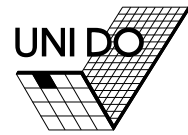


UNIVERSITÄT DORTMUND

■ FACHBEREICH INFORMATIK

Endbericht der PG 348 (*GenScha*):



Evolution von GP-Agenten mit Schachwissen sowie deren Integration in ein Computerschachsystem

Hassan Aburaya, Keno Albrecht, Roderich Groß,
Patrick Gundlach, Martin Kleefeld, Andre Skusa,
Martin Villwock, Thomas Vogd

Betreuer: Dipl.-Inform. Jens Busch
Dipl.-Inform. Wolfgang Kantschik

August 2000

INTERNE BERICHTE INTERNAL REPORTS

Projektgruppe am
Fachbereich Informatik
der Universität Dortmund



*Dem Individuum der Generation 115.
Es gab uns Hoffnung und Zukunft wieder.*

Inhaltsverzeichnis

1	Vorwort	2
2	Einleitung	4
2.1	Von Fruchtfliegen, Geist und Fußball	4
2.2	Biologisch inspirierte Verfahren	5
2.3	Methode und Zielsetzung	6
2.4	Überblick	7
3	Grundlagen	8
3.1	Das Schachspiel	8
3.2	Baumsuchverfahren und Stellungsbewertungen	12
3.3	Maschinelles Lernen und Evolutionäre Algorithmen	33
3.4	Genetisches Programmieren	51
3.5	Schach und Maschinelles Lernen	71
3.6	Bewertung von Spielstärke im Schach	76
4	Entwicklungsumgebung und Werkzeuge	80
4.1	Rechner	80
4.2	SYSGP	80
4.3	C++	96
4.4	Doc++	97
4.5	Make	98
4.6	Versionsmanagement	104
4.7	Oracle	108
4.8	PVM	115
4.9	xboard	118
4.10	Crafty	121
5	Vorgehen	122
5.1	Allgemeiner Überblick	122
5.2	Auswahl der Trainingsdaten	124
6	Architektur	127
6.1	Konzept	127
6.2	Systemdesign	159
7	Analyse und Ergebnisse	199
7.1	Analysetools	199
7.2	Analyse der Partiidatenbank	206
7.3	Analyse der Stufe a	211
7.4	Analyse der Stufe b	235
7.5	Analyse der Stufe c	246
8	Fazit	267
	Literaturverzeichnis	269

1 Vorwort

Die Durchführung von Projektgruppen ist Bestandteil der Studiengänge *Informatik* und *Angewandte Informatik* der *Universität Dortmund*. Im Laufe von zwei Semestern besteht die Möglichkeit, in einem Team mit anderen Studenten eine Aufgabe durchzuführen, die aufgrund ihres Umfangs nicht von Einzelnen und in überschaubarer Zeit zu bewältigen wäre. Das gibt sowohl Gelegenheit zur Übung in Teamarbeit und Koordination, als auch zur Bearbeitung einer komplexen Problemstellung über einen längeren Zeitraum.

Im Falle dieser Projektgruppe wird eine populäre Anwendung der Informatik aus einer anderen Perspektive betrachtet: Das Ziel ist die Erstellung *Schach spielender Programme* unter Anwendung von *Genetischer Programmierung* (GP), einem Verfahren aus dem Gebiet der *Evolutionären Algorithmen* (EA).

Schach spielende und lernende Computerprogramme werden schon seit längerem im Bereich der *Künstlichen Intelligenz* (KI) erforscht. Zu Beginn der Forschungen in der KI betrachtete man das Schachspiel als einen der Schlüssel zum Verständnis des menschlichen Geistes. Darüber hinaus konnten Ergebnisse für den Umgang mit großen Suchräumen erwartet werden. Denn trotz der vollständigen Information, die das Schachspiel kennzeichnet, ist es auch unter Annahme höchster Rechenleistung nahezu unmöglich, alle Möglichkeiten einfach auszurechnen und die Beste daraus zu wählen. Somit sind „intelligente“ Heuristiken nötig, die geschickt „Vermutungen“ erstellen können, welcher Pfad im Suchraum der Lösungen vielversprechend ist.

Die Konstruktion solcher Heuristiken kann dabei auf unterschiedliche Art und Weise erfolgen. Zum einen können menschliche Überlegungen und Strategien direkt in eine algorithmische Formulierung übertragen werden, so wie es beispielsweise bei Suchalgorithmen erfolgt. Ein anderer Weg ist es, den Computer *selber* den Suchraum explorieren zu lassen. Durch Adaptation an vorgegebene Ziele oder eine durch zufällige Veränderungen gekennzeichnete Suche soll eine Lösung gefunden werden, die entweder schwierig algorithmisch zu formulieren oder schlicht unbekannt ist. Dieses Vorgehen wird im *Maschinellen Lernen* (ML) erforscht. Beispielhaft hierfür sind *Künstliche Neuronale Netze* (KNN).

Eine weitere Möglichkeit dem Computer die Erkundung des Suchraumes zu überlassen, liegt in *Evolutionären Algorithmen*. Diese wurden für die Optimierung von Parametern analytisch schwer handhabbarer Funktionen entwickelt, deren Werte bestimmten Randbedingungen genügen sollen. Das Verfahren besteht nun darin, in einer der Darwinschen Evolutionstheorie angelehnten *künstlichen* Evolution Lösungsmöglichkeiten zu testen, schlechte Versuche zu verwerfen und bessere Ergebnisse durch die Anwendung *genetischer Operatoren* – *Mutation* oder *Rekombination* – zu erreichen. Damit wird eine Strategie verfolgt, die zwischen der Bewahrung bisher guter Lösungen und der großräumigen Erkundung des Suchraumes liegt. Mit *Genetischer Programmierung* wird das Verfahren der künstlichen Evolution auf die symbolische Ebene der Programme übertragen. Somit werden sich selbst erstellende und verändernde Programme „gezüchtet“.

Die Aufgabenstellung unseres Projektes ist es, herauszufinden, ob sich die Ansätze des Genetischen Programmierens sinnvoll für das Lernen von Schachpro-

grammen anwenden lassen. Obwohl es sich bei Aufgabenstellung und Lösungsansatz scheinbar um sehr unterschiedliche Domänen handelt – Schach basiert auf streng logischen Gesetzmässigkeiten und die künstliche Evolution nutzt den Faktor *Zufall*, um zu Lösungen zu gelangen – besteht die Vermutung, dass durch eine effektive Exploration des Suchraumes Schach spielende Computerprogramme entstehen können.

Ein großer Vorsatz scheint im Anfang toll;
Doch wollen wir des Zufalls künftig lachen,
Und so ein Hirn, das trefflich denken soll,
Wird künftig auch ein Denker machen.

(J.W. Goethe, Faust II, *Laboratorium*)

2 Einleitung

Das Schachspiel ist in der Computerwissenschaft schon so etwas wie ein alter Bekannter. Aufgrund des logischen Spielaufbaus ist es einfach auf Computern zu implementieren. Trotzdem sind die Fülle der Zugmöglichkeiten und der dadurch entstehende Rechenaufwand für die Vorhersage von günstigen Zugfolgen immens. Dabei sind menschliche Schachspieler dazu in der Lage, Strategien zu bilden und gut zu spielen, ohne dafür sämtliche Züge systematisch durchzurechnen. Erfahrung, Lernen und das Erkennen von *Ballungen* – Mustern auf dem Schachbrett – kennzeichnen das Vorgehen professioneller Schachspieler.

2.1 Von Fruchtfliegen, Geist und Fußball

Abstraktion und die Fähigkeit zum logischen Denken galten lange Zeit als zentrale Eigenschaft des menschlichen Geistes. Das ließ das Schachspiel zu einem besonderen Symbol in der Erforschung Künstlicher Intelligenz werden. Für die KI war Schach nicht nur eine schwierige Aufgabe, deren Lösung auch bei vielen anderen schwierigen Probleme im Bereich der Logik hilfreich sein würde. Das Spiel diente vielmehr als eine Art „Drosophila“ in der Erforschung menschlich-kognitiver Fähigkeiten und deren Umsetzung auf Computern. So wie den Biologen der Aufbau der Fruchtfliege weitestgehend bekannt ist, ist es auch bei der „internen Mechanik“ des Schachspiels. Die Fruchtfliege *Drosophila* ist darüberhinaus für Genetiker interessant, weil ihre genetischen Eigenschaften denen anderer Tiere und insbesondere denen des Menschen ähnlich sind. Daher lassen sich Erkenntnisse im Labor auf andere Organismen übertragen. Analog dazu ging man in der Künstlichen Intelligenz von einer Vergleichbarkeit der strategisch-logischen Vorgehensweise im Schachspiel mit grundsätzlichen Mechanismen menschlichen Denkens aus.

