit von 250.000. Die Daten wurden über der die weißen und schwarzen Individuen wurden über 100 Läufe gemittelt. Die Fitness für Individuen mit einer Knotengrenze von 250000 wurde über 11 Läufe gemittelt.

Fitness	Weiß	Schwarz	Maximaler Funktionssatz
0-999	2,91	8,97	59,09
1.000-1.999	1,94	38,46	22,73
2.000-2.999	16,50	11,54	9,09
3.000-3.999	0,97	8,97	4,55
4.000-4.999	8,74	0,00	4,55
5.000-5.999	0,00	5,13	0,00
6.000-6.100	68,93	26,92	0,00

Tabelle 6.5: Diese Tabelle zeigt, wieviel Prozent der verschiedenen Läufe besser oder gleich gut waren wie die angegebene Fitness.

der schwarzen Individuen scheint der Schritt, einen Algorithmus zu entwickeln, der auf eine Stellung reagiert, schwerer zu sein, als einen Algorithmus zu evolvieren, der agiert. Ist diese Hürde überwunden, können die Evolutionsläufe der schwarzen Individuen ungefähr die gleichen Leistungen erreichen wie für die weißen Individuen. Für die Evolutionsläufe mit dem maximalen Funktionssatz sieht das Ergebnis ganz anders aus, hier erreicht die beste Evolution eine Fitness im Bereich von 4.000-4.999 und 80 % der Evolutionsläufe erreichen keine Fitness, die besser ist als 1999. Durch die geringe Laufzeit der Evolutionsläufe von nur 200 Generationen und der kleinen Demegröße von nur 20 Individuen interpretieren wir das Ergebnis dahingehend, dass die Evolutionsläufe bei dem maximalen Funktionensatz größere Demes und längere Laufzeiten benötigen würde, um zu besseren Ergebnissen zu gelangen, eine weitere Möglichkeit stellen wir in Kapitel 6.3.1.2 vor.

In der Einleitung zu diesem Kapitel wurden die verschiedenen Operations- und Terminalmengen für die Individuen vorgestellt. Die Auswahl ergab sich durch vorbereitende Tests mit kleineren Knotenlimits, um so schneller die Ergebnisse der Evolution zu erhalten. Wir haben aber auch Läufe mit der vollstandigen Funktions- und Terminalmenge und einem Knotenlimit von 100.000 durchgeführt, das Ergebnis ist in der Abbildung 6.28

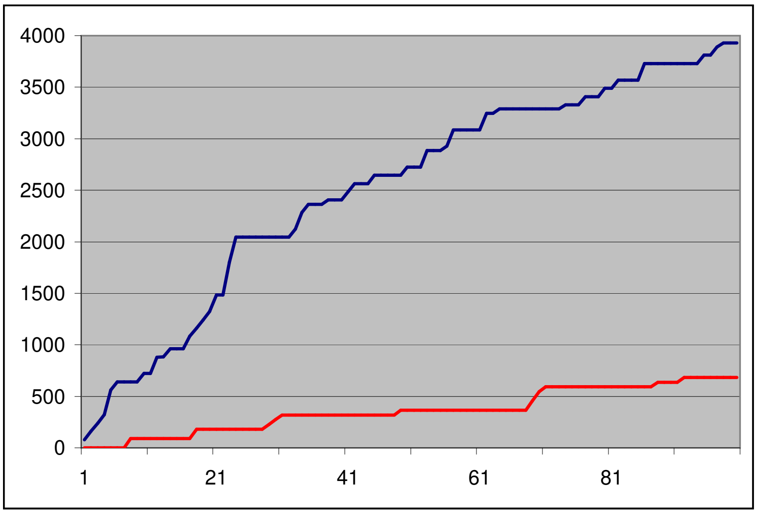

Abbildung 6.28: Der Plot zeigt die durchschnittliche Fitness der Individuen mit der vollständigen Funktions- und Terminalmenge (rote Kurve) und bei gleicher Einstellung mit der eingeschränkten Funktionsmenge (blaue Kurve). Die Daten wurden über 25 Läufe gemittelt.

zusehen. Die Fitness der Läufe mit der eingeschränkten Funktions- und Terminalmenge erreicht im Durchschnitt eine fünf mal höhere Fitness als die Läufe mit der vollständigen Funktionsmenge. Man erkennt für diese Kurve auch nicht den typischen starken Anstieg der Fitness zu Beginn der Evolution. Man kann hier davon ausgehen, dass durch den viel höheren Freiheitsgrad der Evolution eine viel größere Anzahl von Evolutionsläufen und auch mehr Individuen pro Deme notwendig sind um bessere Ergebnisse zu erzielen. Durch den hohen Rechenaufwand für jede Fitnessauswertung war es jedoch nicht möglich dem System die erforderliche Zeit zur Verfügung zu stellen. Im Kapitel 6.3.1.2 haben wir eine Analyse über die Häufigkeit der benutzten Funktionen in den einzelnen Modulen der Individuen durchgeführt. Auf dieser Grundlage könnte man wahrscheinlich eine bessere Auswahl der Funktions- und Terminalmengen durchführen, als sie in den Vortests durchgeführt wurden. Erstaunlich ist aber die geringe Fitness der Individuen, die allein auf den höherdimensionalen Konfigurationsraum der Individuen zurückzuführen ist. Inwieweit der Rückstand bezüglich des Fitnesswertes durch einen längeren Evolutionszeitraum aufzuholen ist, konnte im Zusammenhang mit dieser Arbeit nicht überprüft werden. Durch die extrem langen Fitnessberechungen und den komplexen Aufbau der Individuen sollte man dies besser in einem einfacheren Umfeld überprüfen.

Wie zu Beginn des Kapitels beschrieben, haben wir dem Individuum im Programmflussabschnitt die Möglichkeit gegeben zu bestimmen, welche Züge generiert werden sollen. Hierzu stehen vier Funktionen zur Verfügung, die wir zu den folgenden vier Mengen zusammengefasst haben.

- 1. alle legalen Züge,
- 2. alle legalen Züge und aggressive Züge,
- 3. alle legalen Züge, aggressive Züge und defensive Züge,
- 4. alle legalen Züge, aggressive Züge, defensive Züge und Nullzüge.

Bei Evolutionprozessen mit der Menge 1 wurde im Inidividuum in jedem Programmflussabschnitt die Funktion *legale Züge* verwendet. Wurde die Menge 3 verwendet, konnte die Funktion im Programmflussabschnitt entweder *legale Züge, aggressive Züge* oder *defensive Züge* sein. Dadurch kann das Individuum den Suchraum der Schachzüge freier gestalten und gewisse Aspekte des Spiels stärker betrachten als andere. Durch diese Möglichkeit verliert man natürlich auf der anderen Seite die Gewissheit, dass der beste Zug ermittelt wurde. Dieses Vorgehen ist durch menschliche Spieler motiviert, die auch nicht alle legalen Züge betrachten, sondern gewissen Strategien, oder Vorstellungen folgen um den nächsten Zug zu bestimmen. In der Abbildung 6.29 kann man nun das durchschnittliche Ergebnis von je 25 Evolutionsläufe für jede der vier Möglichkeiten sehen. Die blaue Kurve stellt die Ergebnisse der Menge eins dar, die grüne der Menge zwei, die rote der Menge 3 und die gelbe der Menge 4. Der Plot zeigt eine Zweiteilung der Ergebnisse, so erreichen die Evolutionsläufe mit den Funktionsmengen 2 und 3 und die Evolutionsläufe mit den Funktionsmengen 1 und 4 ähnliche Fitnesswerte. Dies entspricht zum Teil unseren Erwartungen, so sind wir davon ausgegangen, dass eine Evolution mit der Funktionsmenge 4 die im Durchschnitt schlechtesten Ergebnisse liefert, da der Aufbau des Suchbaumes durch die *Nullzüge*-Funktion sehr stark beschnitten werden kann. Wenn die Funktion zu häufig aufgerufen wird, kann es somit zu falschen Bewertungen einer Stellung kommen. Aus den Erfahrungen mit den anderen Systemen haben wir geschlossen, dass die Evolutionsläufe mit der Funktionsmenge 1 im *Durchschnitt* die besten Ergebnisse liefern sollten. Denn sie besteht nur aus der Funktion *legale Züge* und sichert so, dass das Individuum einen vollständigen Suchbaum aufbaut, der den Zugsortierungs- und Stellungsalgorithmen eine gute Bewertung der Situation ermöglicht. Alle anderen Funktionsmengen sollten im Durchschnitt schlechter sein, aber in Einzelfällen sollten bessere Lösungen möglich sein, da die Evolution die zusätzlichen Freiheitsgrade nutzen könnte um die Suche effizienter zu gestalten und somit bessere Lösungen zu finden. Wie man aber am Plot sehen kann, erreichen die Evolutionsläufe mit den Funktionsmengen 2 und 3 die besten Lösungen im Durchschnitt und die Funktionsmengen 1 und 4 scheinen gleich gut zu sein, obwohl der Konfigurationsraum der Individuen gestiegen ist, wie dies auch der Fall ist, wenn die vollständige Funktionsmenge verwendet wird. In dem Fall der vollständigen Funktionsmenge ist die Konfiguration natürlich stärker gewachsen als durch die Verwendung der Funktionsmenge 4 anstelle der Funktionsmenge 1. Man kann jedoch erwarten, dass es zumindestens zu einer leichten Verschlechterung der Ergebnisse im Durchschnitt kommen sollte. Für die Funktionsmengen 2 und 3 kommt es dagegen sogar zu einer Verbesserung der durchschnittlichen Fitness. Wir interpretieren dies als einen Hinweis darauf, dass diese zusätzliche Freiheit für die Individuen wichtig ist, um die Möglichkeiten des Tiefenmoduls besser zu nutzen.

Die Tabelle 6.6 bietet einen anderen Blick auf die Ergebnisse. Hier werden die Ergebnisse der Evolutionsläufe in der letzten Generation als Prozentzahlen für das Erreichen einer Fitnessklasse angegeben. Wenn man diese Ergebnisse für die verschiedenen Programmflussfunktionen vergleicht, sieht man eine starke Ähnlichkeit der Ergebnisse für die Funktionsmengen 1 bis 3. Sie haben alle eine Häufung in der Fitnessklasse ab 2.000, ab 4.000 und ab 6.000. Die Evolutionsläufe für die Funktionsmengen 2 und 3 haben sogar die gleiche Erfolgsrate für die Fitnessklasse 6.000. Die Evolution mit der Funktionsmenge 4 zeigt ein anderes Verhalten, hier zeigen sich nur zwei Häufungspunkte, der eine liegt bei 2.000 und der zweite liegt bei 6.000. Obwohl die durchschnittliche Fitness für die Evolution der Funktionsmenge 1 und 4 fast gleich ist, zeigen sich hier jedoch große Unterschiede, so erreicht keine Evolution die Fitnessklasse 4.000 im Gegensatz zu den anderen Evolutionsläufen. Auch erreichen 15 % mehr Individuen die höchste Fitness gegenüber den Evolutionsläufen mit der Funktionsmenge 1 und ungefähr 9 % weniger als die der Funktionsmengen 2 und 3. Für die Evolutionsläufe mit der Funktionsmenge 4 ist die Fitnessklasse 2.000-2.999 eine Hürde, wenn diese genommen werden kann, erreichen fast alle Evolutionsläufe die höchste Fitnessklasse. Dieses Ergebnis war für uns wichtig bei der Auswahl der Funktionsmengen für die Evolutionsläufe mit einem Knotenlimit von 250.000. Denn Evolutionsläufe, in denen keine guten Individuen erzeugt werden, sind von

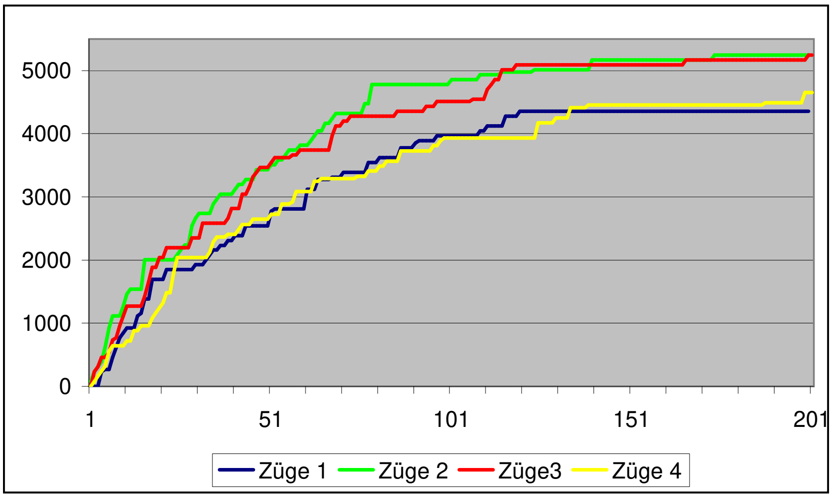

Abbildung 6.29: Der Plot zeigt die durchschnittliche Fitness der Individuen für die verschiedenen Programmflussfunktionen. Die Daten wurden jeweils über 25 Läufe gemittelt.

Fitness	Züge 1	Züge 2	Züge 3	Züge 4
0-999	3,85	0,00	3,85	4,00
1.000-1.999	3,85	0,00	0,00	4,00
2.000-2.999	23,08	15,38	7,69	20,00
3.000-3.999	0,00	0,00	0,00	4,00
4.000-4.999	15,38	7,69	11,54	0,00
5.000-5.999	0,00	0,00	0,00	0,00
6.000-6.100	53,85	76,92	76,92	68,00

Tabelle 6.6: Diese Tabelle zeigt, wieviel Prozent der verschiedenen Läufe besser oder gleich gut waren wie die angegebene Fitness.

der Rechenzeit nicht sehr aufwendig. Die Rechenzeit wird durch die Auswertung der Fitness bestimmt und Spiele gegen spielstarke Gegnerprogramme, also ab Suchtiefe sieben (Fitness 7.000), werden zu rechenzeitaufwendig, um während einer Evolution zu häufig aufgerufen zu werden. Die Hoffnung bestand, dass die Evolutionsläufe entweder das Ziel nicht erreichen und damit wenig Rechenzeit verbrauchen, da die Fitness im Bereich von unter 5.000 liegt, oder das Ziel einer Fitness von über 8.000 erreicht wird.

Zuletzt haben wir den Knotenverbrauch der einzelnen Evolutionsläufe im Durchschnitt betrachtet. Der Plot 6.30 zeigt den Verlauf der Evolutionsläufe mit den vier verschiedenen Funktionsmengen. Obwohl die Kurven den Durchschnittswert über 25 Evolutionsläufe darstellen, kann man deutlich die sprunghaften Veränderungen der Werte während der Evolution erkennen. Beim Vergleich der Daten hat sich gezeigt, dass es eigentlich kein einheitliches Bild für die Entwicklung des Knotenverbrauchs gibt. So gibt es Evolutionsläufe, in denen die besten Individuen über Generationen sehr nahe am Knotenlimit liegen und plötzlich einen starken Einbruch haben oder umgekehrt. Es zeigt sich aber kein einheitliches Bild. Man kann vielleicht eine grundsätzlich Tendenz ableiten, dass die Individuen in den ersten Generationen weniger Knoten verbrauchen und in späteren Generationen mehr Knoten verbrauchen. Jedoch zeigen einzelne Evolutionsläufe teilweise ein anderes Bild, was man sogar im Plot erkennen kann, wenn man sich die Kurve (blau) für die Funktionsmenge 1 betrachtet. Hier sieht man in Generation 175 einen Rückgang von 10.000 Knoten. Erstaunlich ist, dass man die Evolutionsläufe mit den Funktionsmengen 3 und 4 kaum unterscheiden kann, obwohl sie in den Plots der Fitnesswerte

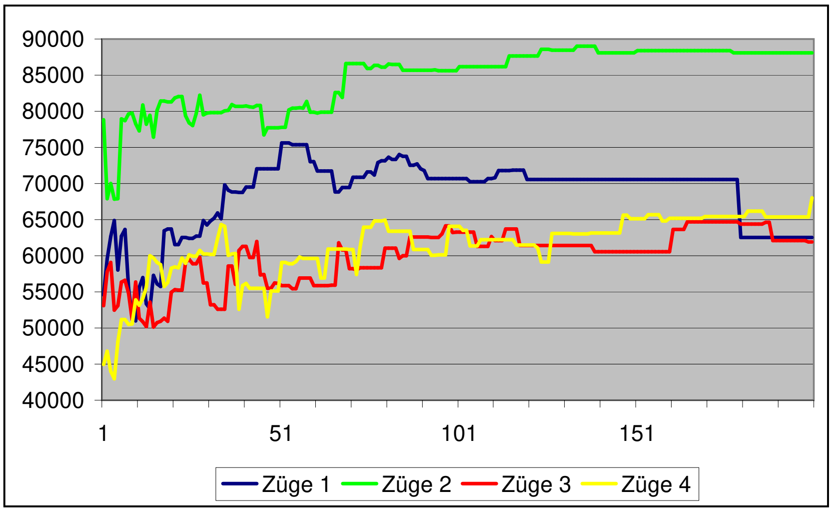

Abbildung 6.30: Der Plot zeigt den durchschnittliche Knotenverbrauch der Individuen für die verschiedenen Programmflussfunktionen. Die Daten wurden jeweils über 25 Läufe gemittelt.

stark unterscheidbar waren. Auch hatten wir erwartet, dass die Evolutionsläufe mit der Funktionsmenge 1 die meisten Knoten während einer Evolution verbrauchen würden, was nicht der Fall ist. Zwar benötigen diese Individuen am Anfang mehr Knoten als die der Funktionsmengen 3 und 4, erreichen aber zum Schluss einen ähnlich hohen Knotenverbrauch wie mit den anderen beiden Funktionsmengen. Den höchsten Knotenverbrauch haben aber die Evolutionsläufe mit der Funktionsmenge 2, also eine Kombination aus *allen legalen* und *aggressiven Zügen*. Eine Erklärung, warum dies so ist, haben wir nicht gefunden, da auch die anderen Funktionsmengen die Funktion *aggressive Züge* beinhalten und es so eigentlich keine Begründung dafür gibt, dass die Funktionsmenge 3 weniger Knoten verbraucht. Die Funktionsmenge 4 hat noch die Funktion *Nullzüge*, was die Anzahl der Knoten eigentlich stark reduzieren kann, dieser Vorteil wurde wahrscheinlich durch häufigere Explorationen des Suchraums in tiefere Ebenen aufgebraucht.