Das Ziel, denkende Computer im Allgemeinen und lernende Schachprogramme im Besonderen zu entwickeln, ist aus verschiedenen Gründen nicht erreicht worden. Aufgrund fortschreitender Kommerzialisierung auch des Computerschachs und immer mehr verfügbarer Rechenleistung setzen sich Ansätze durch, die im Wesentlichen den Suchraum „blind“ und ohne die Anwendung „intelligenter“ Vorgehensweisen durchsuchten. So ist der Turniersieg von *Deep Blue* gegen den Schachweltmeister Gari Kasparow nicht durch einen denkenden Computer erbracht worden, sondern wurde durch die Übermacht der Rechenkraft und die fehlgeschlagene Verwirrungstaktik des menschlichen Spielers verursacht. Eine weitere Feststellung war die, dass man auch mit Hilfe Schach *lernender* Software keineswegs mehr über das menschliche Schachspielen und noch weniger über die grundsätzliche Funktionalität menschlichen Geistes erfahren hatte. Strate-

gien, die denen menschlicher Spieler ähnlich waren, bildeten sich nicht heraus und Schach wurde immer weniger als ein Spezifikum menschlicher Fähigkeiten gesehen.

Aus diesen und anderen Entwicklungen in der KI erklärt sich der Wechsel zu einer neuen Drosophila, dem *Fußball* (RoboCup), sowohl mit Robotern verschiedener Größenklassen als auch in Simulationen gespielt. Intelligenz ist wahrscheinlich nicht zu trennen von einer Einbettung in einen Körper und andauernder Wechselwirkung mit einer Umwelt. Ebenso hängt dieser Begriff wohl mit Tätigkeiten zusammen, denen im Alltagsgebrauch Stumpfsinn oder Einfachheit nachgesagt wird. Dazu gehören beispielsweise der aufrechte Gang, das Eindrehen von Schrauben oder die Benutzung von Messer und Gabel. Gerade diese Fähigkeiten aber sind es, die für Computer oder Roboter zur Zeit noch eine große Herausforderung darstellen. Neben solchen „Alltagsfähigkeiten“ sind es Aufgaben wie Teamarbeit oder die Verfolgung von Gesamtzielen durch Aufteilung in Teilziele, die im RoboCup benötigt werden.¹

2.2 Biologisch inspirierte Verfahren

Zu den verschiedenen Aufspaltungen und Paradigmenwechseln innerhalb der KI-Forschung gehört der noch relativ junge Bereich der *Computational Intelligence* (CI). Durch die Verwendung des Begriffes *computational* wird angedeutet, dass hierbei der Schwerpunkt auf der ingenieurstechnischen Anwendung „intelligenter“ Verfahren liegt. Diese Verfahren grenzen sich gegenüber bis dahin üblichen KI-Verfahren durch eine stärkere Einbeziehung biologischer Vorbilder anstelle der Modellierung mit logischen Kalkülen ab. Das führt auch zu einer wesentlich stärkeren Ausnutzung des Faktors *Zufall* und einer geringeren „Vorhersagbarkeit“ der Ergebnisse. Dabei geht es in erster Linie um die Lösung technischer Probleme und nicht so sehr um Rückschlüsse auf die biologischen Vorbilder oder gar auf die geistigen Eigenschaften des Menschen.

Die CI-Forschung besteht im Wesentlichen aus den Unterbereichen *Evolutionäre Algorithmen*, *Künstliche Neuronale Netze* und *Fuzzy Control*. Bezogen auf den Menschen könnte man diese drei Felder mit den Eigenschaften *geschicktes Raten*², *unartikulierbares Wissen*³ und *schwammig formulierte Regeln*⁴ beschreiben. Insbesondere sind es Kombinationen dieser Teilbereiche und theoretische Fundierungen, an denen in der Computational Intelligence gearbeitet wird.

Genetische Programmierung, ein Teilbereich der ursprünglich aus der Optimierung stammenden *Evolutionären Algorithmen*, ist ein Adaptationsalgorithmus auf symbolischer Ebene. Im Gegensatz zum Lernen mit Künstlichen Neuronalen Netzen lassen sich die Ergebnisse dieses Verfahrens zu jedem Zeitpunkt verstehen und interpretieren. Die Repräsentation ist ein *Programm*, dessen Be-

1 Mit dem RoboCup beschäftigte sich auch die Projektgruppe 340: „Fußball spielende kooperative Multiagentensysteme“.

2 Entspricht den auf „Versuch und Irrtum“ ausgelegten Suchvorgängen der Evol. Algorithmen.

3 Entspricht den menschlichen Gehirnstrukturen nachempfundenen Neuronalen Netzen.

4 Entspricht den sich überschneidenden Regeln in Fuzzy-Control-Systemen.

standteile zu Beginn zufällig zusammengesetzt und im Laufe der simulierten Evolution durch *Genetische Operatoren* – an biologische Vorbilder angelehnte Mechanismen: Reproduktion, Mutation und Rekombination – verändert werden. Eine Bewertung der Tauglichkeit (*Fitness*) eines solchen *Individuums* (Programm) in jeder *Generation* der Evolutionsschleife ermöglicht die Selektion der Individuen, die sich als bessere Problemlöser erwiesen haben. Ein proportional größerer Anteil dieser Programme kann zur Annäherung an das vorgegebene Bewertungsziel führen. Dabei spielt das richtige Maß zwischen der Beibehaltung auch „schlechter“ Lösungen (um evtl. von dort aus an andere Stellen des Suchraumes zu gelangen) und dem Selektionsdruck zur Bevorzugung „guter“ Lösungen eine entscheidende Rolle. Im Vergleich zu anderen traditionellen Lernverfahren der KI versuchen Evolutionäre Algorithmen in sehr viel größerem Maße den „Spagat“ zwischen großflächiger Erkundung des Suchraums und Erhaltung der bisherigen (Zwischen-)Lösungen zu meistern.

2.3 Methode und Zielsetzung

Abseits der Fragen, ob es sich beim Schachspielen um spezifisch menschliche Fähigkeiten handelt, und ob oder wie weit Schachprogramme denken können oder müssen, ergeben sich anhand dieses Spiels interessante Fragestellungen für die Informatik. Dabei liegt das Grundproblem in der Frage, wie in einem Suchraum, der durch seine Größe ein sequentielles Abarbeiten nicht zulässt, nur die „interessanten“ Möglichkeiten näher betrachtet werden können. Menschen benutzen dazu entweder ihre erworbenen Erfahrungen, ihr gelerntes Wissen oder aber ihre Intuition. Hier stellt sich die Frage, inwieweit sich diese Vorgänge formalisieren lassen, bzw. wie für den Computer adäquate Algorithmen formuliert werden können. Die Lösung von Suchproblemen in hochdimensionalen Räumen ist bisher sowohl in Bezug auf Schach, als auch auf andere Probleme des Maschinellen Lernens und der Optimierung auf unterschiedliche Weise angegangen worden. Das Spektrum reicht dabei von Heuristiken, die nach vorgegebenen Regeln die richtigen Pfade in Entscheidungsbäumen finden sollen, bis zu sich selbst verändernden Computerprogrammen, die sich an die gesuchte Lösung anpassen sollen.

Die Wahl von *Genetischer Programmierung* für die Bearbeitung des Suchraumes gültiger Schachzüge ist im Wesentlichen in zwei bereits kurz skizzierten Eigenschaften begründet, der *Suchmächtigkeit* und der Art der *Repräsentation* der Ergebnisse. Die *Suchmächtigkeit* ergibt sich nicht nur aus der Ausnutzung des Zufalls, sondern auch aus der Möglichkeit durch Rekombination Teillösungen als *Bausteine* zu neuen (Teil-)Lösungen „zusammenstecken“ zu können.⁵ Sollte diese Eigenschaft zum Tragen kommen, könnten *Strategien* sukzessive entwickelt werden. Das von uns gewählte modulare und aus aufeinander aufbauenden Stufen bestehende Design des GP-Systems (Siehe auch Kapitel 6) sollen diese grundsätzliche Eigenschaft von GP unterstützen. Die *Repräsentation* der erzeugten Lösungen als lesbare Programme soll darüberhinaus ein Verständnis der

⁵ Das besagt zumindest die *Building Block Hypothese*. Siehe auch 3.3.3.5.

Vorgänge erleichtern, Verbesserungen ermöglichen und auch Aufschluss über die entstandenen Computerstrategien geben.