6.3.1.1 Analyse der Individuen mit einem 250.000 Knotenlimit

Nachdem alle Evolutionsläufe beendet waren, stellte sich für uns die Frage, ob es möglich ist Individuen zu evolvieren, die Programme der Suchtiefe 7 oder 8 schlagen können. Um die Schwere dieser Aufgabe zu erkennen, muss man sich verdeutlichen, dass ein guter Alpha-Beta Algorithmus bei einer Suchtiefe von sieben 2 Millionen und bei einer Suchtiefe von acht 13 Millionen Knoten expandiert. Die Individuen mit einem Knotenlimit von 100.000 konnten Programme der Suchtiefe sechs schlagen, gute Programme benötigen hier 320.000 Knoten. Die Individuen, die Programme der Suchtiefe sechs schlagen, benötigen im Durchschnitt 80.859 Knoten, die besten 20 % der Individuen benötigen zwischen 14.821 – 49.884 Knoten. Der Durchschnittswert für die besten 20 % liegt also bei 35.560 Knoten. Das Individuum, welches den kleinsten Suchbaum aufspannt, benötigt nur $4,6$ % der Knoten, den ein Alpha-Beta Algorithmus benötigt. Die besten Individuen des qoopy-Systems benötigten 50 % der Knoten eines guten Schachalgorithmus um die gleiche Spielstärke zu haben, siehe Abschnitt 6.2.3. Im Durchschnitt benutzen die Individuen des GeneticChess-Systems zwar mehr Knoten als die des qoopy-Systems, jedoch sind auch die Ergebnisse um eine Suchtiefe besser als die des qoopy-Systems. Wenn man nun den Durchschnittswert der besten 20% Programme nimmt, benötigten die Individuen ungefähr 10 % der Knoten eines guten Alpha-Beta Algorithmus. Daher kann man davon ausgehen, dass die Individuen 200.000 Knoten benötigen würden um ein Programm der Suchtiefe sieben zu schlagen und eine Millionen Knoten für ein Programm der Suchtiefe acht. Tests mit Evolutionsläufen, die ein Knotenlimit von einer Millionen hatten, hatten aber eine Rechenzeit, die viel zu hoch war. Daher mussten wir uns auf ein Knotenlimit von 250.000 beschränken, selbst bei diesem Knotenlimit benötigte eine Evolution zwischen 2 und 26 Tagen. Die Hoffnung war, dass dies ausreichen würde, mindestens ein Programm der Suchtiefe sieben zu schlagen. In der Abbildung 6.27 zeigt die Durchschnittsfitness dieser Individuen (die rote Kurve), man erkennt ein sehr stufigen Verlauf der Kurve, dies liegt and der geringen Anzahl von Evolutionsläufen. Trotzdem erreichen sie schon im Durchschnitt eine Fitness von 6.700. Jedoch konnten wir wegen des hohen Rechenzeitaufwandes nicht ausreichend Evolutionsläufe durchführen, um einen glatten Verlauf der durchschnittlichen Fitness zu erreichen. Die Fitness der Individuen in der letzten Generation ergibt folgendes Bild:

- Fitness von 4.000 erreicht eine Evolution
- Fitness von 6.000 erreichen sieben Evolutionsläufe
- Fitness von 7.000 erreicht eine Evolution
- Fitness von 8.000 erreichen zwei Evolutionsläufe

Ungefähr 70 % der Evolutionsläufe erreichten das Ziel nicht, ein Individuum mit einer Fitness von mindestens 7.000 zu evolvieren. Diese Individuen sind ungefähr so stark wie die Individuen mit einen Knotenlimit von 100.000, obwohl sie teilweise deutlich mehr Knoten pro Zug aufspannen als die Individuen der anderen Evolutionsläufe. Man sieht auch hier, dass ein gewisser Selektionsdruck ausgeübt werden muss, um die gewünschten Ergebnisse zu erzielen. Drei Evolutionsrozesse erreichten das gesteckte Ziel oder übertrafen es sogar; wie wir oben dargestellt haben, untersucht ein guter Suchalgorithmus bei einer Suchtiefe von acht 13 Millionen Knoten pro Zug. Die Individuen durften aber nur circa 2 % dieser Knoten pro Zug betrachten und konnten ein Programm der Suchtiefe acht dennoch schlagen. Hier stellt sich sofort die Frage: Kann dies überhaupt sein oder gab es hier einen Fehler in der Auswertung?

Man muss das Ergebnis zuerst einmal an mehreren Stellen relativieren. So expandieren die Individuen nur 2 % der Knoten eines guten Algorithmus, jedoch stehen dem Individuum mehr Funktionen zur Verfügung, als sie ein normaler Algorithmus hat. Sie können demnach eine Stellung viel genauer bewerten, als es das Gegnerprogramm kann. Ein weiterer Vorteil liegt in Funktionen, denen es möglich ist Auswirkungen eines Zuges zu betrachten, ohne ihn zu machen. So konnten die folgenden Funktionen Aspekte einer Stellung nach einem Zug betrachten, *numOfAttacksAfter, numOfDefensedAfter, attackDefenseDiffAfter, maxLooseValueAfterMove, maxCaptureValueAfterMove* und *attackDefenseDiffAfter*. Durch eine geschickte Kombination der Funktionen kann so das Tiefen- und Zugsortierungsmodul nicht nur die aktuelle Situation beurteilen, sondern auch den Effekt eines Zuges. Da beide Module auch Elemente der Materialbewertung haben, können sie schon vor der Berechnung des Materialwertes der Stellung eine recht gute Bewertung der zukünfigen Stellung durchführen. Wichtig ist noch zu wissen, dass das Tiefenmodul jeden Pfad im Suchbaum abschließt und somit noch Wissen über die nächste Suchtiefe indirekt an die Suche weitergibt. Fasst man dies zusammen, untersucht der Algorithmus in Wirklichkeit etwas mehr Stellungen als die angegebenen. Dieser Wert ist vom Ausgrad der Blätter im Suchbaum abhängig. Wenn man etwas konservativer davon ausgeht, dass zu den späteren Zeitpunkten im Durchschnitt nur noch 20 legale Züge möglich sind, würden die Programme somit ein Knotenlimit von 21 mal 250.000 also 5.250.000 Knoten haben. Somit hätte man *nur* eine Einsparung von 60% bei einer Suchtiefe von acht. Wir finden

	Ind A					
Suchtiefe	Mat. 1	Knoten 1	Mat. 2	Knoten 2	Mat. 3	Knoten 3
1	0,1823	264.928	0,3941	250.182	0,3941	250.185
2	0,3301	291.874	0,4138	285.881	0,7783	268.028
3	0,1478	250.178	0,3103	250.186	0,3941	250.174
4	0,5222	250.173	0,3793	275.159	0,5123	281.438
5	0,1577	264.078	**-0,0099**	262.701	0,3054	250.185
6	0,3547	250.171	0,6699	261.033	0,3300	291.846
7	0,0493	250.172	**-0.1724**	250.156	0,1084	263.350
8	0,4877	275.167	0,5468	272.909	0,3300	268.045

Tabelle 6.7: Diese Tabelle zeigt die Ergebnisse der Spiele gegen die verschieden starken Gegnerprogramme des Individuums A und die im Durchschnitt verbrauchten Knoten pro Zug.

es aber bemerkenswert, dass es der Evolution möglich ist, durch bessere Möglichkeiten im Tiefen- und Zugsortierungsmodul eine so massive Einsparung in der Anzahl der betrachteten Stellungen zu machen, wie es hier möglich gewesen ist. An dieser Stelle ist das Individuum bei der Bewertung der Situation näher am Vorgehen eines Menschen als das mechanische Vorgehen der bekannten Schachalgorithmen. Denn jeder Mensch überlegt sich schon bei der Auswahl der Züge, die er näher betrachten möchte, welchen Einfluss der Zug haben wird.

Wenn man sich nun die einzelnen Spiele der Individuen gegen die verschiedenen Gegnerprogramme genauer ansieht, erkennt man viel besser als an dem Fitnesswert, wie die Individuen auf die einzelnen Gegnerprogramme reagieren. In den Tabellen 6.7 und 6.8 zeigen wir drei, beziehungsweise zwei Spiele der besten beiden Individuen, dort *Ind A* und *Ind B* genannt. Für das Individuum A wurden zwei Nachbewertungen durchgeführt, also insgesamt dreimal wurden Spiele gegen alle Gegnerprogramme durchgeführt. Dies wurde gemacht, da hier zweimal Spiele gegen Gegnerprogramme der Tiefe fünf und sieben verloren gegangen sind. Die Werte in der Tabelle geben neben den durchschnittlich benutzten Knoten pro Zug auch den Materialwert am Ende des Spiels an. Der Wert ist der reine Materialwert des Spiels ohne Bewertung von Stellungselementen, er bezieht sich immer auf das Individuum und ist auf den maximalen Materialwert ohne die Könige normiert. Das heißt, ein Ergebnis von eins für ein Individuum bedeutet, dass es keine Figuren verloren hat und das Gegnerprogramm nur noch den König besitzt oder das Spiel gewonnen hat, indem es den König geschlagen hat. Ein Bauer hat somit einen Einfluss von $\frac{100}{4060} \approx 0,0246$ auf den Ergebniswert. Ein negativer Wert zeigt an, dass das Individuum einen Materialverlust erlitten hat und der Wert null zeigt einen ausgeglichenen Materialwert an, siehe hierzu auch Kapitel 5.2.3.

In der Tabelle 6.7 fallen zwei Punkte sofort auf, erstens zwei klar verlorene Spiele gegen Programme der Suchtiefe fünf und sieben und zweitens, dass die Anzahl der durchschnittlichen Knoten immer dem Knotenlimit entspricht. Das Knotenlimit von 250.000 wird hier zwar oft deutlich überschritten, jedoch ist dies ein Seiteneffekt der Evaluation des Tiefenmoduls, siehe auch 5.2.2.3. So wird dem Individuum die Möglichkeit gegeben, alle noch nicht bewerteten Stellungen zu bewerten, nachdem das Knotenlimit erreicht wurde, somit wird dem Individuum die Möglichkeit gegeben die Suche nicht zu abrupt zu beenden. Das Individuum nutzt hier eine Möglichkeit des Rahmenalgorithmus um das Knotenlimit zu erhöhen und somit die Rechenkraft zu steigern. Negativ ist aber, dass wir nun nicht genau erkennen können, wie das Tiefenmodul arbeitet, da wir die Suchtiefe nicht exportiert haben.

	Ind B			
Suchtiefe	Material 1	Knoten 1	Material 2	Knoten 2
1	0,157635	210.267	0,05	80.529
2	0,802956	100.480	0,477833	73.132
3	0,0344828	120.448	0,0591133	220.251
4	0,586207	117.491	0,187192	193.247
5	0,172414	104.687	**-0,0492611**	132.642
6	0,8867	172.808	0	182.915
7	0,0246305	40.602	0,231527	203.662
8	0,586207	60.689	0,719212	200.291

Tabelle 6.8: Diese Tabelle zeigt die Ergebnisse der Spiele gegen die verschieden starken Gegnerprogramme des Individuums B und die im Durchschnitt verbrauchten Knoten pro Zug.

Obwohl das Individuum *Ind A* das Knotenlimit immer ausreizt, können wir aber sagen, dass es sinnvoll den Suchbaum beschneiden muss, denn würde das Individuum dies nicht machen, so könnte nur der erste Zug analysiert werden. Diese Analyse wäre aber sehr schlecht, da nur die unteren 2 Ebenen des ersten Zuges ganz analysiert werden könnten. Dieses Individuum nutzt zwar das Tiefenmodul um den Suchraum zu beschneiden, aber es nutzt auch einen Seiteneffekt des Rahmenalgorithmus, um so viel Stellungen wie möglich zu expandieren.

Individuum *Ind B* zeigt ein Verhalten, das von uns erhofft wurde und auch durch die Fitnessfunktion unterstützt wurde, da bei der Selektion die Anzahl der verbrauchten Stellungen mit berücksichtigt wurde (siehe 5.2.3). Es geht sehr differenziert beim Aufbau des Suchraumes vor und stellt sich somit auf die aktuelle Situation ein, siehe Tabelle 6.8. Erstaunlich ist jedoch, dass der Knotenverbrauch nicht einfach mit der Stärke des Gegnerprogramms steigt, sondern sich, wie es die erste Analyse zeigt, sogar umgekehrt verhält. Von dem geringen Knotenverbrauch gegen die Programme der Suchtiefe sieben und acht bei der ersten Analyse darf man auch nicht den Schluss ziehen, dass während des gesamten Spiels so wenig Stellungen pro Zug verwendet wurden. Die Versuche haben gezeigt, dass es bei Individuen, die ein ähnliches Verhalten wie das Individuum *Ind B* haben, sehr große Schwankungen von Stellung zu Stellung gibt. Dass das Verhalten auch nicht typisch ist, zeigt die zweite Analyse, in der bei dem Gegnerprogramm der Suchtiefe sieben bis zu fünf mal mehr Stellungen untersucht wurden. Da die Individuen nicht vollständig deterministisch arbeiten, zeigen sich Veränderungen im Spielverlauf, so dass sich für beide Seiten von einer Entscheidung Vorteile, aber auch Nachteile ergeben können. Hier sollte erwähnt werden, dass auch die Gegnerprogramme nicht ganz deterministisch agieren. Dies passiert aber nur an einer Stelle des Algorithmus, dies ist die Zugsortierung. Für die Zugsortierung wird ein randomisiertes Quicksort genutzt, dadurch ist die Reihenfolge der Züge bei einer gleichen Zugbewertung unterschiedlich, was zu einem anderen Suchbaum und vielleicht auch zu einem anderen Ergebnis führen kann. Da aber die Bewertung der Züge identisch ist, spricht nichts gegen das Vorgehen. Da die Zugbewertung aber auch eine Heuristik ist, können an dieser Stelle Fehler gemacht werden, die einen Einfluss auf das Spiel haben können. Daher kann man davon ausgehen, dass die Gegnerprogramm zwar immer eine gewisse Durchschnittsstärke haben, diese aber in einem gewissen Intervall schwankt. Jedoch sind die Schwankungen der Gegnerprogramme geringer als die einzelner Individuen, wie man es beim Individuum *Ind A* sieht, so verliert das Individuum während der zweiten Analyse die Spiele gegen die Programme der Suchtiefe fünf und sieben deutlich, gewinnt Spiele gegen diese Programme aber auch deutlich bei der Analyse eins und drei. Dies erkennt man auch beim Individuum *Ind B*, bei der zweiten Analyse hier erzielt das Individuum gegen ein Programm der Suchtiefe fünf nur um ein Remis und bei Suchtiefe sechs einen Materialverlust, der zwei Bauern entspricht. Da nur ein Spiel verloren ging, haben wir jedoch auf einen weiteren Analyselauf verzichtet.

Bei beiden Individuen kann man erkennen, dass sie immer schlechter abschneiden, wenn sie gegen Individuen mit einer ungeraden Suchtiefe spielen müssen. Dies ist ein Effekt, der leicht durch das Schachspiel selber und die Struktur der Gegnerprogramme erklärt werden kann. Alle Gegnerprogramme spannen den Suchraum bis zu einer maximalen Tiefe auf, dann wird spätestens die Bewertung durchgeführt. Wenn eine Stellung nicht in Ruhe ist, kann die Bewertung für einen Zug falsch sein, siehe hierzu auch Kapitel 2.2.6.5. Man kann sich das sehr schnell am Beispiel des Figurentausches klar machen. Nehmen wir einen Damentausch an, eine Suchtiefe von zwei und Schwarz ist am Zug. Schwarz macht also einen Zug mit der Dame, diese kann von Weiß wieder mit der Dame geschlagen werden. Nun wurden zwei Halbzüge gemacht und die Bewertung wird durchgeführt. Schwarz hat nun eine Materialbewertung, die um eine Dame schlechter ist als die von Weiß. Würden jetzt aber die nächsten Züge von Schwarz analysiert, könnte festgestellt werden, dass die weiße Dame von Schwarz im nächsten Zug geschlagen wird und somit die Stellung nicht um einen Damenwert schlechter ist. Das Problem der geraden Suchtiefen ist, dass der Algorithmus nicht die mögliche Antwort auf einen Zug des Gegners mit in die Bewertung einbeziehen kann. Dies hat in vielen Situationen keinen Einfluss, jedoch gibt es ausreichend Situationen, in denen die Algorithmen somit eine *suboptimale* Entscheidung treffen. Dies scheinen die Individuen ausnutzen zu können um zu besseren Ergebnissen zu gelangen als gegen ungerade Suchtiefen. Um dies zu verhindern, müsste man Gegnerprogramme entwickeln, die eine Ruhesuche durchführen. An diesem Beispiel sieht man aber auch, wie flexibel Individuen sind, die durch einen Evolutionsprozess entwickelt wurden.

Es wurden keine Spiele gegen Programme der Suchtiefe neun durchgeführt, da diese wahrscheinlich nicht gewonnen werden könnten, so konnte auch nur ein Programm mit einen Knotenlimit von 100.000 gegen ein Programm der Suchtiefe sieben einen geringen Materialvorteil erkämpfen. Der Hauptgrund lag in der langen Rechenzeit, die für diese Analyse benötigt worden wäre. Da die Gegnerprogramme nicht die Güte der bekannten Schachprogramme haben und selbst diese nur selten während eines Spiels eine Suchtiefe größer als zehn betrachten, haben wir auf Spielen gegen Programme der Suchtiefe neun verzichtet.

6.3.1.2 Analyse der Funktionsverteilung

Abschließend wurde die Struktur der Individuen selbst analysiert, dabei handelt es sich um die Analyse des Individuen ROM's und die Verteilung der verschiedenen Funktionen in die einzelnen Module der Individuen. Bei der Analyse handelt es sich um eine statische Analyse der Individuen, die natürlich nur einen Hinweis auf die Wichtigkeit einzelner Funktionen aus den Funktionsmengen zeigt.

Die Tabelle 6.9 zeigt die durchschnittlichen Werte für Funktionen des Stellungsbewertungsmoduls. Im Kapitel 5.2.2.1 wird die Bedeutung dieser Werte genauer beschrieben, die ersten vier Werte beschreiben dabei die Figurwerte und alle anderen Werte beschäftigen sich mit Stellungselementen.

Wenn man sich die Werte für die verschiedenen Evolutionsläufe anschaut, sieht man deutlich, dass sie ähnlich sind, dies stimmt sowohl bei den Figurwerten als auch bei Werten, welche Stellungselemente bewerten. Bei den Figurwerten sieht man drei große Unterschiede zu den Werten, die man in der Literatur findet, die auf einen Artikel von Shannon [107] zurückgehen. Dabei betragen die relativen Werte für die Königin 900, den Turm 500, den Läufer und den Springer 300 und den Bauern 100, siehe auch 2.2.1. So

Wert	Weiß 100.000	Schwarz 100.000	Weiß 250.000
König	2.028,5	1.840,0	1.565,8
Dame	891,3	862,9	934,5
Läufer	377,9	431,9	390,8
Springer	303,1	319,9	343,8
Turm	136,7	113,3	113,0
Bauer	274,9	276,2	256,8
Zentrumsbauer	25,0	26,3	26,6
Doppelbauer	**24,2**	**28,4**	**26,7**
Freibauer	32,1	43,5	55,7
Rückständiger Bauer	**40,9**	**34,3**	**42,1**
Bauernstruktur	**23,5**	**22,3**	**22,4**
Springermobilität	36,8	48,9	40,4
Springerbonus	32,1	39,3	43,6
Springer Bauer	**61,6**	**59,3**	**52,8**
Läuferpaar	24,4	25,7	27,1
Bauer Läufer Feldfarbe	**21,2**	**20,6**	**25,5**
Blockierter Läufer	**42,0**	**37,7**	**24,0**
Läufer Startposition	**24,7**	**23,2**	**33,4**
Turm auf halboffener Linie	44,2	49,3	42,3
Turm auf offener Linie	36,9	29,9	37,9
Turm	39,9	40,9	44,6
Freibauer und Turm	26,2	21,7	27,9
Rochade verpasst	**31,1**	**26,1**	**20,2**
Rochade durchgeführt	27,0	23,9	25,5
König Grundlinie	**18,0**	**17,4**	**22,2**
Promotionschancen	22,9	31,4	14,6

Tabelle 6.9: Diese Tabelle zeigt die durchschnittlichen Werte für Funktionen des Stellungsbewertungsmoduls. Alle *fett* geschriebenen Werte sind Mali, die vom Materialwert abgezogen werden, wenn sie zutreffen. Im Kapitel 5.2.2.1 werden diese Werte genauer beschrieben.

werden bei Standardstellungsbewertungen der König nicht bewertet, dem GeneticChess-System wurde diese Möglichkeit gegeben, auch um zu überprüfen, ob dies überhaupt genutzt wird. Im Histogramm 6.31 kann man erkennen, dass die Funktion *getKingValue*, welche den Wert des Königs ermittelt, für Individuen mit einem Knotenlimit von 250.000 war dies die viertwichtigste Funktion in der Funktionsmenge, nach den Grundrechenarten. Auffällig bei den evolvierten Werten ist aber der geringe Wert für den Turm, er liegt 400 Punkte unter den Werten, die in der Literatur für den Turm angebeben werden. Am erstaunlichsten war für uns jedoch die Bewertung des Bauern, der einen Wert von ca 260 Punkten bekam und damit um 150 Punkte höher bewertet wurden als ein Turm. Dies ist ein Verhältnis, welches in der Literatur nicht zu finden ist. Sehr erstaunlich ist auch der Unterschied von 50-100 Punkten zwischen Springer und Läufer, obwohl in der Literatur beide Figuren annähernd gleich bewertet werden, oder, wenn es einen Unterschied gibt, der Springer immer besser bewertet wird als der Läufer. Die Bewertung muss aber in Verbindung mit den Stellungskriterien gesehen werden, denn für den Springer gibt es drei stellungsbezogene Kriterien, wobei zwei Boni sind und einer ein Malus. Für den Läufer gibt es vier stellungsbezogene Kriterien, jedoch sind hier drei Mali und nur einer ein Bonus. Der Bonus verfällt sofort, wenn nur noch ein Läufer auf dem Brett steht, wodurch dieser Bonus nicht wie andere Boni bei einer Veränderung der Stellung wieder gewährt werden kann. Wenn man die Boni des Springers und die Mali des Läufers den Figuren hinzuzählt, kann man eine Bewertung erhalten, wie wir sie durch die Literatur erwarten würden. Für den Turm kann man einen Wert, wie er in der Literatur zu finden ist nicht einmal dadurch erreichen, indem man alle Boni zum Figurwert addiert. In den evolvierten Individuen kann man dann maximal den Wert eines Bauern erreichen. Der Wert des Turms ist für uns eines der erstaunlichsten Ergebnisse bei diesen Werten.

Die Gegnerprogramme arbeiten mit den bekannten Werten für die Figuren und obwohl wir die Wahl der Figurwerte der Individuen als nicht gut empfinden, können diese Individuen die Programme mit den Werten der Literatur schlagen. Dies war auch das Ziel der Evolution, erstaunlich bleibt trotzdem die Wahl einiger Werte. Wir können jedoch nicht abschließend analysieren, warum diese Werte für die Individuen von Vorteil sind, da sie innerhalb komplexer Programme verrechnet werden und es dem Programm auch gestattet ist diese Werte intern noch zu verändern. Somit sind die hier aufgezählten Werte nicht unbedingt die Werte, die auch bei einer Bewertung innerhalb eines Spiels genutzt werden.

In den folgenden Abbildungen 6.31 bis 6.34 kann man die Verteilung der Funktionen in den einzelnen Modulen für die Evolution der weißen (gelbe Balken) und schwarzen Individuen (blaue Balken) mit einem Knotenlimit von 100.000 sehen und die Evolutionsläufe mit einem Knotenlimit von 250.000 (rote Balken). Hierzu wurden die besten Individuen der einzelnen Evolutionsläufe ausgewählt und die prozentuale Häufigkeit der verwendeten Funktion bezogen auf die Anzahl aller verwendeten Funktionen im Individuum berechnet.

In der Abbildung 6.31 kann man die Verteilung der Terminalfunktionen des Stellungsmoduls, die Häufigkeit der Terminalfunktionen ist nach denen der Evolutionsläufe mit einem Knotenlimit von 250.000 absteigend sortiert. Die Nicht-Terminalfunktionen waren für das Stellungsmodul die Grundrechenarten, diese sind in der Graphik nicht dargestellt, da sie für alle Evolutionsläufe einen Wert zwischen 12 und 18 % erreichten. Damit wären die kleinen Unterschiede bei der Verteilung der Terminalfunktionen für die Individuen kaum zu erkennen gewesen. Auffällig an diesem Histogramm ist die relative Homogenität in der Verteilung der einzelnen Funktionen. So kann man nicht sagen, dass die Individuen die Gruppe von Funktionen der Figurwerte denen der Stellungskriterien bevorzugen. Hier scheint die Kombination dieser Funktionen wichtig zu sein. Diese kann leider durch diese Art der Analyse nicht erkannt werden. Durch die genutzte Individuenstruktur kann aber die Kombination der Funktionen untereinander durch eine statische Analyse des Individuums nicht herausgefunden werden, da sich ja die aktiven Bereiche des Individuums durch den Programmfluss und somit die Eingabe ergeben. Grundsätzlich zeigt uns das Histogramm aber, dass die von uns gewählten Funktionen für das Individuum alle sinnvoll gewesen sind.

Das Histogramm 6.32 zeigt die Verteilung der Funktionen im Zugmodul, hier sieht man ähnlich wie im Stellungsmodul eine relative Homogenität in der Verteilung der Funktionen, sie ist jedoch nicht ganz so stark ausgeprägt wie bei dem Stellungsmodul ist. In der Graphik sind ebenso wie für das Stellungsmodul nur die Terminalfunktionen abgebildet, die Menge der Nicht-Terminalfunktionen wurde nur durch die Grundrechenarten gebildet und sie haben ähnlich wie für das Stellungsmodul Häufigkeiten im Bereich von 15% ergeben. Einen geringen Unterschied zwischen der Verteilung der Funktionen im Stellungsmodul und im Zugmodul gibt es allerdings doch. Durch die Sortierung nach den Funktionsnamen wird dies etwas deutlicher, denn hier kann man schon erkennen, dass besonders von den weißen und schwarzen Individuen gleiche Funktionen gleich häufig in den Individuen genutzt werden. Die Häufigkeiten für die Individuen mit einem Knotenlimit von 250.000 unterscheiden sich für einige der Funktionen deutlicher als die anderen beiden Evolutionsläufe. Dies ist auch beim Stellungsmodul zu beobachten, wenn auch nicht ganz so deutlich. Wir interpretieren diese Ähnlichkeit dahin, dass diese Funktionen

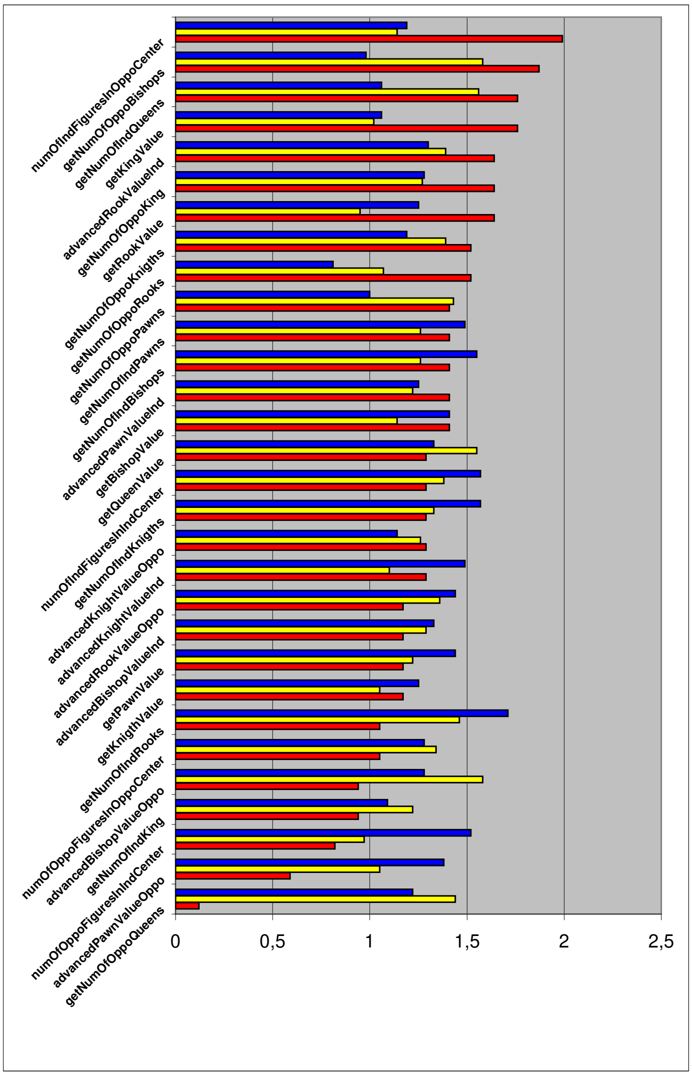

Abbildung 6.31: Der Plot zeigt ein Histogramm aller genutzten Terminalfunktionen für das Stellungsmodul. Die gelben Balken des Diagramms stellen die Evolutionsläufe der weißen Individuen mit einem Knotenlimit von 100.000 dar, die blauen die der schwarzen Individuen und die roten die mit einem Knotenlimit von 250.000, nach diesen sind Balken auch fallend sortiert.

für die Module ähnlich wichtig sind und die Kombination dieser Funktionen wichtiger ist als deren Häufigkeit.

Für das Histogramm des Tiefenmoduls 6.33 ergibt sich ein anderes Bild als für die beiden anderen Module. Hier kann man deutliche Unterschiede zwischen den Häufigkeiten der einzelnen Funktionen sehen. So sind Funktionen, die die Struktur des Suchbaumes beschreiben, wie die Tiefe des Suchbaumes oder die Alpha- und Beta-Grenzen für dieses Modul genauso wichtig, wie Informationen bezüglich des Königs und der Königin. Hierbei werden sowohl die eigenen Figuren betrachtet, als auch die gegnerischen Figuren. Auch werden die Grundrechenarten häufig genutzt, jedoch sind sie hier nicht so wichtig, wie sie es für die anderen Module sind. Ebenfalls wichtig, jedoch nicht genauso wichtig wie die erste Gruppe von Funktionen, sind die Funktionen, die die Anzahl der angegriffenen oder gedeckten Figuren der eigenen Seite oder der gegnerischen Seite berechnen. Die letzte Gruppe bilden die restlichen Funktionen, die für dieses Modul zugelassen wa-

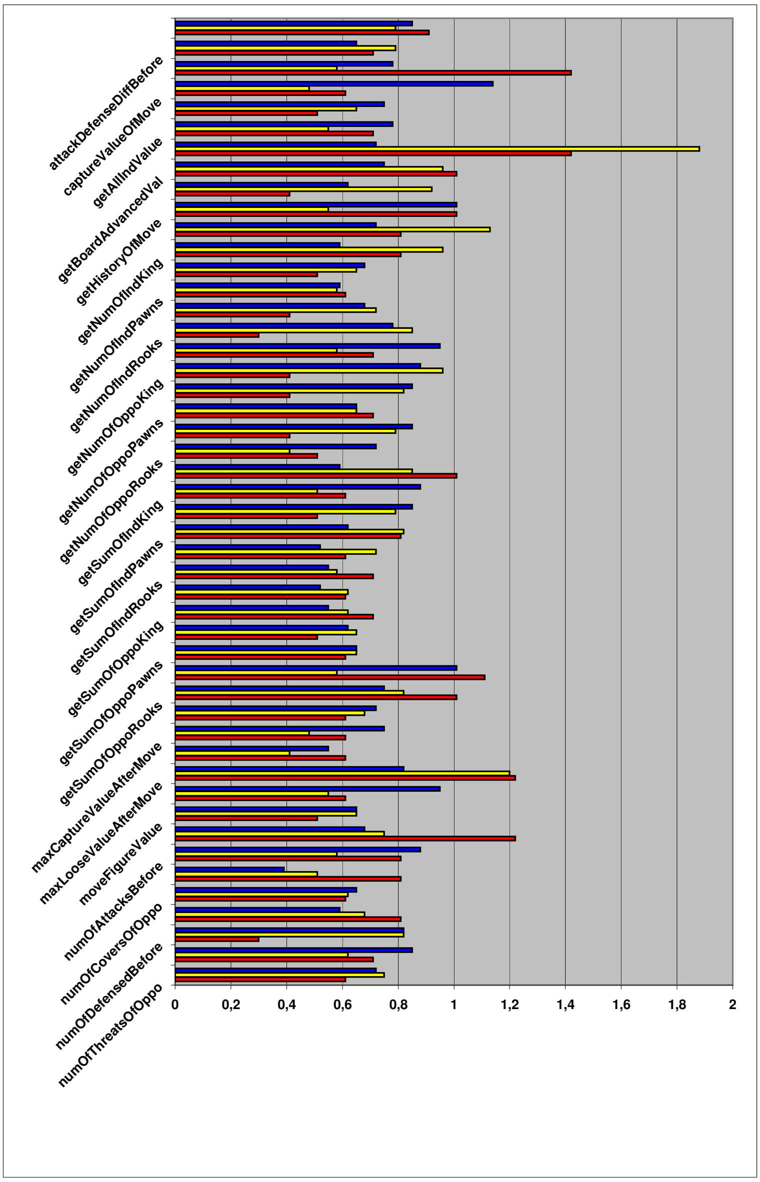

Abbildung 6.32: Der Plot zeigt ein Histogramm aller genutzten Funktionen für das Zugsortierungsmodul. Die gelben Balken des Diagramms stellen die Evolutionsläufe der weißen Individuen mit einem Knotenlimit von 100.000 dar, die blauen die der schwarzen Individuen und die roten die Evolutionsläufe mit einem Knotenlimit von 250.000.

ren, hierunter befinden sich hauptsächlich Funktionen, die eine Bewertung der möglichen Züge durchführen.