Ein weiterer Grund, einen Evolutionären Algorithmus zur Bearbeitung dieser Aufgabe gewählt zu haben, ist der Reiz, ein schwieriges Problem der Logik mit naturanalogen und zufallsbasierten Vorgehensweisen zu konfrontieren. Erstaunlicherweise führten ja gerade logikbasierte Ansätze des Maschinellen Lernens nicht zu einem intelligenten Schachprogramm. Wie bereits zu Beginn erwähnt, nutzt der menschlicher Spieler auch so „unlogische“ oder „unscharfe“ Hilfsmittel, wie *Intuition*, *Erfahrung* oder *Mustererkennung*. Es ist die Vermutung zu klären, ob Verfahren mit „eingebauten Unsicherheiten“ eher dazu in der Lage sind, diese menschlichen Eigenschaften nachzuahmen.

Sollte es tatsächlich gelingen, Schachprogramme zu *evolvieren*, die über eine akzeptable Spielstärke verfügen, so kann das ein Schritt auf dem Weg zur automatischen maschinellen Erzeugung von *Strategien* bei der Lösung schwieriger Probleme sein. Nicht nur die Tatsache, *dass* so ein Programm Schach spielen kann und vielleicht sogar gewinnt, ist interessant, sondern *wie* es spielt und auf eine Lösung *gekommen* ist. Damit eröffnen sich Anwendungsfelder, die weit über eine akademische Spielerei hinausgehen.

2.4 Überblick

Im folgenden Kapitel 3 werden die Grundlagen für die folgenden Kapitel – die grundsätzlichen Aspekte des Schachspiels (insbesondere auf Computern), Maschinelles Lernen und Genetische Programmierung – erläutert. Kapitel 4 beinhaltet die Vorstellung der von uns benutzten Entwicklungsumgebungen und Werkzeug. Hier sind besonders die Programmbibliothek SYSGP, sowie das Datenbanksystem Oracle hervorzuheben. Das Kapitel 5 gibt einen Überblick über die Herangehensweise an die zu bewältigenden Aufgaben. Anschliessend wird im Kapitel 6 die Architektur unseres Schach-GP-Systems vorgestellt, wobei das Konzept und das Systemdesign näher beschrieben werden. Kapitel 7 stellt die durchgeführten Analysen vor und fasst die erzielten Ergebnisse zusammen. Das Kapitel 8 ist ein zusammenfassendes Fazit mit einem Ausblick auf mögliche weitere Arbeiten und Verbesserungsmöglichkeiten.

3 Grundlagen

3.1 Das Schachspiel

„Das Schachspiel übertrifft alle anderen Spiele so weit wie der Chimborasso einen Misthaufen.“

(Arthur Schopenhauer)

3.1.1 Zur Geschichte

Auf der Suche nach den Spuren des Schachspiels findet man in der Etymologie einen mächtigen Reisegefährten. Es wird angenommen, dass das Schachspiel zwischen 500 und 100 v. Chr. in Indien entwickelt wurde. Seinen Namen erhielt es in Persien von dem Wort „šāh“, das für „Schah“ bzw. „König“ steht. Persisch ist auch immer noch die Bezeichnung des Turms im Englischen („rook“ bzw. pers. „rukh“). Von den Arabern wurde das Spiel über Nordafrika nach Spanien gebracht, von wo es um 1000 nach Mitteleuropa kam. In Deutschland wurde es um 1050 erstmals erwähnt. Das Schachspiel war anfangs nur in höfischen Kreisen bekannt. Um 1300 fand es aber Verbreitung in allen Volksschichten. Im 14. Jh. wurde das einfarbige, nur durch Linien in Quadrate eingeteilte Schachbrett durch das heute noch übliche, in schwarze und weiße Felder unterteilte Schachbrett ersetzt. Im letzten Viertel des 15. Jh. wurden verschiedene Neuerungen, wie z. B. die Rochade und die Bauernumwandlung, eingeführt. Außerdem erhielten verschiedene Figuren – insbesondere die Dame – eine größere Beweglichkeit, was einen bedeutenden Einfluss auf die Spieldynamik hatte.

Die heute bedeutenden Schachnationen im Osten Europas erreichte das Schachspiel nicht über das maurische Spanien, sondern unmittelbar aus Indien und Persien. Noch heute finden sich das „Boot“ und der „Elephant“ des indischen Schachs in den russischen Figurenbezeichnungen „ladja“ (Turm) und „slon“ (Läufer). Hanseatische Nowgorodfahrer überbrachten die europäischen Regelmodifikationen [STR96].

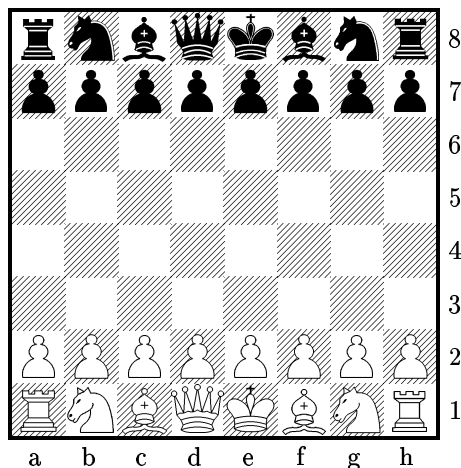
1866 krönte sich der Österreicher Wilhelm Steinitz selbst zum Weltmeister. Seit dieser Zeit ist Schach Sport. Seinen Aufstieg zum Leib-und-Magen-Sport des Kommunismus verdankte das Schachspiel dem gefälschten Lenin-Zitat „Schach ist eine Gymnastik des Geistes“.

Erst in jüngster Zeit wurde ein neues Kapitel in der Historie des Schachspiels begonnen: Schach ist die beliebteste Online-Sportart im World Wide Web.

3.1.2 Regeln

Das Schachbrett wird zwischen den beiden Spielern so aufgelegt, dass jeder Spieler in seiner rechten unteren Ecke ein weißes Feld findet. Die 64 Felder sind in senkrechte *Linien* (Bezeichnung a–h) und waagerechte *Reihen* (Bezeichnung 1–8) aufgeteilt. Ein Spieler führt die weißen Schachfiguren – korrekt als *Schachsteine* bezeichnet – und der andere die schwarzen. Die weißen Bauern stehen zu Beginn auf den Feldern der zweiten Reihe und die schwarzen Bauern auf denen

der siebten. Auf den Feldern der ersten wie der letzten Reihe stehen in dieser Reihenfolge: Turm, Springer, Läufer, Dame, König, Läufer, Springer und Turm. Die weiße Dame steht also zu Beginn des Spiels immer auf einem weißen Feld und die schwarze Dame auf einem schwarzen Feld.



Alle Steine außer den Bauern dürfen im Spielverlauf *schlagen* wie sie ziehen, wobei der in der Bahn stehende Stein des Gegners entfernt wird, und an dessen Stelle der schlagende Stein gesetzt wird. Die Bauern dagegen schlagen ein Feld schräg nach links oder rechts vorwärts, obwohl sie nur gerade vorwärts ziehen dürfen.

Das Ziel des Spiels ist es, den gegnerischen König *matt* zu setzen, d. h. ihn anzugreifen und ihm gleichzeitig jede Bewegungsfreiheit zu nehmen, so dass er kein Feld mehr zur Verfügung hat, das vom Gegner nicht beherrscht würde. Wenn ein Spieler keinen Zug mehr ausführen kann, ohne seinen König einer direkten Bedrohung auszusetzen, so tritt ein *Patt* ein und das Spiel endet *remis* (unentschieden). Wenn kein Spieler mehr die Möglichkeit hat, den gegnerischen König matt zu setzen, endet die Partie ebenfalls remis, genauso wie bei dreimaliger Wiederholung einer Stellung, oder wenn sich die Spieler auf ein Remis einigen.

Die Schachsteine haben unterschiedlichen Wert und unterschiedliche Gangarten: Die *Bauern* können bei jedem Zug nur ein Feld (von der Grundstellung aus auch zwei Felder) gerade vorwärts gehen. Die *Türme* bewegen sich gradlinig beliebig weit vorwärts, rückwärts oder seitwärts, die *Läufer* ebenso auf den Diagonalen des Brettes. Die *Dame* kann bei jedem Zug wie der Turm oder wie der Läufer ziehen. Der *König* geht nur auf ein benachbartes Feld vorwärts, rückwärts, seitwärts oder schräg⁶. Die *Springer* bewegen sich zwei Felder in gerader und dann ein Feld in schräger Richtung. Nur die Springer dürfen andere Steine überspringen. Wenn ein Bauer auf die letzte Reihe kommt, verwandelt er sich in einen beliebigen anderen Stein (König ausgenommen) seiner Farbe [JKA83].

⁶ Eine Ausnahme bildet die *Rochade*, die vom König und einem Turm gleichzeitig ausgeführt wird.

Im Turnierschach wird mit speziellen *Schachuhren* gespielt. Als Richtschnur gelten drei Minuten Bedenkzeit pro Zug.

3.1.3 Die Schachpartie und ihre Phasen

Eine Schachpartie besteht in idealisierter Form aus drei Phasen, die man als *Eröffnung*, *Mittelspiel* und *Endspiel* bezeichnet. Da diese Phasen nicht wirklich definiert sind, lassen sich die Übergänge zwischen ihnen nur erahnen.

Eine Schachpartie besteht aber nicht obligatorisch aus diesen drei Phasen, vielmehr kann sie durch *Fehler* zu fast jedem beliebigen Zeitpunkt abrupt beendet werden. Insbesondere im Mittelspiel werden etwa die Hälfte aller Partien entschieden.

Die erste Phase ist die *Eröffnung*. In ihr verlassen die Steine ihre Startpositionen, um günstige Ausgangsbasen für das sich anschließende Mittelspiel zu besetzen. Seit etwa 1920 hat sich jedoch die Ansicht durchgesetzt, die Eröffnung sei nicht anders zu behandeln als das Mittelspiel. In der Praxis sieht man in der Eröffnung oft das mechanische Abspulen memorierter Zugfolgen, die sich bewährt haben. Der erste eigenständige Zug am Ende einer solchen Zugfolge markiert dann in der Regel den Übergang zum Mittelspiel.

Das *Mittelspiel* beinhaltet den eigentlichen Kampf um Vorteile. Diese Vorteile können sich sowohl in Mehrbesitz an Spielsteinen als auch in Kontrolle über bedeutende Areale des Schachbretts äußern.

Im *Endspiel* versucht der im Vorteil befindliche Spieler, den im Mittelspiel errungenen Vorteil, der in der Regel von geringem absoluten Wert ist, durch „Rauskürzen“ neutral zu bewertender Spielelemente relativ zu vergrößern. Daher ist es charakteristisch für Endspiele, dass von den ursprünglich 32 Steinen einer Schachpartie nur ein Bruchteil am Endspiel beteiligt ist. Als Zeitpunkt des Übergangs vom Mittel- zum Endspiel kann man oft den Damentausch annehmen.

Das Spiel in den einzelnen Partiephasen ist sehr verschieden. Wesentlicher Grund dafür ist die Gangart der Bauern, die als einzige Schachsteine nur in eine Richtung, und nicht symmetrisch in mehrere, ziehen dürfen. Im Mittelspiel verleihen sie einer Schachposition ein „Skelett“, welches die Lager beider Parteien stützt und überdies voneinander trennt. Im Endspiel fehlen die meisten Bauern, und die Lager haben sich aufgelöst. Weiße und schwarze Steine können sich bunt gemischt in jeder Ecke des Brettes befinden. Symmetrien machen sich viel stärker bemerkbar als noch im Mittelspiel, weil die Bauern als „Symmetriebrecher“ weitestgehend verschwunden sind.

Für den Schachspieler ist dies von Bedeutung, weil in diesen beiden Partiephasen unterschiedliche Denkmuster Anwendung finden, auf die im folgenden eingegangen werden soll.

3.1.4 Menschenschach

Ein jeder hat schon einmal eine Partie Schach gespielt, wenn auch vielleicht eher „unter Freunden“. Der Otto-Normal-Schächer schätzt die Erfolgsaussichten einzelner weniger Züge intuitiv ab und wählt dann einen der betrachteten Züge.

Wie man sehen wird, verhält es sich unter professionellen Schachspielern selten anders.

Die erfolgreiche Zugwahl eines Schachspielers umfasst zwei wesentliche Komponenten:

- die Stellungsbeurteilung.
Bei der Stellungsbewertung zerlegt der menschliche Schachspieler die Stellung in Cluster, welche aus relativen Figurenbeziehungen oder absoluten Figurenstellungen bestehen. Dies stellte der holländische Psychologe Adriaan deGroot bei seinen Untersuchungen in den vierziger Jahren fest [DEG65]. Je mehr und je größere Cluster ein Schachspieler in seinem Gedächtnis bereit hält, und je besser er sie gedanklich verarbeiten kann, desto besser wird er spielen. In diesem Zusammenhang offenbart sich der Unterschied zwischen Mittel- und Endspiel: im Endspiel stehen „freie“ Cluster aus relativen Figurenabhängigkeiten im Vordergrund, während sich die Ballungen im Mittelspiel auf die Bauernstruktur oder auf starre Koordinaten beziehen.
- die Variantenberechnung.
Erstaunlich ist, dass Schachmeister selten weiter vorausschauen als Anfänger, und, dass sie dann meistens auch nur eine Handvoll möglicher Züge prüfen. Der Trick besteht dabei darin, dass die Wahrnehmung des Schachbrettes „gefiltert“ wird: Der Meister sieht bei der Betrachtung einer Schachstellung einfach keine schlechten Züge [HOF85]. Der Schachmeister kann also die einzelnen Äste des Spielbaums schnell ausfindig machen, aber letzteren vor seinem inneren Auge aufzuspannen, ist für ihn genauso schwer wie für den Amateur. In der Variantenberechnung werden auch von Meisterspielern sehr viele Fehler begangen.

3.1.5 Computerschach

Von Kepler und Comenius, von Leibniz und Descartes bis zu La Mettrie und Schopenhauer galt das Beherrschen des Schachspiels als Metapher für das Beherrschen der Welt und für die Denk- und Entscheidungsfähigkeit des Menschen.

Das 1921 von E. Borel erstmals auf die Spieltheorie angewandte Minimax-Prinzip in Verbindung mit der 1937 von Alan Turing eingeführten Turingmaschine ermöglichte einen theoretischen Angriff auf das Selbstverständnis des Menschen durch u.a. Claude Shannon, John von Neumann und Turing selbst. Alan Turing war es auch, der sich 1948 an die praktische Arbeit machte und ein Schachprogramm entwickelte. Dieses Programm, wie auch fast alle folgenden, legte das Gewicht auf die systematische Berechnung von Varianten und nicht auf eine an Nuancen reiche Stellungsbeurteilung. Aber erst 1951 gelang es an der Universität Manchester, eine einfache Schachaufgabe zu lösen. Für die Bewertung einer einzigen Position benötigte die Maschine zwei Sekunden [STR96]. 45 Jahre später sollte der Spezialrechner „Deep Blue“ in einer einzigen Sekunde 10^5 Stellungen betrachten und einen Wettkampf gegen Weltmeister Kasparow gewinnen.

„Der nächste Weltmeister im Schachspiel könnte ein Computer sein, der übernächste einer, der von Computern programmiert wurde.“

([STR96])

3.2 Baumsuchverfahren und Stellungsbewertungen

Die Aufgabe eines *optimalen* Computerschachprogrammes ist es, den *besten* Zug zu generieren. Das Schachspiel ist zumindest so komplex, dass kein Rechner in angemessener Zeit alle Züge im voraus berechnen kann. Deshalb schränkt man die Länge der zu betrachtenden Zugfolgen stark ein. Da durch die Zugfolge nicht zwingend das Ende des Spiels erreicht sein muss, und eine Aussage über die Güte der Zugfolge wünschenswert ist, erfolgt in der Regel eine Stellungsbewertung der erreichten Brettposition.

Nach einer begrifflichen Einführung im folgenden Abschnitt 3.2.1 werden in Abschnitt 3.2.2 Methoden besprochen, deren Zweck die Nichtbeachtung möglichst vieler irrelevanter Zugfolgen ist. Dies sind diejenigen, die von einer anderen Folge *dominiert* werden.

Abschnitt 3.2.3 behandelt zunächst die Laufzeitverbesserung mit geschickten Zugreihenfolgen. Die Effizienz vieler Algorithmen wird von dieser Reihenfolge maßgeblich beeinflusst. Es werden Heuristiken und im Suchprozess erhaltenes Wissen benutzt. Darüberhinaus wird eine populäre Möglichkeit der Effizienzsteigerung durch den Verzicht auf das Zugrecht (Null-move) vorgestellt.

Abschnitt 3.2.4 befasst sich mit der Bewertung der Brettposition, die nach einer Zugfolge erreicht wird. Zum einen geht hier jede Figur, je nach Typ unterschiedlich gewichtet, in einer sogenannten Materialbewertung ein. Eine andere oft gewählte Möglichkeit liegt darin, den Wert der taktischen Positionierung des Figurengefüges zu messen.

Abschließend wird kurz auf mögliche Verbesserungen hingewiesen, die an den etwas risikofreudigeren Leser, der mit *Fehlern* des Mitspielers spekuliert, adressiert sind.