Eine deutliche Unterteilung der Häufigkeiten von Funktionen erkennt man auch in der Abbildung 6.34, hier werden die Branchingfunktionen des Tiefenmoduls dargestellt. Die Histogramme für die Stellungs- und Zugmodule für die Branchingfunktionen werden hier nicht gezeigt, da sie eine ähnliche Homogenität aufweisen wie die Histogramme für die Funktionen. Auch für die Branchingfunktionen sind Funktionen wichtiger, die die Struktur der Stellung analysieren, als die Funktionen, die Bewertungen von Zügen vornehmen.

An diesen Beispielen kann man erkennen, dass die verschiedenen Module doch sehr unterschiedlich mit den angebotenen Funktionsmengen umgehen. So gibt es bei dem Tie-

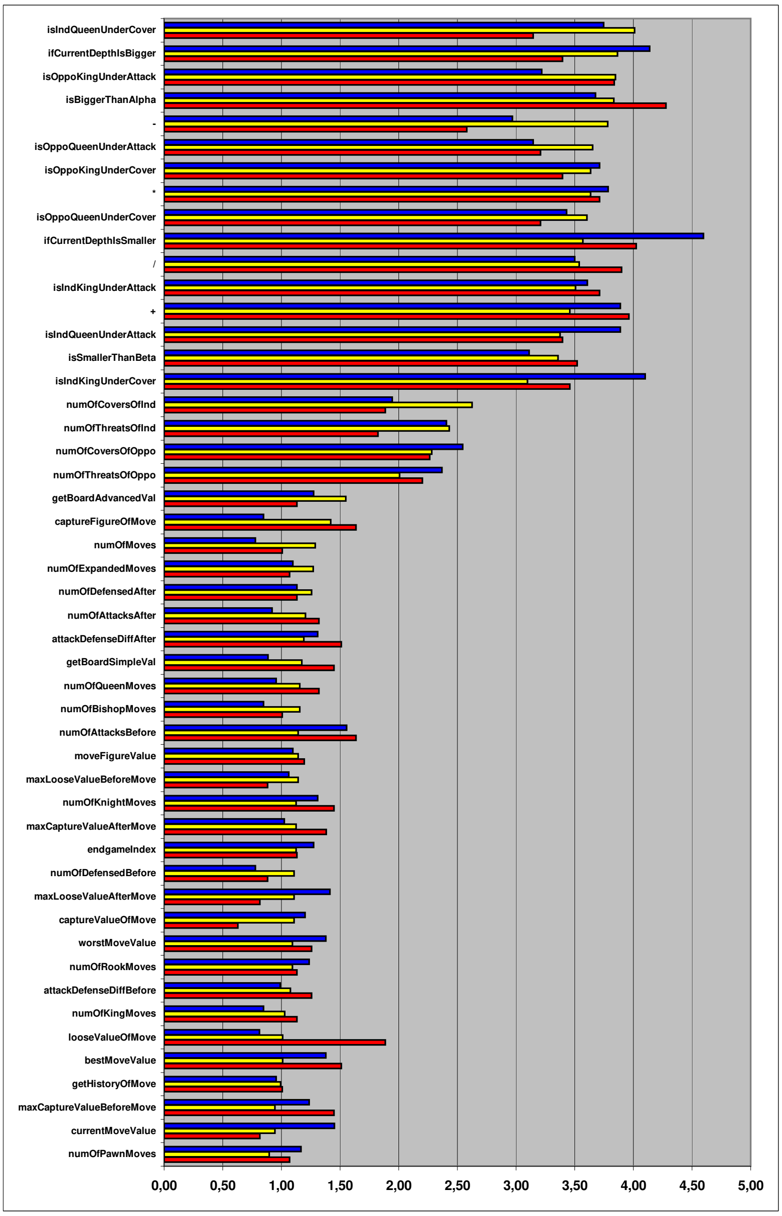

Abbildung 6.33: Der Plot zeigt ein Histogramm aller genutzten Funktionen für das Tiefenmodul. Die gelben Balken des Diagramms stellen die Evolutionsläufe der weißen Individuen mit einem Knotenlimit von 100.000 dar, die blauen die der schwarzen Individuen und die roten die Evolutionsläufe mit einem Knotenlimit von 250.000.

fenmodul eine deutliche Strukturierung der Häufigkeiten, die es für die anderen Module nicht gibt. Eine weitere Analyse der Funktionsmengen für die Schachindividuen wäre sicherlich von Vorteil und könnte sehr wahrscheinlich auch noch das Potential der Programme erhöhen. Problematisch ist jedoch der hohe Zeitaufwand für solche Analysen, da wir nicht davon ausgehen können, dass Evolutionsläufe mit geringerem Knotenlimit und damit schwächeren Individuen die gleichen Funktionsmengen benötigen würden wie stärkere Individuen. Dies kann man schon an den Unterschieden der Funktionshäufigkeiten für die Programme mit einer Fitness von 6.000 und der von 8.000 sehen. An dieser Stelle könnte der Einsatz von Developmental Genetic Programming (DGP) [66] vielleicht ein Ausweg sein, um trotz des hohen Rechenaufwandes zu einer besseren Auswahl der Funktionsmengen zu kommen.

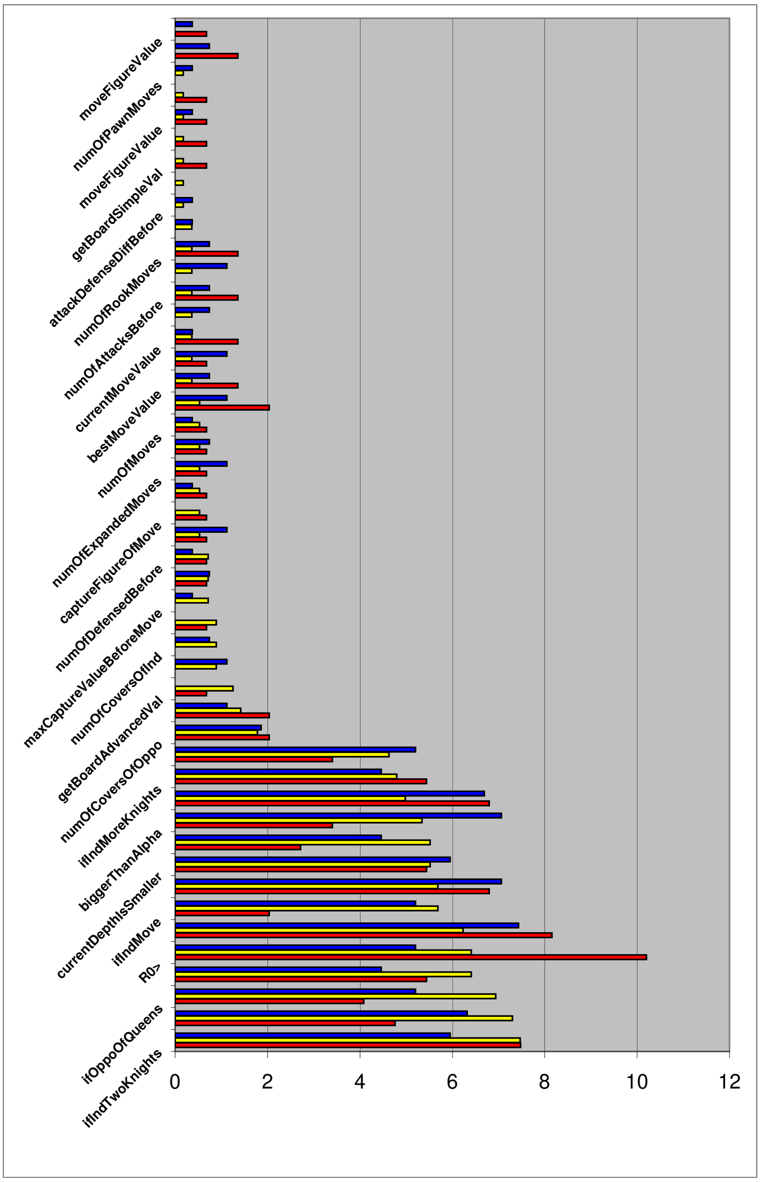

Abbildung 6.34: Der Plot zeigt ein Histogramm aller genutzten Branching-Funktionen für das Tiefenmodul. Die gelben Balken des Diagramms stellen die Evolutionsläufe der weißen Individuen mit einem Knotenlimit von 100.000 dar, die blauen die der schwarzen Individuen und die roten die Evolutionsläufe mit einem Knotenlimit von 250.000.

6.3.2 Fazit GeneticChess

Das GeneticChess-System hat unser Hauptziel, spielstarke Individuen zu evolvieren, die mindestens Programme der Suchtiefe sieben schlagen können, erreicht. Weiterhin konnten wir auch eine Steigerung der Spielstärke durch einen komplexeren Aufbau der Individuen gegenüber dem qoopy-System erreichen. Dies konnte die Steigerung der Spielstärke gegenüber dem qoopy-System erreichen, ohne dass wir den Individuen eine höhere Rechenleistung erlaubten. So hatten beide Systeme ein Limit von 100.000 Knoten, die zur Analyse des nächsten Zuges genutzt werden konnten. Konnten die Individuen des qoopy-Systems maximal ein Programm der Suchtiefe fünf schlagen, so konnten die Individuen des GeneticChess-Systems Programme der Suchtiefe sechs schlagen. Bei einem Knotenlimit von 250.000 ist es dem GeneticChess-System sogar gelungen zwei Individuen zu evolvieren, die Programme der Suchtiefe acht schlagen und ein Individuum, das Programme der Schuchtiefe sieben schlägt.

Da die Stärke von Schachprogrammen eigentlich mittels des Elowertes ermittelt wird, stellt sich für uns noch die Frage, wie stark ein Programm der Suchtiefe fünf oder sechs ist. Hierzu gibt es verschiedene Untersuchungen, die alle ungefähr zum gleichen Ergebnis kommen. Wir haben in der Abbildung 6.35 die Ergebnisse einer Untersuchung von Ernst Heinz dargestellt [47]. Er konnte zeigen, dass zwischen dem dritten und neunten Halbzug die Programme eine Steigerung von 200 Elopunkten erzielten, jenseits des neunten Halbzuges sinkt die Steigerung der Spielstärke auf 70 Elo. Ein Programm der Suchtiefe fünf hat einen Elowert von 1.600, da die Individuen des qoopy-Systems Programme der Suchtiefe fünf schlagen können und gegen Programme der Suchtiefe sechs verlieren liegt ihr Elowert bei ungefähr 1700. Die Individuen des GeneticChess-Systems mit einem Knotenlimit von 10.0000 können Programme der Suchtiefe sechs schlagen und erreichen somit ungefähr einen Elowert von 1.900. Die stärksten Individuen des GeneticChess-Systems können Programme der Suchtiefe acht schlagen und hätten somit einen Elowert von über 2.200. Da wir aber im Abschnitt 6.3.1.1 gesehen haben, dass die Individuen gegen Programme mit einer geraden Suchtiefe weniger Probleme hatten, die Spiele zu gewinnen, als gegen Programme mit einer ungerade Suchtiefe. Was durch die Implementierung der Gegnerprogramme ohne Ruhesuche herrührt. Daher gehen wir davon aus, dass die Individuen ungefähr eine Spielstärke von 2.000 Elo haben. Dies entspricht einem Expertenlevel bei den Schachspielern.

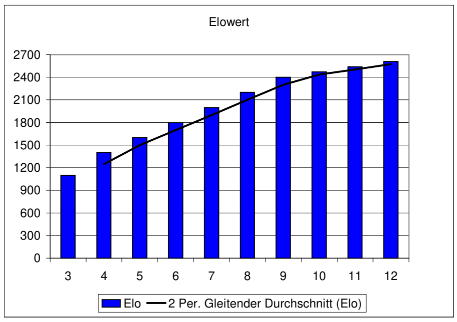

Abbildung 6.35: Das Balkendiagramm zeigt die Entwicklung des Elowertes bezogen auf die Suchtiefe eine Schachprogramms. Die Werte sind aus [47] entnommen.

Wenn man diese Ergebnisse mit anderen Ergebnissen vergleicht, die evolutionäre oder lernende Verfahren verwendet haben, um schachspielende Programme zu entwickeln, so haben wir einen sehr großen Fortschritt erzielt. Als Beispiele seien hier die Arbeiten von Keller et al. in [68], sie konnten nur eine vereinfachte Variante des Schachproblem lösen; oder Hauptman und Sipper [45, 44], die Programme zur Lösung von Endspielproblemen evolvierten; und Kendall und Whitwell [69], die die Parameter für die Materialbewertung evolvierten um so eine Verbesserung eines Alpha-Beta Algorithmus zu erhalten. Als ein anderes Verfahren ist noch das SAL-System von Gherrity zu erwähnen [40]. Nach einer langen Lernphase konnte es acht Remis in 4.200 Spielen gegen Gnuchess erreichen, bei der gewählten Einstellung für Gnuchess, erreichte es ein spielstärke von ungefähr 1.200 Elo. Im Vergleich zu diesen Ergebnissen, haben wir einen großen Sprung bei der Spielstärke und der Komplexität der Programme gemacht. So konnten zwei Individuen Programm der Suchtiefe acht schlagen und haben somit die Spielstärke eines Schachexperten erreicht, dies entspricht einem Elowert von ungefähr 2.200 Punkten.

Neben diesen Leistungen ist uns aber besonders wichtig, wie die Individuen diese Ergebnisse erzielen konnten. Hier ist der geringe Knotenverbrauch der Individuen beim Aufspannen des Suchraumes zu unterstreichen. So konnten die Programme diese Leistungen mit nur 2% des Suchraumes erreichen, den ein guter Alpha-Beta Algorithmus für die gleiche Aufgabe benötigt hätte. Wie wir in Kapitel 6.3.1.1 aber genauer untersucht haben, kommt man durch die komplexen Berechnungen, die wir den einzelnen Modulen ermöglichen, nur zu einer Knotenersparnis von 60%. Dies ist immer noch ein sehr guter Wert und zeigt zudem, dass man durch Erweiterungen der Module eines Standardalgorithmuses ein erhebliches Entwicklungspotential freisetzen kann. Man müsste an dieser Stelle die Algorithmen genauer hinsichtlich der Laufzeit analysieren, ob der erhöhte Aufwand bei der Berechnung der Stellungen den Gewinn in den Einsparungen im Suchraum ausgleicht. Bei Spielen gegen Programme der Suchtiefe sieben und acht, können wir dies auf jedenfall bestätigen, denn hier war die Berechung der Individuen um einen Faktor von bis zu 20 schneller als die der Gegnerprogramme. Da wir bei der Entwicklung dieses Systems, aber auch immer die Möglichkeit der Lösung anderer Probleme mit berücksichtigt haben. Glauben wir, dass dieses System gut bei Problemen eingesetzt werden kann, bei denen die Auswertung eines Punktes im Suchraum sehr aufwendig ist und eine starke Einsparung von Knoten im Suchraum auch mit einer aufwendigeren Berechung zur Auswahl der Punkte im Raum sinnvoll ist.

Der Ansatz Module eines bekannten Suchalgorithmus durch die Genetische Programmierung zu erweitern hat sich bewährt und erstaunliche Ergebnisse hervorgebracht. Auch der Einsatz eines rekursiven Individuums kann als Erfolg gewertet werden. So konnten die Individuen ohne Probleme die Rekursion nutzen, um das gestellte Problem zu lösen.