3.2.1 Begriffe

Gegenstand der Betrachtungen sind zunächst 2-Personen-Nullsummenspiele mit vollständiger Information. Somit entspricht bei Spielende der Gewinn des einen Spielers dem Verlust des anderen. Für eine mathematische Definition von 2-Personen-Nullsummenspielen mit vollständiger Information siehe etwa [KÜM92]. Die beiden Spieler werden im Folgenden MAX und MIN genannt. Der MAX-Spieler möchte den Gewinn maximieren, der MIN-Spieler möchte den des MAX-Spielers minimieren. Dem Spieler MAX gehören die weißen Figuren, weshalb er auch das Spiel eröffnen möge.

Die Zugmöglichkeiten, die ein Schachspiel ermöglicht, legen folgende informale Definition eines *Spielbaumes* nahe. Die Knotenmenge ist die Menge der Zugfolgen, die leere Zugfolge eingeschlossen. Anders ausgedrückt entspricht die Menge der Knoten der Familie der Brettstellungen, die durch Zugfolgen beliebiger Länge erreicht werden können. Es gibt demnach verschiedene Knoten für ein

und dieselbe Brettstellung, falls diese auf unterschiedlichen Wegen erreichbar ist. Der Wurzelknoten entspricht der Startaufstellung. Die Knoten seien disjunkt in MAX- und MIN-Knoten zerlegt. MAX-Knoten sind solche, in denen Spieler MAX am Zug ist (z. B. die Wurzel) und MIN-Knoten die übrigen. Diese Definition ist eindeutig, da nach jeder Zugfolge genau ein Spieler am Zug ist. Da letztendlich jeder Knoten genau einer Zugfolge entspricht, lassen sich nun auf triviale Weise Kanten zwischen einigen MAX- und MIN-Knoten definieren. Eine Kante zwischen zwei Knoten A (o.B.d.A ein MAX-Knoten) und B (MIN-Knoten) gibt es genau dann, wenn die zu B gehörende Zugfolge $\Delta\gamma$ entspricht, wobei Δ die Zugfolge gemäß Knoten A ist und die Brettstellung B durch den Zug γ von Spieler MAX aus der Brettstellung A erreicht werden kann. Gibt es eine Kante (a, b) in einem gerichteten Graphen, so wird a als Elter von b , und b als Kind von a bezeichnet. Die Blätter entsprechen gerade den Brettstellungen in denen das Spiel zu Ende ist. Jedem Blatt J lässt sich das Spielergebnis $f(J)$ aus Sicht des MAX-Spielers zuordnen (beim Schach: Gewonnen, Verloren, oder Remis). Somit ist der Spielbaum vollständig definiert.

Der Minimaxwert $\text{minimax}(K)$ eines Knotens K ist wie folgt definiert:

$$\text{minimax}(K) = \begin{cases} f(K) & , K \text{ ist Blatt} \\ \max_{I \text{ Kind von } K} \text{minimax}(I) & , K \text{ ist innerer MAX-Knoten} \\ \min_{I \text{ Kind von } K} \text{minimax}(I) & , K \text{ ist innerer MIN-Knoten} \end{cases}$$

Aus der Definition lässt sich direkt ein rekursiver Ansatz zur Berechnung des Minimaxwertes herleiten. Man beachte, dass eine geschickte Implementierung mit einem Speicherplatz $O(d(G))$ und einer Rechenzeit $O(G)$ auskommt, wobei d die Tiefe eines beliebigen Spielbaumes G ist. Es ist allerdings im Falle von Schach nicht in angemessener Zeit möglich alle Knoten zu expandieren. Abbildung 1 enthält einen Beispielbaum, der mit Minimax bewertet ist. Die hervorgehobenen Kanten zeigen einen Pfad von der Wurzel zu dem Blatt auf, welches den Minimaxwert bestimmt hat. Oft wird die entsprechende Zugfolge auch Hauptvariante genannt. Hauptvarianten sind bestmögliche Zugfolgen und müssen nicht eindeutig sein.

Analog zu [REI89] sei eine MAX-Strategie S eines Spielbaumes G ein Unterbaum, der wie folgt rekursiv definiert ist:

Die Wurzel von G sei die Wurzel von S . Ist ein MAX-Knoten v von G auch in S vorhanden, so hat der entsprechende Knoten in S genau ein Kind, welches als solches auch in G existiert. Ist ein MIN-Knoten von G auch in S , so enthält S genau die Kinder, die in G vorkommen mit zugehörigen Kanten.

Eine Spielstrategie hat unabhängig von der Reaktion des Mitspielers stets genau eine Antwort parat. Mit einem Minimax-Algorithmus erhält man auch eine optimale Strategie, wenn man die lokalen Minimaxwerte an allen Knoten sichert. Im Falle von Schach hat entweder genau einer der beiden Spieler eine Gewinnstrategie oder beide haben eine Remisstrategie.

Aufgrund der Remisregelung beim Schach wird sichergestellt, dass jedes komplette Spiel einer Zugfolge konstanter Länge entspricht. Der Spielbaum hat also

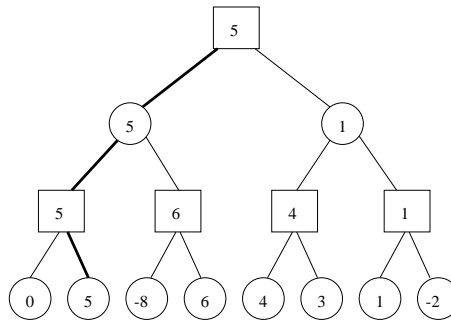


Abbildung 1: Spielbaum mit Minimaxbewertung an den Knoten. MAX-Knoten sind durch Quadrate dargestellt, MIN-Knoten durch Kreise.

konstante Tiefe. Da in jedem Zug maximal 16 Figuren auf maximal⁷ 64 Felder (jeweils) gezogen werden können und es daneben nur wenige Ausnahmезüge gibt, ist auch die Anzahl der Züge stets durch eine Konstante begrenzt. Somit gibt es nur konstant viele Knoten, und einen stets gewinnmaximierenden Mitspieler vorausgesetzt, ist sowohl der optimale nächste Zug, als auch eine optimale Spielstrategie in konstanter Zeit berechenbar. Allerdings ist deren Anzahl mit über 10^{100} unglaublich groß.

3.2.2 Algorithmen zur Berechnung des Minimax-Wertes

Es werden nun Algorithmen vorgestellt, die den Minimaxwert möglichst effizient exakt berechnen. Aufbau und Inhalt sind im Wesentlichen [REI89] entnommen.

3.2.2.1 Minimax und Negamax

Der Minimaxalgorithmus (siehe Algorithmus 1) besteht aus zwei einfachen Prozeduren Min und Max, welche den Minimaxwert eines Min- bzw. Max-Knotens aus den Minimaxwerten seiner Kinder berechnen. Der Algorithmus wird mit `mmMAX(root)` aufgerufen. Die Korrektheit folgt direkt aus der Definition des Minimaxwertes. Der Baum in Abbildung 1 ist mit dem Minimax-Algorithmus bewertet. An den Blättern steht der Wert des Spielausgangs.

Die Prozedur `mmMAX` spiegelt das Verhalten des MAX-Spielers ebenso wieder, wie die Prozedur `mmMIN` die des MIN-Spielers. Die alternierende Zugreihenfolge kommt im wechselseitigen Aufruf zum Ausdruck.