Kapitel 7

Fazit und Ausblick

Begonnen haben wir die Arbeit mit dem Zitat von Strouhal: „**Der nächste Weltmeister im Schachspiel könnte ein Computer sein, der übernächste einer, der von Computern programmiert wurde.**“ [111]. Dies war kein Ziel der Arbeit, aber ein Leitmotiv, dem man durch diese Arbeit schon ein Stück näher gekommen ist. So kann man jetzt Individuen evolvieren, die auf dem Niveau eines Experten Schach spielen. Diese Entwicklung benötigte bei der klassischen KI circa 25 Jahre, siehe Kapitel 2.1. Hier muss man natürlich eingestehen, dass die langsame Entwicklung der Spielstärke auch der geringen Rechenleistung der Computer geschuldet ist.

Zusammenfassung

Die Arbeit verfolgt die Entwicklung eines Systems zur Evolution von Schachprogrammen, es wurden dazu drei verschiedene Systeme entwickelt, die verschiedene Fähigkeiten, Strukturen und Ergebnisse hatten. Begonnen wurde mit dem *ChessGP* System, dies ist ein sehr komplexes und ambitioniertes System. Es gibt der Evolution sehr große Gestaltungsmöglichkeiten, arbeitet dabei mit Standard GP-Strukturen und einem hierarchischen Aufbau. Die Hierarchisierung wurde für die folgenden Systeme beibehalten, wenn auch in einer einfacheren Variante, jedoch zeigten die Standard GP-Strukturen hier ihre Grenzen, daher wurden für die folgenden Systeme neue GP-Strukturen entwickelt, um komplexere Individuen evolvieren zu können. Dadurch soll der extrem komplexe Aufbau eines Individuums des ChessGP Systems vereinfacht werden. Denn diese Individuen bestehen aus drei hierarchischen Modulen, die zusammen aus 19 GP-Modulen bestehen, siehe Kapitel 5.1.1.2. Die besten Schachindividuen konnten zwar sinnvoll Schach spielen, jedoch erreichten sie nur eine Spielstärke, die einem Alpha-Beta Algorithmus der Suchtiefe zwei entspricht 6.1.4.

Die Linear-Tree und Linear-Graph Struktur wurden entwickelt, um komplexere Lösungswege zu evolvieren und somit auch komplexere Probleme zu lösen. Dies wurde durch die Kombination von Programmfluss und Datenfluss in einer GP-Struktur ermöglicht, siehe Kapitel 4.1 und 4.2. Die Linear-Graph Strukturen zeigten für die Aufgabenstellung die besten Ergebnisse. Im Kapitel 4.4 sind die Ergebnisse dieser Strukturen mit Benchmarkproblemen beschrieben. Neben der gestellten Aufgabe, komplexere Probleme zu lösen, hatten die neuen GP-Strukturen noch einen weiteren Vorteil gegenüber den Standardstrukturen, so können sie Lösungen zu Problemen mit viel kleineren Pools erzeugen und hatten gleichzeitig noch eine kleinere Streuung bei der Güte der Ergebnisse. Dies ist nicht nur für das Schachproblem sinnvoll, sondern auch, wenn man die Evolution von Programmen für industrielle Probleme nutzt. Denn durch die Nutzung kleinerer Demes kann sehr viel Rechenzeit eingespart werden.

Im *qoopy* System wurde die Linear-Graph Struktur zum ersten Mal angewendet, die Individuen wiesen gleichzeitig einen viel einfacheren Aufbau auf. Es zeigte gegenüber dem ChessGP-System eine extreme Steigerung in der Spielstärke der Individuen. Diese rührt hauptsächlich durch den einfacheren Aufbau der Individuen und die Konzentration auf die Verbesserung eines Alpha-Beta Algorithmus her. So konnten die Individuen aus dem qoopy-System bei gleicher Spielstärke im Vergleich zu einem guten Alpha-Beta Algorithmus 50% des Suchraums einsparen. Die besten Individuen erreichten eine Spielstärke, die besser ist als die eines Schachalgorithmus der Suchtiefe fünf. Das Ergebnis wurde dabei hauptsächlich durch das gute Zugsortierungsmodul der Individuen erzielt, siehe Kapitel 6.2.3.

Im *GeneticChess* wurde die Aufteilung der Individuen vom qoopy-System übernommen, jedoch wurde ihnen eine größere Freiheit bei der Lösung ihrer Aufgabe zugestanden. Dies kann man anhand der neuen *State-Space* Struktur erkennen, siehe Kapitel 4.3. Dabei handelt es sich im Aufbau und in der Ausführung der Struktur um einen rekursiven Suchalgorithmus. Anders als im qoopy-System bestehen alle drei Module der Individuen aus GP Programmen. Sie haben durch die Funktions- und Terminalmengen eine viel größere Freiheit bei der Evolution von Programmen als die Individuen des qoopy-Systems. Hierdurch konnte noch einmal eine deutlichere Steigerung der Spielstärke der Individuen erreicht werden. Bei gleicher Einschränkung des Suchraumes, wie beim qoopy-System, konnten die Individuen Programme mit einer Suchtiefe von sechs schlagen, bei einer Vergrößerung des Suchraumes konnten sogar Individuen evolviert werden, die Programme der Suchtiefe acht schlagen konnten. Dies entspricht in etwa einem Elowert von 2200, dies ist die Spielstärke eines Schachexperten. Gleichzeitig konnten die Individuen auch eine deutliche Verringerung der expandierten Knoten erzielen. So konnte man bei einer genaueren Analyse sehen, dass 60% der Knoten des Suchraumes eingespart werden, siehe Kapitel 6.3.2 und 6.3.1.1.

Ausblick

In dem Vorgehen, einen Suchalgorithmus mit Hilfe von GP-Modulen zu verbessern und zu optimieren, wie man es im qoopy- und noch besser im GeneticChess-System sehen kann, kann ein erhebliches Potential bei der Evolution von Programmen freigesetzt werden. Dadurch, dass man dem GP-System bekannte und notwendige Funktionen zur Lösung eines Suchproblems zur Verfügung stellt, können diese durch die Evolution auf sinnvollen Kombination getestet werden. Dadurch können sehr schnell und effizient für neue Problemstellungen maßgeschneiderte Algorithmen entwickelt werden. Denn oft sind sinnvolle Funktionen für Lösungen von Teilproblemen bekannt, jedoch können von den Entwicklern nicht alle Kombinationen ausgetestet werden, da man entweder nicht die Zeit hat oder der Zielalgorithmus nicht zu spezialisiert sein darf, da sonst die Entwicklungskosten für mehrere Algorithmen zu leisten wären. Die Möglichkeit spezialisierte Algorithmen für beliebige Probleme zu erstellen, könnte durch eine Weiterentwicklung des GeneticChess-Systems geleistet werden.

Im GeneticChess wurde dieses Vorgehen genutzt, um zu den guten Ergebnissen zu kommen, gleichzeitig zeigte sich hier auch ein Problem, denn wenn es zu große Funktions- und Terminalmengen gibt, war die Leistung des Systems schlecht, dies kann man im Kapitel 6.3.1 deutlich sehen. Um nun aber zu einem System zu kommen, das automatisch die wichtigen Funktionen herausfiltert, muss man hier eine Analyse der benötigten Funktionen durchführen. Dies kann durch eine Häufigkeitsanalyse der Funktionen in den besten Programmen erreicht werden, wie es hier nachträglich durchgeführt wurde, siehe Kapitel 6.3.1.2. Dieses Vorgehen sollte dann aber während der Evolution durchgeführt werden, um eine gute Teilmenge aus den Funktions- und Terminalmengen herauszufiltern. Eine weitere Möglichkeit das Problem der Funktionsmengen zu lösen wäre der Einsatz von Developmental Genetic Programming (DGP), wie sie Robert Keller in den Artikeln [66] und [67] vorgestellt hat. Hierbei werden die Funktionsmengen während einer Koevolution optimiert, um so eine Vereinfachung der Evolution der Individuen zu schaffen. Eine Problematik bei beiden Verfahren ist der zusätzliche Zeitaufwand, der durch die Auswahl der richtigen Funktionsmenge entsteht.

Ein Ausweg aus der Problematik der Rechenzeit, sowohl für die Koevolution als auch für die Evolution besserer Individuen, ist eine Parallelisierung des Systems. Um den nächsten Schritt in der Evolution von Schachindividuen durchzuführen ist eine Parallelisierung unumgänglich, da schon die Evolution der besten Individuen bis zu 26 Tage auf einem Rechner mit 2,5 GHz gedauert hat. Das Beste wäre eine Kombination der Verfahren, die für das ChessGP und qoopy-System verwendet wurden. Es sollten nur einzelne Spiele und später in der Evolution nur Teile eines Spiels an die Clients im Internet verteilt werden. Durch dieses Vorgehen hat man weiterhin die Kontrolle über die Evolution und kann gleichzeitig sehr viel Rechenzeit zur Verfügung gestellt bekommen. Ein weiterer Punkt wäre die Evolution paralleler Schachindividuen, dadurch könnten die Individuen parallele Hardware nutzen und so noch eine Verbesserung der Spielstärke erreichen. In der Dissertation von Herrn Niehaus wurden schon parallele Individuen evolviert, so dass die Entwicklung solcher Schachprogramme möglich wäre, siehe hierzu [85] Kapitel 11.

Ein weiterer Punkt zur Verbesserung der Spielstärke ist die Verwendung von Datenbanken, hier gäbe es die Möglichkeit Eröffnungs- und Endspieldatenbanken mit einem evolvierten Individuum zu nutzen. Jedoch gäbe es dann nur ein Nebeneinander zweier Verfahren, sinnvoller wäre es hier, dem Individuum die Möglichkeit zu geben mit speziellen Funktionen direkt auf die Datenbanken zuzugreifen um so die Suche zu unterstützen. Der Einsatz von Datenbanken ermöglicht Schachprogrammen normalerweise eine Steigerung der Spielstärke von 200 bis 300 Elopunkten. Wenn man dies auch für die evolvierten Individuen zugrunde legt und gleichzeitig noch eine Steigerung der Suchtiefe auf zwölf erreicht, müssten die Individuen ungefähr einen Elowert von 2.800-2.900 erreichen. Als Deep Blue Kasparov schlug, hatte dieser einen Elowert von 2.775, nach dem Rating der FIDE hatte Garry Kasparov im Oktober 2005 einen Elowert von 2.812. Somit könnten evolvierte Individuen mit diesen Fähigkeiten der neue Weltmeister werden und sie wären von einem Computer programmiert worden. Somit hätte man eine neue Qualität im Kampf zwischen Mensch und Maschine erreicht.

Kapitel 8

Anhang

Die GP-System verwendet eine Vielzahl von Funktionen, die schachspezifisches Wissen verwenden und auch nur für schachspielende Individuen sinnvoll sind. Alle diese Funktionen haben neben den individuelle Besonderheiten, die in den folgenden Abschnitten erklärt werden.

8.1 Funktionen ChessGP

8.1.1 Funktionen der Materialbewertungsebene

Die Funktionen für die Materialbewertungsebene müssen es dem Individuum erlauben, das Schachbrett als Eingabe zu nutzen und zusammen eine Bewertung für das Brett zu berechnen. Hierzu verwendet die Ebene einen Datenstruktur die aus einem dreier-Tupel besteht, dass aus Feld-, Figur- und Zahlenelementen zusammengesetzt ist, siehe hierzu Abschnitt 5.1.1.6.

Operationen zur Steuerung des Programmflusses:

boardLoop Die Funktion wiederholt die folgenden n Operationen für alle Felder des Schachbrettes. Der Wert n wird aus dem Akkumulator Register gelesen. Der Wert n wird immer modulo 100 gerechnet, somit ist die Obergrenze der Schleifengröße 100 Instruktionen. Eine weitere Besonderheit der Funktion ist, dass weitere *Loop* Funktionen innerhalb der Schleife ignoriert werden, somit werden eine Schachtelung von Schleifen und damit verbundene Laufzeitprobleme vermieden.

rowLoop Diese Funktion wiederholt die folgenden n Operationen für alle Felder einer Reihe des Schachbrettes. Ansonsten analog zu boardLoop.

lineLoop Diese Funktion wiederholt die folgenden n Operationen für alle Felder einer Zeile des Schachbrettes. Ansonsten analog zu boardLoop.

pieceLoop Wiederholt die folgenden n Operationen für alle Felder, die von der Figur erreicht werden können, die durch den Akkumulator bestimmt wird.

ifThen Die Methode ist eine Sprungfunktion wie sie für lineare Strukturen verwendet wird. Wenn das Zahlenelement der Datenstruktur im Akkumulator größer als Null ist, werden die n nächsten Instruktionen übersprungen. Sprünge sind immer in Richtung der Ausführungsreihenfolge.

Operationen Ermittlung von Figurwerten:

whichPiece Gibt den Typ der Figur zurück, die im Figurelement der Akku Datenstruktur gespeichert ist.

pieceValue Liefert den Figurwert der Figur, auf die im Akku verwiesen wird. Weiße Figuren bekommen dabei positive Werte und schwarze negative Werte. Die Absolutbeträge sind gleich: Bauer = 10, Springer und Läufer = 30, Türme = 50, Dame = 90 und König = 1000.

attackedByValue Ermittelt den Wert der Figur, die das Feld angreift, auf das der Akku verweist. Greifen mehrere Figuren das Feld an, wird der kleinste Wert genommen.

attackedByNumber Ermittelt die Anzahl der Figur, die das Feld angreift, auf das der Akku verweist. Greifen mehrere Figuren das Feld an, wird der kleinste Wert genommen.

legalMovesNumber Berechnet die Anzahl der legalen Züge, die die Figur durchführen kann, die auf dem im Akku gespeicherten Feld steht.

Operationen zum Verschieben des aktuellen Feldes:

rowShift Die Methode verschiebt das Feld, das im Akku gespeichert ist um eine variable Anzahl von Reihen. Dazu wird das Zahlenelement des Akku-Register verwendet. Der Wert wird immer modulo 8 gerechnet, so dass ein Feld maximal um acht Reihen verschoben werden kann. Ist der Wert positiv wird in die Richtung der schwarzen Grundlinie verschoben, bei negativen Werten in Richtung der weißen Grundlinie. Analog die Funktion **lineShift**.

northWestShift Die Funktion verschiebt das Feld auf der Diagonale, die von oben rechts nach unten links verläuft. Die Randbedingungen sind analog zur rowShift Funktion. Analog die Funktion **northEastShift**.

Lade- und Speicheroperationen:

loadAll Die Methode kopiert den gesamte Inhalt der Datenstruktur eines Registers, dessen Adresse übergeben wird, in den Akku.

loadPiece Die Methode kopiert den Inhalt der Figur-Komponente der Datenstruktur eines Registers, dessen Adresse übergeben wird, in den Akku.

loadValue Die Methode kopiert den Inhalt der Wert-Komponente der Datenstruktur eines Registers, dessen Adresse übergeben wird, in den Akku.

loadLine Die Methode kopiert den Zeilenposition der Feld-Komponente der Datenstruktur eines Registers, dessen Adresse übergeben wird, in den Akku. Dazu wird für die Feldposition im Akku die Zeilenposition berechnet und damit Zeilenposition der Feld-Komponente im Register verändert.

loadRow Die Methode kopiert den Reihenposition der Feld-Komponente der Datenstruktur eines Registers, dessen Adresse übergeben wird, in den Akku. Dazu wird für die Feldposition im Akku die Reihenposition berechnet und damit Reihenposition der Feld-Komponente im Register verändert.

storeAll Die Funktion kopiert den gesamten Inhalt des Akkus in einem Register, dessen Adresse übergeben wird.

storePiece Die Funktion kopiert den Inhalt der Figur-Komponente des Akkus in einem Register, dessen Adresse übergeben wird.

storeValue Die Funktion kopiert den Inhalt der Wert-Komponente des Akkus in einem Register, dessen Adresse übergeben wird.

storeLine Die Funktion kopiert den Zeilenwert der Feld-Komponente des Akkus in einem Register, dessen Adresse übergeben wird. Analog die Funktion **storeRow**.

Operationen zum Setzen von Konstanten:

setAll Die Methode kopiert den gesamte Inhalt einer Konstante in den Akku.

setPiece Die Funktion kopiert den Inhalt der Figur-Komponente einer Konstante in den Akku.

setValue Die Funktion kopiert den Inhalt der Wert-Komponente einer Konstante in den Akku.

setLine Die Funktion kopiert den Zeilenwert der Feld-Komponente einer Konstante in den Akku. Analog die Funktion **setRow**.

swap Tauscht die Werte eines Registers mit denen des Akkus.

Arithmetische Operationen

Zuzüglich zu den schachspezifischen Operationen, verwendet die Materialbewertungsebenen noch die folgenden Arithmetische Operationen.

add Addition des Zahlenwertes der Datenstruktur, die Übergeben wird mit dem Akku. Der Wert kann ein Registerwert, eine Konstante oder das Ergebnis einer Operation sein. Analog die Funktionen **sub, mul, div**.

absolute Ersetzt den Zahlenwert der Datenstruktur im Akku durch den Betrag des Wertes.