3.2.2.2 Negamax

Der Algorithmus Negamax (siehe Algorithmus 2) verschmilzt die beiden Prozeduren zu einer einzigen. Dabei berechnet er für MAX-Knoten den gleichen Wert, wie der Minimaxalgorithmus, im Falle eines MIN-Knotens aber nur die Negation davon. Der Wert gibt also immer den Gewinn aus Sicht des sich am Zug

⁷ Hierbei handelt es sich nur um eine unscharfe, obere Schranke.


```

mmMAX(position  $K$ )
BEGIN
  IF LEAF( $K$ ) THEN RETURN  $f(K)$ 
  ELSE
    determine successor positions  $K.1, \dots, K.w$ 
    integer  $value = -\infty$ 
    FOR  $j = 1$  TO  $w$  DO
       $value = \max(value, \text{mmMIN}(K.j))$ 
    RETURN  $value$ 
  END

```

```

mmMIN(position  $K$ )
BEGIN
  IF LEAF( $K$ ) THEN RETURN  $f(K)$ 
  ELSE
    determine successor positions  $K.1, \dots, K.w$ 
    integer  $value = \infty$ 
    FOR  $j = 1$  TO  $w$  DO
       $value = \min(value, \text{mmMAX}(K.j))$ 
    RETURN  $value$ 
  END

```

Algorithmus 1: Minimax-Verfahren

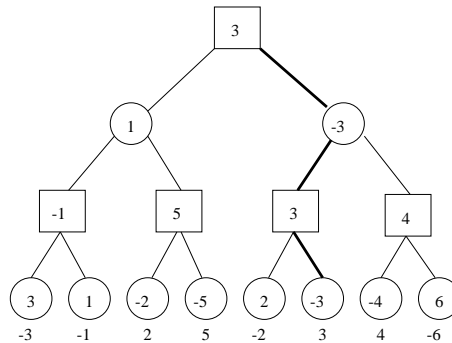


Abbildung 2: Spielbaum mit NegaMaxbewertung an den Knoten. Der jeweilige Spielausgang ist unter den Blättern abzulesen.

befindlichen Spielers an. Unabhängig welcher Spieler dies ist, wird dieser stets versuchen, den Zug auszuführen, mit dem ein Knoten mit minimalen Wert erreicht wird. Das sind solche Knoten, bei denen der Gegenspieler möglichst wenig gewinnt bzw. möglichst viel verliert. Dies erreicht er auch dann, wenn er die Werte der direkten Nachfolger negiert und das Maximum von den erhaltenen Werten als seinen Wert annimmt.

```

NegaMax(position  $K$ )
BEGIN
  IF LEAF( $K$ ) THEN
    IF MAX-Node( $K$ ) RETURN  $f(K)$ 
    ELSE RETURN  $-f(K)$ 
  ELSE
    determine successor positions  $K.1, \dots, K.w$ 
    integer  $value = -\infty$ 
    FOR  $j = 1$  TO  $w$  DO
       $value = \max(value, -\text{NegaMax}(K.j))$ 
    RETURN  $value$ 
END

```

Algorithmus 2: NegaMax-Verfahren

Abbildung 2 zeigt eine NegaMaxbewertung an einem Beispielbaum. Für MAX-Knoten erhält dieser die gleiche Bewertung, wie bei Minimax. Die Hauptvariante ist hervorgehoben.

Die Algorithmen Minimax und NegaMax haben gemeinsam, dass sie den gesamten Baum expandieren. In dem folgenden Abschnitt wird sich herausstellen, dass dies zur Berechnung des exakten Wertes gar nicht nötig ist. Vielmehr können große Teile des Baumes unter bestimmten Umständen einfachen ausgelassen (abgeschnitten, engl. cut-off) werden.

3.2.2.3 Alpha-Beta

Der Alpha-Beta-Algorithmus (siehe Algorithmus 3 und 4) ist sicherlich das bekannteste „Suchbaum-Reduktionsverfahren“ ([REI89]). Der ursprüngliche Alpha-Beta-Algorithmus, der auf Ideen von McCarthy aufbaut und von Newell, Shaw und Simon veröffentlicht wurde, ist von Knuth und Moore als Branch & Bound bezeichnet worden, auch wenn sich die Bedeutung von Branch & Bound zu einer ganzen Algorithmenklasse gewandelt hat. Der hier vorgestellte, vollständige Alpha-Beta-Algorithmus expandiert den Baum in einer festen Reihenfolge, lässt aber alle Unterbäume aus, die nach momentanem Informationsstand das Endergebnis nicht beeinflussen können.

```

1   $\alpha\beta_{\text{MAX}}$ (position  $K$ ;integer  $\alpha, \beta$ )
2  BEGIN
3    integer  $j, w, value$ ;
4    determine successor positions  $K.1, \dots, K.w$ 
5    IF LEAF( $K$ ) THEN RETURN  $f(K)$ 
6     $value = \alpha$ 
7    FOR  $j = 1$  TO  $w$  DO
8      BEGIN
9         $value = \max(value, \alpha\beta_{\text{MIN}}(K.j, value, \beta))$ 
10       IF  $value \geq \beta$  THEN RETURN  $value$            //  $\beta$ -Schnitt
11      END
12    RETURN  $value$ 
13  END

```

Algorithmus 3: Max-Teil des $\alpha\beta$ -Verfahrens

```

1   $\alpha\beta_{\text{MIN}}$ (position  $K$ ;integer  $\alpha, \beta$ )
2  BEGIN
3    integer  $j, w, value$ ;
4    determine successor positions  $K.1, \dots, K.w$ 
5    IF LEAF( $K$ ) THEN RETURN  $f(K)$ 
6     $value = \beta$ 
7    FOR  $j = 1$  TO  $w$  DO
8      BEGIN
9         $value = \min(value, \alpha\beta_{\text{MAX}}(K.j, \alpha, value))$ 
10       IF  $value \leq \alpha$  THEN RETURN  $value$            //  $\alpha$ -Schnitt
11      END
12    RETURN  $value$ 
13  END

```

Algorithmus 4: Min-Teil des $\alpha\beta$ -Verfahrens

Die Information, die beim Alpha-Beta-Verfahren aus vorherigen Knotenexpansionen aufgehoben wird, besteht aus einer unteren Schranke α und einer

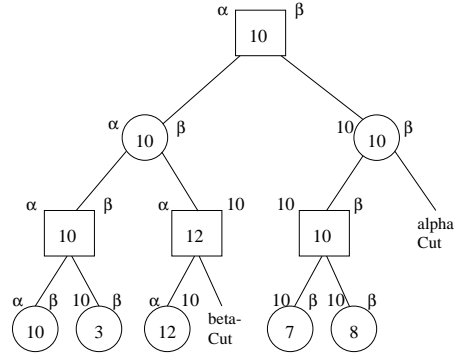


Abbildung 3: Spielbaum mit Alpha-Beta-Bewertung an den Knoten

oberen Schranke β für den zu berechnenden Minimaxwert. Dem MAX-Spieler ist also ein Gewinn von α sicher, umgekehrt muss der MIN-Spieler höchstens β zahlen. Im Laufe des Algorithmus wird sich das durch die Werte ergebende *Suchfenster* verkleinern. Der erste Aufruf lautet $\alpha\beta_{\text{MAX}}(\text{root}, -\infty, \infty)$. In Abbildung 3.2.2.3 sind die Aufrufparameter jeweils links und rechts neben den Knoten zu finden. Dabei seien a und b hinreichend klein, bzw. groß. Im Folgenden wird die Korrektheit bewiesen. Eine anschauliche Begründung findet der Leser in [REI89].

Knoten K werde mit (K, α, β) aufgerufen

- \exists MAX-Knoten A , der echter Vorgänger von K ist, (1)
der mindestens α zurückliefert, falls $\text{depth}(K) \geq 1$
- \exists MIN-Knoten I , der echter Vorgänger von K ist, (2)
der maximal β zurückliefert, falls $\text{depth}(K) \geq 2$

Beweis der Zwischenbehauptung

Sei G ein beliebiger Spielbaum. Die Behauptung wird mittels Induktion über die Ebenen e des Baumes G gezeigt. Für $e = 2$, also Tiefe 1, ist nur Teil (1) zu zeigen. Als Knoten A kommt nur die Wurzel in Frage. Der β -Wert ist unverändert (∞), falls der α -Wert noch nicht erhöht wurde, gilt die Aussage, da der an der Wurzel berechnete Wert endlich ist. Andernfalls gibt es ein Kind mit dem Wert α , da der *value*-Wert in den Zeilen 7 bis 11 der zur Wurzel gehörenden FOR-Schleife, als Aufrufparameter dient, und das Maximum der bereits betrachteten Rückgabewerte annimmt (anfangs $-\infty$). Da der lokale *value*-Wert nicht kleiner wird und mindestens α ist, ist die Behauptung für $e = 2$ bewiesen. Für $e = 3$ erfolgen Aufrufe der Form $\alpha\beta_{\text{MAX}}(\text{Wurzel.i.j}, \alpha, \beta)$, wobei der α -Wert nur bei dem $\alpha\beta_{\text{MIN}}(\text{Wurzel.i}, \alpha, \beta)$ -Aufruf, der β -Wert nur bei dem $\alpha\beta_{\text{MAX}}(\text{Wurzel.i.j}, \alpha, \beta)$ -Aufruf geändert werden kann. Es wird gezeigt, dass die Wurzel Eigenschaft (1) erfüllt. Der *value*-Wert wird während der FOR-Schleife (7-11) nur erhöht, wenn ein *returnValue*(wurzel.i) größer ist als er selbst. Da er anfangs mit $-\infty$ initialisiert wird, beschreibt er stets das Maximum der zurückgegebenen Werte bereits betrachteter Kinder. Da der *value*-Wert nicht verkleinert wird, liefert

die Wurzel entsprechend der MAX-Knoten-Eigenschaft mindestens den Wert α zurück. Analog lässt sich über den *value*-Wert des Kindes `wurzel.i` argumentieren. Dieser ist mindestens β und `wurzel.i` erfüllt somit die Eigenschaft (2). Gelte nun die Behauptung für alle $i \leq e$. Sei E ein Knoten der Ebene e , E_i sein i -ter Nachkomme, welcher mit (E_i, α_i, β_i) aufgerufen werde. Wir zeigen nur Teil (1), der andere folgt analog. Für $\alpha_i = \alpha$ ist nichts zu zeigen, da der zu K gehörige A -Knoten bereits die Ungleichung erfüllt. Sei also $\alpha_i > \alpha$, insbesondere ist dann K MAX-Knoten. Wieder gibt es nun ein bereits betrachtetes Kind $K.i$, welches einen Wert der Größe α zurückgeliefert hat. Damit ist der *value*-Wert am Knoten K bei der Rückgabe mindestens α . K genügt also den Anforderungen, die Behauptung ist für $e + 1$ richtig.