8.1.2 Funktionen der Planungsebene

Die Aufgabe der Planungsebene ist es zu einer gegebenen Stellung eine Zugliste zu ermitteln. Mit diesen Zuglisten wird das Individuum, dann in der Variantenebene den Suchbaum aufspannen. Dazu wird eine komplizierte Datenstruktur verwendet, die aus einem Zahlenwert, einer Suchmaske für einen Zug und einer Liste von Zügen besteht, siehe hierzu Abschnitt 5.1.1.6. Zu der Datenstruktur ist noch hinzuzufügen, dass es für jede Komponente der Datenstruktur ein spezielles Element gibt, das als Wildcard fungiert. Dieses Ersetzungssymbol ($*$), bezeichnet einen beliebigen Inhalt und somit können ganze Mengen von Züge beschreiben werden. Dies ermöglicht aber auch verschiedene Funktionen Zuglisten zu Filtern und Züge zu vergleichen.

Operationen zur Steuerung des Programmflusses

b_ifThen Die Operation realisiert eine Verzweigung im Programm. Ist der Zahlwert im Akku größer 0 wird der rechte von zwei Pfaden ausgewertet, sonst der linke.

b_callAdfA Diese Funktion ruft mit der aktuellen Brettstellung ein Programm der Materialbewertungsebenen auf, hierbei wird über die Zahlkomponente des Akkumulators das Programm bestimmt.

Arithmetische Operationen

b_add Addition einer Zahl zur Zahlkomponente im Akku. Das Ergebnis wird in ein Register geschrieben, welches durch den Knoten der ausgewertet wird angegeben wird. Analog die Funktionen **b_sub, b_mul, b_div**.

Mengen Operationen auf den Zuglisten

b_getAllMoves Schreibt alle legalen Züge für die aktuelle Stellung in die Zugliste eines Registers.

b_unionize Die Methode vereinigt die Zugliste des Akkus mit der eines übergebenen Registers. Das Ergebnis ersetzt die Liste des Akkumulators.

b_difference Differenz der Zugliste des Akkus und eines Registers. Das Ergebnis ersetzt die Liste des Akkumulators.

b_sortByValue Die Zugliste des Akkus wird gemäß des Zahlenwerte absteigend sortiert.

b_cardinality Die Länge der Zugliste des Akkus wird in das Zahlenfeld des Akkus geschrieben.

b_subset Die Funktion filtert die Teilmenge der Züge aus dem Akku, die durch die Suchmaske eines anderen Attributs angegeben wird. Die neue Liste ersetzt die Zugliste des Akkus.

b_subsetByBorder Die Methode arbeitet analog zur Funktion b_subset, nur das der Zahlenwert des übergebenen Attributs nun als untere Schranke für die Menge genutzt wird.

b_number Anzahl der Züge in der Zugliste des Akkus, die der Suchmaske des übergebenen Attributes entsprechen. Der Wert wird in die Zahlenkomponente des Akkus geschrieben.

b_numberByBorder Die Methode arbeitet analog zur Funktion b_number, nur das der Zahlenwert des übergebenen Attributs nun als untere Schranke für Operation genutzt wird.

b_set2Mask Die Funktion sucht eine minimale Suchmaske zu einer Zugliste. Minimal bedeutet, dass so wenig Wildcards wie möglich benutzt werden.

b_incrementIfEqual Die Funktion erhöht den Zahlenwert im Akku um eins, wenn die Zugliste des Akkus mit der des übergebenen Registers übereinstimmt.

b_setAllValues Setzt den Zahlenwert jedes Zuges der Zugliste eines Registers auf den Wert, der in der Zahlenkomponente des Registers steht.

b_increaseAlsValues Erhöht das Zahlattribut jedes Zuges der Zugliste eines Registers, um den Wert, der in der Zahlenkomponente des Registers steht.

b_storeMoveValue Speichert die Zahl aus der Suchmaske des Akkus in dem i-ten Zug der Zugliste des Akkus, wobei i dem Zahlattribut des Akkus entspricht.

b_loadMedianMoveValue Berechnet den Median der in der Zugliste vorkommenden Züge und speichert den Wert im Akku.

b_loadMinMoveValue Speichert den niedrigsten Wert der Zugliste im Akku.

b_loadMaxMoveValue Speichert den höchsten Wert der Zugliste im Akku.

b_loadMoveValue Speichert der Wert des i-ten Zuges der Zugliste im Akku. Der Wert i wird durch den Akku bestimmt.

Lade- und Speicher Operationen

b_loadValue Lädt das Zahlattribut eines Registers in das Zahlattribut des Akkus.

b_storeValue Speichert das Zahlattribut des Akkus in das eines Registers.

b_loadValueIndirect Das Zahlenattribut des Registers wird im Akku gespeichert, dabei wird die Adresse des Registers übergeben (indirekte Adressierung).

b_storeValueIndirect Speichert das Zahlattribut des Akkus in das eines Registers indirekt.

b_loadSearchMask Diese Operation lädt eine Suchmaske aus einem Register und speichert sie im Akku.

b_storeSearchMask Speichert die Suchmaske des Akkus in der eines Registers.

b_loadSearchmaskIndirect Wie b_loadSearchMask nur mit indirekter Adressierung.

b_storeSearchmaskIndirect Wie b_storeSearchMask nur mit indirekter Adressierung.

b_storeMoveList, b_loadAll Analog zu b_loadValue.

b_loadMoveList, b_storeAll Analog zu b_storeValue.

b_storeMoveListIndirect, b_storeAllIndirect Analog zu b_storeValueIndirect.

b_loadMoveListIndirect, b_loadAllIndirect Analog zu b_loadValueIndirect.

b_swap Tauscht die Inhalte von Akku und einem Register aus.

Operationen zum setzen von Konstanten und sonstige

setAll2WildCard Diese Operation setzt bis auf das Zahlenattribut der Suchmaske alle Elemente der Suchmaske eines Registers auf Wildcardwerte.

setSourceRow2WildCard Analog zu setAll2WildCard berührt aber nur das Row-Attribut der Datenstruktur.

setSourceLine2WildCard Analog zu setAll2WildCard berührt aber nur das Line-Attribut der Datenstruktur.

setDestinationRow2WildCard Analog zu setAll2WildCard berührt aber nur das DestinationRow-Attribut der Datenstruktur.

setDestinationLine2WildCard Analog zu setAll2WildCard berührt aber nur das DestinationLine-Attribut der Datenstruktur.

setDirection2WildCard Analog zu setAll2WildCard berührt aber nur das Direction-Attribut der Datenstruktur.

setPiece2WildCard, setHitPiece2WildCard Analog zu setAll2WildCard berührt aber nur das Piece-, HitPiece-Attribut der Datenstruktur.

setColour2WildCard Analog zu setAll2WildCard berührt aber nur das Colour-Attribut der Datenstruktur.

setSourceRow Diese Operation setzt das Reihen-Attribut des Startfeldes in der Suchmaske eines Registers auf den übergebenen Wert.

setSourceLine, setDestinationRow, setDestinationLine, setDirection, setPiece, setHitPiece, setValue, setColour, setSearchMaskValue Analog zu setSourceRow verändert das Attribut der Datenstruktur.

copyPiece2HitPiece Diese Funktion kopiert innerhalb der Suchmaske des Akkus den Figurtyp in das Typ-Feld der geschlagenen Figur.

copyHitPiece2Piece Analog zu copyPiece2HitPiece.

8.1.3 Funktionen der Variantenebene

Operationen zur Steuerung des Programmablaufs

callAdfA Ruft das Programm der Matreialbewertungsebene mit dem Index auf, der in dem Akkumulator gespeichert ist.

ifThen Vorwärtsgerichteter Sprung.

exit Stoppt die Ausführung des Entscheidungs-Moduls und startet das Blatt-Modul. Diese Operation wird im Fall der Wurzel ignoriert.

cut Stoppt die Ausführung des Entscheidungs-Moduls und startet das Blatt-Modul. Es schneidet die weiteren Züge des Elternknotens ab. Diese Operation wird im Fall der Wurzel ignoriert.

Arithmetische Operationen

add-, sub-, mult-, divConst Die Methode verknüpfen eine Konstante mit dem Wertelement des Akkus, durch eine Grundrechenart.

add-, sub-, mult-, divColourConst Analog, verändert jedoch ein Farbregister.

add-, sub-, mult-, divLevelConst Analog, verändert ein Levelregister.

add-, sub-, mutl-, divColourValue2Global Die Methode verknüpfen den Wert des Farbregisters mit dem Wert des Akkus. Wird ein Blatt- oder Entscheidungsmodul der Seite Weiß ausgeführt wird das Weiße Farbregister benutzt, sonst das schwarze.

add- ,sub- ,mult- ,divLevelValue2Global Analog.

Lade- und Speicheroperationen

storeValueInAdfA Die Methode kopiert das Wertelement des Akkus in den Indexwert für die Materialebene.

storeValueInAdfB Die Methode kopiert das Wertelement des Akkus in den Indexwert für die Planungsebene.

loadValueFromAdfA Die Methode kopiert den Indexwert für die Materialebene in das Wertelement den Akku.

loadValueFromAdfB Die Methode kopiert den Indexwert für die Planungsebene in das Wertelement den Akku.

storeColourRegister Die Funktion kopiert das gesamte Akkuregister in ein Farbregister. Ob das Ziel des Kopiervorgangs das schwarze oder das weiße Register ist, hängt davon ab, für welche Farbe

storeAdfAIn-, storeAdfB-, storeValueInColourRegister Analog zu storeColourRegister kopiert jedoch nur die entsprechenden Elemente eines Registers.

storeLevelRegister Die Funktion kopiert den Akku in das aktuelle Levelregister.

storeAdfA-, storeAdfB-, storeValueInLevelRegister Analog zu storeLevelRegister, kopiert jedoch nur die entsprechenden Elemente eines Registers.

load- ,loadAdfA- ,loadAdfB- ,loadValueInColourRegister Analog zu den store-Methoden kopiert nur in das Akkuregister.

load- ,loadAdfA- ,loadAdfB- ,loadValueInLevelRegister Analog zu den store-Methoden kopiert nur in das Akkuregister.

Operationen zum setzen von Konstanten und sonstiges

getDepth Ermittelt die aktuelle Tiefe des Suchbaumes und speichert sie im Akku.

setAdfA Setzt den Indexwert der Materialebene auf eine konstante Zahl.

setAdfB Setzt den Indexwert der Planungsebene auf eine konstante Zahl.

setValue Setzt das Wertelement des Akkus auf eine konstante Zahl.

8.2 Funktionen qoopy

Zustandsfunktionen die aktuelle Brettstellung überprüfen, sie haben einen Boolschen-Wert als Rückgabewert.

Eröffnung Die Methode prüft, ob eine Eröffnung vorliegt. Dies ist der Fall, wenn der weiße Spieler über mindestens neun Figuren in der ursprünglichen Position befinden.

Mittelspiel Ein Mittelspiel liegt vor, wenn mehr als fünf Figuren vom Typ Springer, Läufer, Turm oder Dame nicht in der Anfangsstellung sind, oder keine Eröffnung festgestellt wird.

Endspiel Ein Endspiel liegt vor wenn keine Eröffnung oder kein Mittelspiel erkannt wird.

Schlagzug Ein Schlagzug liegt vor, wenn die aktuelle Brettstellung durch einen Zug erreicht wurde, bei dem eine Figur geschlagen wurde.

WeissAmZug Die Methode gibt den Wert wahr zurück, wenn der weiße Spieler in der aktuellen Brettstellung am Zug ist.

Schach Es wird Schach angezeigt, wenn der Spieler der am Zug ist in der untersuchten Brettstellung im Schach steht.

Wurzel Gibt wahr zurück, wenn der Knoten des Spielbaums der ausgeführt wird die Wurzel ist.

Bauernzug Ein Bauernzug liegt vor, wenn der Zug, der zur aktuellen Brettstellung geführt hat ein Bauernzug war. Analog die Funktionen **Sringerzug, Läuferzug, Turmzug, Damenzug, Königszug**.

Promotion Eine Promotion wird erkannt, wenn der Zug, der zur aktuellen Brettstellung geführt hat ein, eine Umwandlung eines Bauern war.

100000KnotenVerbraucht Die Methode gibt wahr zurück, wenn bei der aktuellen Suche mindestens 100.000 Knoten verbraucht wurden.

Die Terminal Funktionen des Tiefen Modul bilden zusammen mit den Zustandsfunktionen das Schachwissen des GP-Module im qoopy System.

Figurenwert Der Wert der Figur, die gezogen wurde, um die aktuelle Brettstellung zu erreichen.

WertDerGeschlagenenFunktion Die Methode gibt den Wert der geschlagenen Figur zurück, wenn keine Figur geschlagen wurde wird der Wert 0 zurückgegeben.

Zugnummer Die Funktion ermittelt die Anzahl der Halbzüge, die zum erreichen der Stellung gebraucht wurden.

Suchtiefe Die aktuelle Suchtiefe im Spielbaum.

Resttiefe Die restliche Suchtiefe, die das Suchtiefenmodule noch maximal ausführen kann.

ResttiefenDelta Der Deltawert gibt an, um wie viel die Resttiefe im nächsten Schritt verändert wird. Der Wert von Delta kann nur -2, -1, 0, 1 oder 2 sein.

Alpha Die Alpha-Scharanke des Alpha-Beta Algorithmus im aktuellen Knoten.

Beta Die Beta-Scharanke des Alpha-Beta Algorithmus im aktuellen Knoten.

Stellungswert Der Wert der aktuellen Brettstellung in Suchbaum. Der Wert wird durch das Stellungsbewertungsmoduledes Individuums ermittelt.

LetzterVerzweigungsgrad Der Verzweigungsgrad des Knotens, als der Spieler das letzte Mal gezogen hat.

Zugsortierungswert Die Methode ermittelt den Wert für den Zug, der zur aktuellen Stellung geführt hat. Hierzu wird das Zugsortierungsmodul des Individuums genutzt.

Knotenverbrauch Der aktuelle prozentuale Verbrauch, des für diese Suche zur Verfügung stehenden Kontenmenge.

AnzahlSpringerLäuferTurmDame Anzahl der Springer, Läufer, Türme und der Dame auf dem Feld.

Die Funktionsmenge des Tiefen Modul haben anders als die Terminalfunktionen kein Schachwissen.

+, -, * / Die arithmetischen Funktionen verknüpfen den Akkumulator mit einem anderen Operanden und legen das Ergebnis wieder im Akkumulator ab.

IncHorizon Die Funktion erhöht den Deltawert, mit dem die Resttiefe, nach der Ausführung des Suchtiefen-Modules, verändert wird. Der Deltawert kann maximal um den Wert zwei erhöht werden. Analog die Funktion **DecHorizon**.

Sinus, Sigmoid Die Sinusfunktion und die Sigmoidfunktion $\frac{1}{1+e^{-operand}}$.

SpeicherLevelReg Die Funktion speichert ein Levelregister im Akkumulator.

SpeicherGameReg Die Funktion speichert ein Gameregister im Akkumulator.

SpeicherSearchReg Die Funktion speichert ein Searchregister im Akkumulator.

Laden Die Funktion lädt den Wert des Operanden in das Level-, Game- oder Searchregister.

If Wenn der Konditionalteil der Funktion wahr ist, wird die Funktion in der Konklusion ausgeführt, sonst wird die Funktion übersprungen.

8.2.1 Parameter der Gegnerprogramme im qoopy-System

1. Bauer [100], Springer [340], Läufer [340], Turm [500], Dame [900]
2. Bestrafung für einen Läufer in der Anfangsposition [15].
3. Bonus für einen Bauern, der das Zentrum des Schachbrettes erreicht hat [15].
4. Bestrafung, wenn zwei Bauern auf der gleichen Linie stehen [30].
5. Bonus für einen Bauern, der auf beiden Nachbarlinien keinen gegnerischen Bauern hat [20].
6. Bestrafung für einen Bauern der keinen Bauern auf den Nachbarlinien hat, der näher an der ersten Reihe des Gegners ist [15].
7. Bonus für einen Freibauern, der durch einen Turm beschützt wird [20].
8. Bonus für einen Bauern, wenn er in der nähe der ersten Reihe des Gegners ist (Promotionsmöglichkeit) [300].
9. Bonus wenn beide Läufer vorhanden sind [20].
10. Bestrafung für einen Läufer, der sich in der Startposition aufhält [15].
11. Bestrafung für Läufer in einer blockierten Stellung. Eine blockierte Stellung ist in diesem Zusammenhang eine Stellung in der drei eigene Bauern sich im Bereich des erweiterten Zentrums befinden und Felder besetzen, die die gleiche Farbe haben wie die Felder auf denen sich der Läufer bewegt [20].

12. Bonus für den Springer, wenn es sich um einen blockierte Stellung handelt. Für einen Springer handelt es sich um eine blockierte Stellung, wenn sich mehr als sechs Bauern im erweiterten Zentrum auf halten. Durch die Fortbewegungsweise des Springers wird er aufgewertet [20].
13. Bonus für den Springer, wenn er mehr als sechs Felder erreichen kann [15].
14. Bonus für das Pferd in einer geschlossenen Position [20].
15. Bestrafung wenn auf jeder Seite des Pferdes ein generischer Bauer steht [25].
16. Bonus für den Turm, falls er von dem anderen Turm gedeckt wird [15].
17. Bonus für den Turm, wenn er auf eine halb offenen Linie steht [15].
18. Bonus für den Turm, wenn er auf eine offenen Linie steht [15].
19. Bonus für den Turm für andere Vorteile [10].
20. Bestrafung für den König, wenn er die erste Reihe während des Eröffnungs- oder Mittelspiels verlässt [15].
21. Bonus nachdem eine Rochade ausgeführt wurde [20].
22. Bestrafung für die Rochade, wenn es eine Schwäche in der Bauernstruktur vor dem König gibt [15].
23. Bestrafung wenn die Möglichkeit einer Rochade vergeben wurde [20].
24. Zufallswert, der zum Wert der Stellung addiert oder subtrahiert wird [20].