Für Blätter und die Wurzel verhält sich der Algorithmus wie Minimax. Wenn ein sonstiger Knoten K mit einem (K, α, β) -Aufruf erreicht wird, dann ist aufgrund von Zwischenbehauptung 1 sicher, dass es *Alternativen* gibt, mit denen der MAX-Spieler mindestens Wert α sichert und der MIN-Spieler maximal Wert β zahlen muss. Dieser Knoten verursacht also gewiß nicht einen Minimaxwert außerhalb des Suchfensters. Ohne eine Fehlberechnung an der Wurzel (!) zu riskieren, kann mindestens der Wert α im Falle eines MAX-Knotens, der Wert β sonst zugeordnet werden. Die Aufrufe innerhalb der Zeilen 7 bis 11 berücksichtigen die einzelnen Zugmöglichkeiten. Im Falle eines MAX-Knotens bleibt der β -Wert konstant. Der α -Wert ändert sich nur, wenn eine bessere Zugfolge gefunden wurde, er kann sich insbesondere innerhalb der Schleife nur erhöhen. Wird der α -Wert mindestens so groß wie β , so kann die weitere Knotenexpansion abgebrochen werden. Der Wert, den der Knoten zurückgibt, ist ja dann mindestens β , für den MIN-Spieler gibt es aber eine Möglichkeit höchstens β zu zahlen. Der MIN-Knoten kann analog betrachtet werden. Da die Parametergrenzen und die Schnitte keine echten Einschränkungen bewirken, die Maximum- und Minimumbildungen im Prinzip der Berechnung des Minimaxwertes entsprechen und es zumindest für die Berechnung des Minimaxwertes an der Wurzel auch möglich ist, statt eines Wertes außerhalb eines Suchfensters den alpha- oder beta-Wert zurückzuliefern, arbeitet der Algorithmus korrekt. \square

Es wird also der Minimaxwert an der Wurzel korrekt berechnet. Man beachte, dass der von einem inneren Knoten zurückgegebene Wert nicht dem vom Minimaxalgorithmus berechneten Wert entsprechen muss.

In Algorithmus 5 sei noch die im Wesentlichen [REI89] entnommene Negamax-Version genannt, welche die beiden Prozeduren in einer vereint. Der *value*-Wert entspricht dabei stets dem möglichen Gewinn aus Sicht des Spielers der gerade am Zug ist. Deswegen wird im Falle eines Blattes auch nicht mehr $f()$, sondern $g()$ zurückgegeben, welches wie folgt definiert ist:

$$g(K) = \begin{cases} f(K) & \text{falls } K \text{ MAX-Knoten} \\ -f(K) & \text{falls } K \text{ MIN-Knoten} \end{cases}$$

Wie beim Minimaxalgorithmus besucht der $\alpha\beta$ -Algorithmus jeden Knoten maximal einmal, der Programmrumpf kostet $O(1)$ Rechenzeit. Die Laufzeit ist also $O(V)$, wobei V die Anzahl der Knoten ist. Durch die Schnitte fallen aber ei-

```

1   $\alpha\beta$ (position  $K$ ; integer  $\alpha, \beta$ )
2  BEGIN
3    integer  $j, w, value$ ;
4    determine successor positions  $K.1, \dots, K.w$ 
5    IF LEAF( $K$ ) THEN RETURN  $g(K)$ 
6     $value = \alpha$ 
7    FOR  $j = 1$  TO  $w$  DO
8      BEGIN
9         $value = \max(value, -\alpha\beta(K.j, -\beta, -value))$ 
10       IF  $value \geq \beta$  THEN RETURN  $value$           // Schnitt
11      END
12    RETURN  $value$ 
13  END

```

Algorithmus 5: $\alpha\beta$ -Verfahren (Negamaxversion)

nige Unterbäume weg. Wie sehr sich die Laufzeitverbesserung durch die Schnitte auswirkt, ist von Faktoren wie Verzweigungsgrad, Blattbeschriftung, also vom einzelnen Spiel, abhängig. Beim Schach ist in etwa ein Verzweigungsgrad von 40 gegeben. Kann z. B. nach der Evaluierung des zweiten Kindes ein Schnitt stattfinden, so brauchen die restlichen 38 nicht betrachtet zu werden.

3.2.2.4 Alpha-Beta-Verbesserung

Für den $\alpha\beta$ -Algorithmus gibt es verschiedene Verbesserungsansätze. Im folgenden werden wichtige vorgestellt.

Suchfenstertechnik:

Je kleiner das durch das Parametertupel (α, β) gegebene Suchfenster ist, desto größer sind die Schnittmöglichkeiten. Die Laufzeit des Algorithmus hängt im Wesentlichen von der Anzahl und Stärke der Schnitte ab. Anstatt den Algorithmus mit $(-\infty, \infty)$ aufzurufen, besteht die Möglichkeit, ein eingeschränktes Suchfenster (α, β) zu benutzen. Ist das Ergebnis im Intervall (α, β) , so stimmt es mit dem Minimaxwert überein. Liefert der Algorithmus eine Intervallgrenze, z. B. α zurück, ist eine Wiederholungssuche erforderlich. Diese erfolgt dann mit dem Suchfenster $(-\infty, \alpha + 1)$. Prinzipiell ist auch das Fenster $(-\infty, \alpha)$ zulässig. In jedem Fall wird der Minimax-Wert zurückgeliefert. Implementierungstechnisch ist die zweite Variante vorzuziehen, da eine Wiederholungssuche genau dann zu tätigen ist, wenn eine Intervallgrenze ermittelt wurde. Die Suchfenstertechnik kann rekursiv angewandt werden.

F-Verbesserung:

Die „failsoft“-Verbesserung ist eine leichte Modifikation des $\alpha\beta$ -Algorithmus (siehe Algorithmus 6). Bei der eben besprochenen Suchfenstertechnik kann es vorkommen, dass der im Wesentlichen komplette Baum mit dem unbefriedigenden Ergebnis expandiert wird, dass eine von beiden Intervallgrenzen zurückgeliefert

wird. Da der Algorithmus oft aufgerufen wird, ohne zu wissen, ob das Ergebnis innerhalb der Intervallgrenzen liegt, wäre eine genauere Abschätzung jenseits dieser wünschenswert. Um dies zu erreichen wird *value* wie bei Negamax mit $-\infty$ initialisiert. Damit wird das Maximum der berechneten *value*-Werte an den Kindern zurückgeliefert, auch wenn es kleiner als α ist. Zusätzlich muss die Programmzeile

$$\begin{aligned} value &= \max(value, -\alpha\beta(K.j, -\beta, -value)) \\ &\quad \text{zu} \\ value &= \max(value, -\alpha\beta(K.j, -\beta, -\max(value, \alpha))) \end{aligned}$$

abgeändert werden. Auf diese Weise wird eventuell ein Wert außerhalb des Suchfensters berechnet, der zu effizienteren Wiederholungssuchen führt. Die Schnitte sind jedoch genau die gleichen, weshalb auch die Wiederholungssuche zur exakten Berechnung im eben angesprochenen Fall durchgeführt werden muss.

```

1  F- $\alpha\beta$ (position  $K$ ; integer  $\alpha, \beta$ )
2  BEGIN
3    integer  $j, w, value$ ;
4    determine successor positions  $K.1, \dots, K.w$ 
5    IF LEAF( $K$ ) THEN RETURN  $g(K)$ 
6     $value = -\infty$ 
7    FOR  $j = 1$  TO  $w$  DO
8      BEGIN
9         $value = \max(value, -F-\alpha\beta(K.j, -\beta, -\max(value, \alpha))$ 
10       IF  $value \geq \beta$  THEN RETURN  $value$           // Schnitt
11      END
12    RETURN  $value$ 
13  END
```

Algorithmus 6: F- $\alpha\beta$ -Verfahren (Negamaxversion)

L-Verbesserung:

Die weit verbreitete L-Verbesserung, berechnet nicht den exakten Minimaxwert, sondern lediglich den besten Zug. Dabei werden in einem Knoten mit m Nachkommen zunächst die ersten $m-1$ evaluiert und anschließend der letzte mit dem Suchfenster $(value, value+1)$ aufgerufen. Auf diese Weise wird der Aufruf schnell abgearbeitet und es kann entschieden werden, ob der Minimaxwert kleiner, gleich oder größer als $value$ ist. Genau im letzten Fall ist der m -te Zug der beste. Die L-Verbesserung (last move improve) kann im Gegensatz zur F-Verbesserung nicht rekursiv angewendet werden, da eine exakte Berechnung nicht immer vorliegt.