8.3 Funktionen GeneticChess

8.3.1 Materialbewertungs Modul

getFigureValue Die Funktion gibt den Wert einer abgefragten Figur zurück. Der Wert einer Figur wird mittels des Individuen ROMs bestimmt.

getFigureValueAtPos Die Funktion gibt den Wert einer Figur an einer Position des Brettes zurück. Steht auf dem Feld keine Figur, wird der Wert 0 zurück gegeben.

getWhiteValue, getBlackValue Die Funktion gibt den Materialwert aller weißen (schwarzen) Figuren zurück.

getNumOfWhitePawns, -Knigths, -Bishops, -Rooks Die Funktion gibt den Materialwert des weißen Bauern (Pferdes, Läufers, Turmes) zurück, wie sie durch das Individuen ROM definiert wird.

getNumOfBlackPawns, -Knigths, -Bishops, -Rooks Die Funktion gibt den Materialwert des schwarzen Bauern (Pferdes, Läufers, Turmes) zurück, wie sie durch das Individuen ROM definiert wird.

numOfWhiteFiguresInBlackCenter Die Funktion berechnet die Anzahl der weißen Figuren im schwarzen Zentrum. Analog die Funktionen **numOfBlackFiguresInBlackCenter, numOfWhiteFiguresInWhiteCenter, numOfBlackFiguresInWhiteCenter**.

advancedPawnValueWhite Die Funktion berechnet den Bonus für den weißen Bauern, wenn er sich im erweiterten Zentrum aufhält. Analog die Funktionen **advancedKnightValueWhite, advancedBishopValueWhite, advancedRookValueWhite, advancedPawnValueBlack, advancedKnightValueBlack, advancedKnightValueBlack, advancedBishopValueBlack, advancedRookValueBlack**.

endgameIndex Die Funktion bewertet, ob eine Stellung eine Endspielstellung ist.

8.3.2 Zugbewertungs Modul

numOfAttacksBefore Die Anzahl der angegriffenen Figuren bevor der aktuelle Zug ausgeführt wird. Dies wird von dem Spieler mit dem Zugrecht aus betrachtet.

numOfDefensedBefore Die Anzahl der gedeckten Figuren bevor der aktuelle Zug ausgeführt wird.

attackDefenseDiffBefore Die Differenz der angegriffenen Figuren und der gedeckten Figuren bevor der Zug ausgeführt wird.

numOfAttacksAfter Die Anzahl der angegriffenen Figuren nachdem der aktuelle Zug ausgeführt wird.

numOfDefensedAfter Die Anzahl der gedeckten Figuren nachdem der aktuelle Zug ausgeführt wird.

attackDefenseDiffAfter Die Differenz der angegriffenen Figuren und der gedeckten Figuren nachdem der Zug ausgeführt wird.

captureValueOfMove Wert der Figur, die durch den Zug geschlagen würde. Wenn keine Figur geschlagen wird, gibt die Funktion den Wert Null zurück.

captureFigureOfMove Die Methode gibt die interne Zahl der Spielfigur zurück, die durch den letzten Zug geschlagen wurde.

looseValueOfMove Die Methode berechnet den Wert der Figur die durch den letzten Zug geschlagen wurde.

maxLooseValueAfterMove Die Funktion berechnet den maximalen Materialverlust, der nach einem Zug auftreten kann.

maxLooseValueBeforeMove Die Funktion berechnet den maximalen Materialverlust, der durch einen Zug bei der aktuellen Stellung passieren kann.

maxCaptureValueAfterMove Die Funktion berechnet den maximalen Materialverlust, der nach einem Zug auftreten kann.

maxCaptureValueBeforeMove Die Funktion berechnet den maximalen Materialgewinn, der durch einen Zug bei der aktuellen Stellung passieren kann.

numOfThreatsOfSide Die Methode berechnet die Anzahl der Züge, die eine eigene Figur bedrochen.

numOfCoverOfSide Die Funktion zählt alle Deckungen für eigene Figuren.

moveFigureValue Die Methode gibt den Wert der Figur zurück, die greade am Zug ist.

getValue Die Methode gibt den Wert eines Knotens im Suchbaum zurück, der für diese Stellung berechnet wurde.

getBoardValue Der einfache Matrialwert der Stellung.

8.3.3 Tiefen Modul

currentDepth Die Funktion gibt die aktuelle Tiefe in der Rekursion an.

maxDepth Die Funktion gibt die maximale erlaubte Tiefe für die Rekursion an, die durch einen Parameter angegeben wird.

numOfExpandedNodes Die Methode zählt die Anzahl der expandierten Züge in der aktuellen Stellung.

numOfUnexpandedNodes Die Methode zählt noch nicht expandierten Züge der aktuellen Stellung.

isInWindow Die Funktion überprüft ob sich der Wert der Stellung in dem Alpha-Beta Fenster befindet oder nicht.

moveRang Die Funktion gibt den Rang des aktuellen Zuges in der Liste aller Züge nach der Sortierung an.

boardValue Die Funktion gibt einen den Wert der aktuellen Stellung an.

moveValue Die Funktion gibt die Wert des Zuges an.

wasBeatMove Test ob der aktuelle Zug ein Schlagzug ist.

numOfKingMoves Anzahl der Züge die ein König durchführen kann. Analog die Funktionen **numOfQueenMoves, numOfPawnMoves, numOfBishopMoves, numOfKnightMoves, numOfRookMoves**.

bestMoveValue Der Wert des besten Zuges. Analog die Funktionen **worstMoveValue, currentMoveValue**.

numOfThreatsOfInd Anzahl der Bedrohungen für das Individuum. Analog die Funktionen **numOfThreatsOfOppo, numOfCoversOfOppo**.

ifCurrentDepthIsBigger Test ob die aktuelle Tiefe größer ist als ein gegebener Wert. Analog die Funktion **ifCurrentDepthIsSmaller**.

isBiggerThanAlpha Test ob ein gegebener Wert größer ist als der Alphawert. Analog die Funktion **isSmallerThanBeta**.

8.3.4 Entscheidungsfunktionen

isOpenPhase Es wird überprüft ob das Spiel noch in der Eröffnungsphase ist. Analog die Funktionen **isMiddlePhase, isEndPhase**.

ifCurrentDepthIsBigger Überprüft ob die aktuelle Tiefe im Suchbaum größer ist als die maximale Tiefe. Analog die Funktion **ifCurrentDepthIsSmaller**.

isBiggerThanAlpha Die Methode überprüft ob der aktuelle Wert der Stellung größer als der Alphawert ist. Analog die Funktion **isSmallerThanBeta**.

isIndKingUnderAttack Die Methode überprüft ob der König des Individuums von einer anderen Figur angegriffen wird. Analog die Funktionen **isIndQueenUnderAttack, isOppoKingUnderAttack, isOppoQueenUnderAttack**.

isIndKingUnderCover Die Methode überprüft ob der König des Individuums gedeckt ist. Analog die Funktionen **isIndQueenUnderCover, isOppoKingUnderCover, isOppoQueenUnderCover**.

8.3.5 Branchingfunktionen

R0< und R0> Überprüft ob das Register Null größer oder kleiner einem Wert ist.

ifIndOfQueens Die Funktion überprüft ob in der Stellung des Individuums noch eine Königin vorhanden ist. Analog die Funktionen **ifIndTwoBishops, ifIndMoreKnights, ifIndMoreBishops**.

ifOppoOfQueens Die Funktion überprüft ob in der Stellung des Gegners noch eine Königin vorhanden ist. Analog die Funktionen **ifOppoTwoBishops, ifOppoTwoKnights**.

currentDepthIsBigger Überprüft ob die aktuelle Tiefe im Suchbaum größer ist als die maximale Tiefe. Analog die Funktion **currentDepthIsSmaller**.

biggerThanAlpha Die Methode überprüft ob der aktuelle Wert der Stellung größer als der Alphawert ist. Analog die Funktion **smallerThanBeta**.

Über den Autor

Von 1992 bis 1998 Studium der Informatik mit Nebenfach Betriebswirtschaftslehre an der Universität Dortmund, Abschluss DiplomInformatiker. Von 1998 bis 2003 wissenschaftlicher Mitarbeiter in der Arbeitsgruppe von Prof. Dr. Wolfgang Banzhaf am Lehrstuhl für Systemanalyse der Universität Dortmund. Seit 2002 Mitarbeiter und Gesellschafter der Firma Dortmund Intelligence Project GmbH. Schwerpunkte der wissenschaftlichen Arbeit liegen auf den Gebieten der Genetischen Programmierung.

Publikationen

[2002b] R. Groß, K. Albrecht, W. Kantschik und W. Banzhaf, *Evolving Chess Playing Programs*, In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002), p. 740-747, Morgan Kaufmann Publishers, San Francisco, CA 94104, USA , 2002.

Der Eigenanteil beträgt 30 %, er umfasst das Schreiben des Papers, die Begleitung der Implementierung und die Durchführung der Tests.

[2002a] W. Kantschik und W. Banzhaf, *Linear-Graph GP - A new GP Structure*, In: Genetic Programming, Proceedings of EuroGP'2002, James A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan und Andrea G. B. Tettamanzi (eds.), LNCS 2278, p. 83-92, Springer-Verlag, Berlin, 2001.

Der Eigenanteil beträgt 90 %, er umfasst das Schreiben des Papers, die Implementierung des Systems und die Durchführung der Tests.

[2001] W. Kantschik und W. Banzhaf, *Linear-Tree GP and its comparison with other GP structures*, In Genetic Programming, Proceedings of EuroGP'2001, J. F. Miller, M. Tomassini, P. Luca Lanzi, C. Ryan, A. G. B. Tettamanzi und W. B. Langdon (eds.), LNCS 2038, p. 302-312, Springer-Verlag, Berlin, 2001.

Der Eigenanteil beträgt 90 %, er umfasst das Schreiben des Papers, die Implementierung des Systems und die Durchführung der Tests.

[2000] J. Busch, W. Kantschik, H. Aburaya, K. Albrecht, R. Groß, P. Gundlach, M. Kleefeld, A. Skusa, M. Villwock,T. Vogd und W. Banzhaf *Evolution von GP-Agenten mit Schachwissen sowie deren Integration in ein Computerschachsystem* SYS Report, Nummer SYS-01/00, ISSN 0941-4568, 308 Seiten, Univ. Dortmund, Informatik.

Der Eigenanteil beträgt 15 %, er umfasst die Koordination der PG, Korrekturlesen, Auswahl der Tests, Anpassung des SYSGP-Systems.

[1999c] P. Dittrich, A. Skusa, W. Kantschik, and W. Banzhaf *Dynamical Properties of the Fitness Landscape of a GP Controlled Random Morphology Robot* In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'99), vol. 2, p. 1002-1008, Morgan Kaufmann, San Francisco, CA, Orlando, Florida, July 13-17, 1999.

Der Eigenanteil beträgt 15 %, er umfasst die Anpassung des SYSGP-Systems für die Robotersteuerung.

[1999b] W. Kantschik, P. Dittrich, M. Brameier, and W. Banzhaf *Empirical Analysis of Different Levels of Meta-Evolution* Congress on Evolutionary Computation (CEC99), July 6-9, 1999, Washington DC, USA, vol. 3, p. 2086-2093, IEEE, 1999.

Der Eigenanteil beträgt 80 %, er umfasst das Schreiben des Papers, die Implementierung des Systems und die Durchführung der Tests.

[1999a] W. Kantschik, P. Dittrich, M. Brameier, und W. Banzhaf *Meta-Evolution in Graph GP* In: Genetic Programming, Second European Workshop (EuroGP'99), Proceedings, R. Poli, P. Nordin, W.B. Langdon, T.C. Fogarty (eds.), LNCS 1598, p. 15-28, Springer, Berlin, 1999

Der Eigenanteil beträgt 80 %, er umfasst das Schreiben des Papers, die Implementierung des Systems und die Durchführung der Tests.

[1998] M. Brameier, W. Kantschik, P. Dittrich, und W. Banzhaf *SYSGP - A C ++ library of different GP variants* Series Computational Intelligence", Internal Report of SFB 531, No. 48/98, ISSN 1433-3325, Univ. of Dortmund, D-44221 Dortmund, Germany, 1998.

Der Eigenanteil beträgt 50 %, er umfasst die Implementierung des SYSGP-Systems und die Durchführung der Tests.

Literaturverzeichnis

[1] G.M. Adelson-Velsky, V.L. Arlazarov, A.R. Bitman, A.A. Zhivotovsky, and A.V. Uskov. Programming a computer to play chess. In *1st Summer School on Mathematical Programming*, volume II, pages 216–252, 1969.

[2] K. Albrecht, R. Arnold, M. Gähwiler, and R. Wattenhofer. Aggregating information in peer-to-peer systems for improved join and leave. In *Peer-to-Peer Computing*, pages 227–234, 2004.

[3] I. Althöfer. On the K-Best Mode in Computer Chess: Measuring the Similarity of Move Proposals. *ICCA Journal*, 20(3):152–165, September 1997.

[4] P. J. Angeline. Genetic programming and emergent intelligence. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 4, pages 75–98. MIT Press, 1994.

[5] P. J. Angeline. Multiple interacting programs: A representation for evolving complex behaviors. *Cybernetics and Systems*, 29(8):779–806, 1998.

[6] P.J. Angeline. Subtree crossover: Building block engine or macromutation? In J. Koza, K. Deb, M. Dorigo, D. Fogel, M. Garzon, H. Iba, and R. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 9–17, San Francisco, CA, 1997. Morgan Kaufmann.

[7] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann, San Francisco und dpunkt verlag, Heidelberg, 1998.

[8] D. F. Beal. *The Nature of Minimax Search.* PhD thesis, University of Maastricht, 1999. Diss.Nr.99-3.

[9] D. F. Beal and M. C. Smith. *Learning Piece Values Using Temporal Differences.* Department of Computer Science, University of London, http://ybishop.cjb.net.

[10] D. F. Beal and M. C. Smith. Learning Piece Values Using Temporal Differences. *ICCA Journal*, 20(3):147–151, September 1997.

[11] D. F. Beal and M. C. Smith. Learning Piece-square Values Using Temporal Differences. *ICCA Journal*, 22(4):223–235, December 1999.

[12] H. C. Bennet. *The Universal Turing Machine*, chapter Logical Depth and Physical Complexity, pages 227 – 257. Hamburg/Berlin (Kammerer & Unverzagt), 1988.

[13] H. Berliner. *Chess as Problem Solving: The Development of a Tactics Analyzer.* PhD thesis, Computer Science Dept. Carnegie-Mellon University, 1974.

[14] H. Berliner. *The System.* Gambit Publication Ltd, 1999. ISBN 1 901 983 10 2.

[15] J. Birmingham and P. Kent. Tree-Searching and Tree-Pruning Techniques. In M.R.B. Clarke, editor, *Advances in Computer Chess 1*, pages 89–97. 1977.

[16] J. Birmingham and P. Kent. Mate at a Glance. In M.R.B. Clarke, editor, *Advances in Computer Chess 2*, pages 122–130. 1980.

[17] S. De Blecourt. The legacy of arpad elo - the development of a chess-rating system. Technical Report VRT-2, Universiteit von Amsterdam, Faculteit der Psychologie, 1998.

[18] M. Brameier and W. Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26, February 2001.

[19] M. Brameier and W. Banzhaf. Evolving teams of mutiple predictors with Genetic Programming. Technical report, Universität Dortmund SFB 531, 2001. Data available from the authors.

[20] M. Brameier and W. Banzhaf. Explicit control of diversity and effective variation distance in linear genetic programming. In James A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan, and Andrea G. B. Tettamanzi, editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 37–49, Kinsale, Ireland, 3-5 April 2002. Springer-Verlag.

[21] M. Buro. From Simple Features to Sophisticated Evaluation Functions. In H.J.van den Herik and H.Iida, editors, *Computers and Games: Proceedings CG'98. LNCS 1558*, pages 126–145. Springer Verlag, Berlin, 1999.

[22] J. Busch, W. Kantschik, H. Aburaya, K. Albrecht, R. Groß, P. Gundlach, M. Kleefeld, A. Skusa, M. Villwock, T. Vogd, and W. Banzhaf. Evolution von GP-Agenten mit Schachwissen sowie deren Integration in ein Computerschachsystem. SYS Report SYS-01/00, ISSN 0941-4568, Systems Analysis Research Group, Univ. Dortmund, Informatik, 10 2000.

[23] S. Cracraft and Stanback J. Gnu chess, 1984. Public chess software.

[24] R. L. Cramer. A representation for adaptive generation of simple sequential programs. *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, pages 183–187, 1985.

[25] C. F. Cramerer. *Behavioral game theory: experiments in strategic interaction.* Princeton University Press, Princeton, New Jersey, 2003.

[26] D. D. Davis and C. A. Holt. *Experimental Economics.* Princeton University Press, Princeton, New Jersey, 1993.

[27] C. Donninger. Null Move and deep search: Selective search heuristics for obtuse chess programs. In *ICCA Journal*, volume 16, pages 137–143. 1993.

[28] H. L. Dreyfus. *What Computers Can't Do: The Limits of Artificial Intelligence.* NY: Harper & Row, New York, revised, edition edition, 1979.

[29] H. L. Dreyfus. *What Computers Still Can't Do: A Critique of Artificial Reason.* USA: MIT Press, Cambridge, MA, 1993.

[30] J. Farley. *Java Distributed Computing.* O'Reilly, 1998.

[31] M. M. Flood. Some experimental games. Technical Report Research Memorandum RM-789, RAND Corporation, Santa Monica, 1952.

[32] M. M. Flood. Environmental non-stationarity in a sequentail decisison-making experiment. pages 287–300, 1954.

[33] M. M. Flood. Game-learning theory and some decision making experiments. pages 139–158, 1954.

[34] L. Fogel, A. Owens, and M. Walsh. *Intelligence Through Simulated Evolution*. Wiley, New York NY, 1966.

[35] S. Forrest and M. Mitchell. Relative building-block fitness and the building-block hypothesis. In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 109–126. Morgan Kaufmann, San Mateo, CA, 1993.

[36] R. M. Friedberg. A learning machine - part i. *IBM Journal of Research and Development*, pages 2–11, 1958.

[37] J. Fürnkranz. Machine learning in computer chess: The next generation. *International Computer Chess Association Journal*, 19(3):147–161, September 1996.

[38] C. Gathercole and P. Ross. Dynamic training subset selection for supervised learning in genetic programming. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer, editors, *Parallel Problem Solving from Nature III*, volume 866 of *LNCS*, pages 312–321, Jerusalem, 9-14 October 1994. Springer-Verlag.

[39] J. Dollimore G.F. Coulouris and T. Kindberg. *Distributed Systems, Concepts and Design*. Addison-Wesley, 2 edition, 1994.

[40] M. Gherrity. *A Game Learning Machine*. PhD thesis, San Diego, University of California at San Diego, 1993.

[41] K.R.C. Greer, P.C. Ojha, and D.A. Bell. A Pattern-oriented Approach to Move Ordering: The Chessmaps Heuristics. *ICCA Journal*, 22(1):13–21, March 1999.

[42] R. Grimbergen. Plausible Move Generation Using Move Merit Analysis with Cut-Off Thresholds in Shogi. In *Computers and Games: Proceedings CG2000.*, Hamamatsu, Japan, 2000.

[43] R. Gross, K. Albrecht, W. Kantschik, and W. Banzhaf. Evolving chess playing programs. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 740–747, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[44] A. Hauptman and M. Sipper. Analyzing the intelligence of a genetically programmed chess player. In Franz Rothlauf, editor, *Late breaking paper at Genetic and Evolutionary Computation Conference (GECCO'2005)*, Washington, D.C., USA, 25-29 June 2005.

[45] A. Hauptman and M. Sipper. GP-endchess: Using genetic programming to evolve chess endgame players. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 120–131, Lausanne, Switzerland, 30 March - 1 April 2005. Springer.

[46] E. Heinz. Extended Futility Pruning. *ICCA Journal*, 21(2):75–83, June 1998.

[47] E. A. Heinz. A new self-play experiment in computer chess.

[48] E.A. Heinz. Efficient Interior-Node Recognition. *ICCA Journal*, 21(3):157–168, September 1998.

[49] E.A. Heinz. DARKTHOUGHT Goes Deep. *ICCA Journal*, 21(4):228–229, December 1998.

[50] E.A. Heinz. Adaptive Null-Move Pruning. *ICCA Journal*, 22(3):123–132, September 1999.

[51] E.A. Heinz. *Scalable Search in Computer Chess*. Vieweg, Germany, 2000.

[52] J. Hertzberg. *Planen: Einführung in die Planerstellungsmethoden der künstlichen Intelligenz*. Number Band 65 in Informatik. BI Wissenschaftsverlag, Mannheim, Wien, Zürich, 1989. ISBN 3-411-03223-5.

[53] J. Holland. *Adaption in Natural and Artifical Systems*. MI: The University of Michigan Press, 1975.

[54] L. Huelsbergen. Toward simulated evolution of machine language iteration. In J. R. Koza, D. E. Goldberg, D. B. Fogle, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 315 – 320, Stanford University CA., 1996. MIT Press Cambridge.

[55] R.M. Hyatt and M. Newborn. Crafty Goes Deep. *ICCA Journal*, 20(2):79–86, June 1997.

[56] H. Iida, K. Handa, and J. Uiterwijk. Tutoring Strategies in Game-Tree Search. *ICCA Journal*, 18(4):191–205, December 1995.

[57] A. Junghanns. Fuzzy numbers as a toll in chess programs. *ICCA Journal*, 17(3):41–48, 1994.

[58] A. Junghanns, C. Posthoff, and M. Schlosser. Search with fuzzy numbers. In *FUZZY IEEE/IFES'95*, pages 978–986, Yokohama, Japan, March 1995.

[59] H. Kaindl. *Problemlösen durch heuristsche Suche in der Artificial Intelligence*. Springer-Verlag, Wien, New York, 1989.

[60] G. K. Kalisch, J. W. Milnor, J. F. Nash, and E. D. Nering. Some experimental n-persons games. Technical Report Research Memorandum RM-948, RAND Corporation, Santa Monica, 1952.

[61] G. K. Kalisch, J. W. Milnor, J. F. Nash, and E. D. Nering. Some experimental n-persons games. pages 301–327, 1954.

[62] W. Kantschik and W. Banzhaf. Linear-Tree GP and its comparison with other GP structures. In J. F. Miller, M. Tomassini, P. Luca Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 302–312, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.

[63] W. Kantschik and W. Banzhaf. Linear-Graph GP - A new GP Structure. In J. F. Miller, M. Tomassini, P. Luca Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2002*, volume 2278 of *LNCS*, pages 83–92, Kinsale, Irland, 3-5April 2002. Springer-Verlag.

[64] W. Kantschik, P. Dittrich, M. Brameier, and W. Banzhaf. Empirical analysis of different levels of meta-evolution. In P. J. Angeline and V. W. Porto, editors, *1999 Congress on Evolutionary Computation (CEC'99)*, volume 3, pages 2086–2093, Washington D.C., July 6-9 1999. IEEE Press, Piscataway NJ.

[65] W. Kantschik, P. Dittrich, M. Brameier, and W. Banzhaf. Metaevolution in graph GP. In Riccardo Poli, Peter Nordin, William B. Langdon, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'99*, volume 1598 of *LNCS*, pages 15–28, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[66] R. E. Keller and W. Banzhaf. The evolution of genetic code in genetic programming. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1077–1082, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[67] R. E. Keller and W. Banzhaf. Evolution of genetic code on a hard problem. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 50–56, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[68] R. E. Keller, W. A. Kosters, M. van der Vaart, and M. D. J. Witsenburg. Genetic programming produces strategies for agents in a dynamic environment. In Hendrik Blockeel and Marc Denecker, editors, *Proceedings of the Fourteenth Belgium/Netherlands Conference on Artificial Intelligence (BNAIC'02)*, pages 171–178, Leuven, Belgium, 21-22 October 2002.

[69] G. Kendall and G. Whitwell. An evolutionary approach for the tuning of a chess evaluation function using population dynamics. In *In proceedings of Congress on Evolutionary Computation 2001 (CEC'01)*, pages 995–1002, COEX Center, Seoul, Korea, May 27-29 2001.

[70] G. Kendell and K. Whitewell. An evolutionary approach for the tunint of a chess evaluations function using population dynamics. In *Procedddings of the 2001 IEEE Congress on Evolutionary Computation*, pages 995–1002, 2001.

[71] G. Klaus and H. Leibscher. *Systeme Information Strategien.* VEB Verlag Technik Berlin, Berlin DDR, 1974.

[72] D. E. Knuth and W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

[73] J. Koza. *Genetic Programming.* MIT Press, Cambridge, MA, 1992.

[74] J. Koza. *Genetic Programming II.* MIT Press, Cambridge, MA, 1994.

[75] C. W. G. Lasarczyk, P. Dittrich, and W. Banzhaf. Dynamic subset selection based on a fitness case topology. *Evolutionary Computation*, 12(2), 2004.

[76] J. R. Levenick. Inserting introns improves genetic algorithm success rate: Taking a cue from biology. In Rick Belew and Lashon Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 123–127, San Mateo, CA, 1991. Morgan Kaufman.

[77] R. Levinson and R. Snyder. Adaptive Pattern-Oriented Chess. In L.A. Birnbaum and G.C. Collins, editors, *Machine Learning: Proceedings of the 8th International Workshop (ML91)*, pages 85–89, 1991.

[78] R. Levinson and R. Weber. Chess neighborhoods, function combination, and reinforcement learning. In T. Marsland and I. Frank, editors, *omputers and Games: Second International Conference, CG 2001*, volume 2063 / 2001 of *Lecture Notes in Computer Science*, pages 26–28. Springer-Verlag Heidelberg, October 2001.

[79] T.A. Marsland and J. Schaeffer, editors. *Computers, Chess, and Cognition.* Springer-Verlag, New York, 1990.

[80] A. Matanovic, editor. *640 best 64 golden games.* Sahoviski Informator, Beograd, 1996.

[81] M. Minsky. *Semantic Information Processing.* The M.I.T Press., Cambridge, Mass., 1968.

[82] P. Mysliwietz. *Konstruktuion und Optimierung von Bewertungsfunktionen beim Schach.* PhD thesis, Tehnical University of Denmanrk, Lyngby, 1994.

[83] J. F. Nash. Equilibrium points in N-person games. *Proc. Nat. Acad. Sci. U.S.A.*, 36:48–49, 1950.

[84] J. F. Nash. *Non-cooperative Games.* PhD thesis, Princeton University, 1950.

[85] J. Niehaus. *Graphbasierte Genetische Programmierung.* PhD thesis, Universität Dortmund, 2003.

[86] J. Niehaus and W. Banzhaf. Adaption of operator probabilities in genetic programming. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 325–336, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.

[87] J. Niehaus and W. Banzhaf. More on computational effort statistics for genetic programming. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 168–177, Essex, 14-16 April 2003. Springer-Verlag.

[88] N. J. Nilsson. *Problem-Solving Methodes in Artificial Intelligence.* Computer Science Series. McGraw-Hill Book Company, 1971.

[89] J. P. Nordin. A compiling genetic programming system that directly manipulates the machine code. Cambridge, 1994. MIT Press.

[90] J. P. Nordin and W. Banzhaf. Complexity compression and evolution. Pittsburgh, Penn., USA, 1995. Proceedings of the Sixth International Conference of Genetic Algorithms, Morgan Kaufmann Publishers.

[91] J. P. Nordin and W. Banzhaf. Evolving turing complete programs for a register machine with self modifying code. Pittsburgh, Penn., USA, 1995. Proceedings of the Sixth International conference of Genetic Algorithms, Morgan Kaufmann Publishers.

[92] P. Nordin, F. Francone, and W. Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. Technical Report SYS-3/95, Universität Fachbereich Informatik Lehrstuhl XI, Dortmund, May 1995.

[93] M. Olmer, W. Banzhaf, and P. Nordin. Evolving real-time behavior modules for a real robot with genetic programming. In *Proceedings of the international symposium on robotics and manufacturing*, Montpellier, France, 1996.

[94] M. O'Neill and C. Ryan. Genetic code degeneracy: Implications for grammatical evolution and beyond. In J. Nicoud D. Floreano and F. Mondada, editors, *ECAL99*, pages 149–153, Berlin, 1999. Springer-Verlag.

[95] M. O'Neill, C. Ryan, M. Keijzer, and M. Cattolico. Crossover in grammatical evolution: The search continues. In *EuroGP 2001*, pages 337–347, 2001.

[96] W. Pijls and A. De Bruin. Game Tree Algorithms and Solution Trees. In H.J.van den Herik and H.Iida, editors, *Computers and Games: Proceedings CG'98. LNCS 1558*, pages 195–204. Springer Verlag, Berlin, 1999.

[97] A. Plaat. *Research Re: search & Re-search*. PhD thesis, Rotterdam, Netherlands, 1996.

[98] A. Plaat, J. Schaeffer, W. Pijls, and A. De Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87:255–293, 1996.

[99] R. Poli. Evolution of graph-like programs with parallel distributed genetic programming. In Thomas Back, editor, *Genetic Algorithms: Proceedings of the Seventh International Conference*, pages 346–353, Michigan State University, East Lansing, MI, USA, 19-23 July 1997. Morgan Kaufmann.

[100] D. Rasmussen. Parallel chess searching and bitboards. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2004. Supervised by Prof. Jens Clausen.

[101] A. Reinefeld. A minimax algorithm faster than alpha-beta. In J. van den Herik, L.S. Herschberg, and J.W.H.M. Uiterwijk, editors, *Advances in Computer Chess*, volume 7, pages 237–25, 1994.

[102] A. Reinfeld. An improvement of the scout tree search algorithm. *ICCA Journal*, 6(4):4–14, June 1989.

[103] I. Roizen and J. Peal. A minimax algorithm better than alhpa-beta? yes and no. In *Artificial Intelligence*, volume 21, pages 199–220. 1983.

[104] J. Schaeffer. The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212, 1989.

[105] H-P. Schwefel. *Evolution and Optimum Seeking*. John Wiley & Sons, Inc., 1996.

[106] C.E. Shannon. A chess-playing machine. *Scientific American*, 182(314):48–51, February 1950. Reprinted in The World of Mathematics, edited by James R. Newman, Simon and Schuster, NY, Vol. 4, 1956, pp. 2124-2133.

[107] C.E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41(314):256–275, March 1950.

[108] J. H. Slagle and J. K. Dixon. Experiments with some programs that search game trees. *Journal fo the ACM*, 2(16):189–207, 1969.

[109] M. Spies. *Unsicheres Wissen.* Spektrum Verlag, Heidelberg, Berlin, Oxford, 1993. ISBN 3-86025-006-X.

[110] G. Stockman. A minimax algorithm better than Alpha-Beta? *Artificial Intelligence*, 12:179–196, 1979.

[111] E. Strouhal. *Acht mal acht: zur Kunst des Schachspiels.* Springer Verlag, 1996. ISBN 3-211-82775-7.

[112] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[113] O. D. Tabibi and N. S. Netanyahu. Verified null-move pruning. Technical Report CAR-TR-980, CS-TR-4406, UMIACS-TR-2002-39, Center for Automation Research, University of Maryland, 2002. Published in ICGA Journal, Vol. 25, No. 3, pp. 153–161.

[114] P. Tadepalli. Planning in games using approximately learned macros. In *Proceedings of the Sicth International Workshop on Machine Learning*, pages 221–223, Ithaca, 1989. Morgan Kaufmann.

[115] A. Teller. Evolving programmers: The co-evolution of intelligent recombination operators. Technical Report Currently unpublished technical report (Homepage), Department of Computer Science, Carnegie Mellon University, 1996.

[116] A. Teller and M. Veloso. PADO: A new learning architecture for object recognition. In Katsushi Ikeuchi and Manuela Veloso, editors, *Symbolic Visual Learning*, pages 81–116. Oxford University Press, 1996.

[117] A. Teller and M. Veloso. Pado: A new learning architecture for object recognition. In *Symbolic Visual Learning*, pages 81 –116. Oxford University Press, 1996.

[118] A. Teller and M. Veloso. Smart mutation. Technical Report Currently unpublished technical report (Homepage), Department of Computer Science, Carnegie Mellon University, 1996.

[119] C. Thornton. The building block fallacy, 1997.

[120] S. Thrun. Learning to play the game of chess. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 1069–1076. The MIT Press, Cambridge, MA, 1995.

[121] W. Tunstall-Pedoe. Genetic algorithms optimizing evaluation functions. *ICCA Journal*, 14(3):119–128, 1991.

[122] A. Turing. Chess. In B.V. Bowden, editor, *Faster than Thought*, pages 286–295, 1953.

[123] A. van Tiggelen and H. J. van den Herik. Alexs: An optimization approach for the endgame (knnp). *Advances in computer Chess 6*, pages 161–177, 1991.

[124] F. Vega-Redondo. *Evolution, Games, and Economic Behaviour.* Oxford University Press, Oxford, UK, 1996.

[125] J. von Neumann and O. Morgenstern. *The Theory of Games and Economic Behavior*. Princeton University Press, Princeton, New Jersey, 2nd edition edition, 1947.

[126] J. W. Weibull. *Evolutionary Game Theory*. MIT Press, Cambridge, Massachusetts, London, England, 1995.

[127] M. L. Wong and K. S. Leung. Evolving recursive functions for the even-parity problem using genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 11, pages 221–240. MIT Press, Cambridge, MA, USA, 1996.

Index

T

V

Z