Nullfenstertechnik:

Man bezeichnet (α, β) als Nullfenster, wenn β genau um eins größer ist als α . Bei der F-Verbesserung beispielsweise wird das letzte Kind mit $(value, value+1)$ aufgerufen. Als Ergebnis ist zumindest klar, ob der Minimaxwert maximal den

Wert *value* annimmt oder ob er echt größer ist. Der Aufruf verläuft sehr effizient, da das Fenster so klein ist.

Bei der Nullfenstertechnik wird allgemein das erste Kind des Knotens auf übliche Weise evaluiert. Anschließend werden nachfolgende Unterbäume mit diesem Wert verglichen, gegebenenfalls muss eine genauere Berechnung erfolgen. Da dieser effiziente boolesche Test oft ausreicht und somit mit einem Vergleich am nächsten Unterbaum fortgefahren werden kann, erreicht man mit dieser Technik eine Laufzeitverbesserung.

Scout:

Der Scout-Algorithmus [PEA80] besteht aus zwei booleschen Funktionen, die für einen MAX- bzw. einen MIN-Knoten testen, ob der Wert eines Unterbaumes schlechter (also für einen MAX-Knoten kleiner) als ein Referenzwert ist. Eine weitere Funktion (*eval*) berechnet den zu einem Unterbaum gehörenden Wert. Zur Berechnung des Minimax-Wertes an einem MAX-Knoten wird zunächst mittels des ersten Kindes ein Referenzwert ermittelt. Mit diesem und der booleschen Funktion werden die restlichen Unterbäume untersucht. Ist der Referenzwert unterlegen, erfolgt eine genaue Wiederholungssuche mittels *eval*, deren Ergebnis den Referenzwert ersetzt. Auf diese Weise wird der Referenzwert mit jeder Wiederholung größer, die Wahrscheinlichkeit einer weiteren Wiederholungssuche nimmt ab.

Der Scout-Algorithmus kann rekursiv angewendet werden, ein *eval*-Aufruf benutzt also einen *eval*-Aufruf für das erste Kind und vergleicht anschließend die restlichen Unterbäume mit dem Referenzwert. Die Minimaxwert-Berechnung an der Wurzel erfolgt mit *eval(wurzel)*.

„Es konnte gezeigt werden, dass der Scout-Algorithmus in vielen Anwendungen dem $\alpha\beta$ -Verfahren überlegen ist“ [REI89]. Aus implementierungstechnischen Gründen sind für die Praxis solche Versionen relevant, die die beiden booleschen Funktionen und die *eval*-Funktion zu einer Prozedur verschmelzen. Dies ist mit der Nullfenstertechnik möglich, so dass schließlich dem $\alpha\beta$ -Verfahren ähnliche Varianten gefunden wurden, von denen eine im Folgenden vorgestellt werden soll.

Negascout:

Die *eval*-Funktion wird durch ein „offenes Suchfenster“, die booleschen Testfunktionen werden durch ein Nullfenster simuliert. Zunächst wird die Minimax-Version mit zwei Prozeduren vorgestellt. Die Zeilen 7, 9, 10, 11 und 15 sind im $\alpha\beta$ -Verfahren noch nicht vorhanden gewesen, die restlichen stimmen überein. Die neu eingeführte Variable *hivalue* bildet mit *lovalue* das Suchfenster. Auf übliche Weise erfolgt der Aufruf für das erste Kind. Die restlichen Nachfolger werden mit dem stets angepassten Nullfenster durchsucht (Zeile 15). Kompliziert erscheint Zeile 11, in der entschieden wird, ob eine Wiederholungssuche stattfinden soll. Die erste Bedingung ist nach Vorbesprechung klar: Nur wenn ein besserer (im Falle eines MIN-Knotens ein kleinerer) Wert mit *t* vorliegt, kann eine Wiederholungssuche erforderlich sein. Ist *t* allerdings schon außerhalb des Suchfensters, kann ein Schnitt durchgeführt werden, eine Wiederholungssuche ist hierfür nicht

nötig. Der Test $j > 1$ sichert, dass die erste Bewertung, die exakt vorgenommen wird, nicht wiederholt wird. Die letzte Bedingung $depth < d - 1$ berücksichtigt, dass eine Wiederholungssuche bei Tiefe $depth - 1$ das gleiche Ergebnis liefern würde, da die Kinder Blätter sind.

Interessant ist, dass auch die Wiederholungssuche rekursiv Negascout benutzt. Das Negascout-Verfahren [REI83] ist u. a. für Spiele mit hohen Verzweigungsfaktoren zu empfehlen, weswegen ein ähnliches in der Schachmaschine *Belle* zum Einsatz kommt.

Der Negascout-Algorithmus findet sich in den Algorithmen 7 und 8.

```

1  NegaScoutMAX(position  $K$ ; integer  $\alpha, \beta$ )
2  BEGIN
3    integer  $j, w, t, lovalue, hivalue$ ;
4    determine successor positions  $K.1, \dots, K.w$ 
5    IF LEAF( $K$ ) THEN RETURN  $f(K)$ 
6     $lovalue = \alpha$ 
7     $hivalue = \beta$ 
8    FOR  $j = 1$  TO  $w$  DO
9      BEGIN
10        $t = \text{NegaScout}_{MIN}(K.j, lovalue, hivalue)$ ;
11       IF  $t > lovalue$  AND  $t < \beta$  AND  $j > 1$  AND  $depth < d - 1$  THEN
12          $t = \text{NegaScout}_{MIN}(K.j, t, \beta)$            // Wiederholungssuche
13          $lovalue = \max(lovalue, t)$ ;
14         IF  $lovalue \geq \beta$  THEN RETURN  $lovalue$  // Schnitt
15          $hivalue = lovalue + 1$                      // neues Nullfenster
16       END
17     RETURN  $lovalue$ 
18  END

```

Algorithmus 7: Max-Teil des Negascout-Verfahrens

Ohne weiteren Kommentar sei noch die äquivalente Negamax-Version des Negascout-Algorithmus' (siehe Algorithmus 9) vorgestellt, die im nächsten Abschnitt der Ausgangspunkt für Verbesserungen ist.

NegaScout-Verbesserungen:

Die Verbesserungen des NegaScout-Algorithmus zielen auf die Wiederholungssuche ab, die Nullfenstersuche ist ohnehin optimal. Es werden die F-Verbesserung, die L-Verbesserung, die TL-Verbesserung und eine verbesserte Informationsaquisition angesprochen.

F-Verbesserung:

In der Praxis wird der NegaScout-Algorithmus nicht in der Grundversion verwendet, sondern eher in der verbesserten F-NegaScout-Variante. Technisch besteht die Verbesserung in einer genaueren Abschätzung von Werten, die sich außerhalb des Suchfensters befinden, im Rahmen einer Nullfenstersuche. Auf diese

```

1  NegaScoutMIN(position  $K$ ; integer  $\alpha, \beta$ )
2  BEGIN
3    integer  $j, w, t, lovalue, hivalue$ ;
4    determine successor positions  $K.1, \dots, K.w$ 
5    IF LEAF( $K$ ) THEN RETURN  $f(K)$ 
6     $lovalue = \alpha$ 
7     $hivalue = \beta$ 
8    FOR  $j = 1$  TO  $w$  DO
9      BEGIN
10        $t = \text{NegaScout}_{MAX}(K.j, lovalue, hivalue)$ ;
11       IF  $t < hivalue$  AND  $t > \alpha$  AND  $j > 1$  AND  $depth < d - 1$  THEN
12          $t = \text{NegaScout}_{MAX}(K.j, \alpha, t)$  // Wiederholungssuche
13          $hivalue = \min(hivalue, t)$ ;
14         IF  $hivalue \leq \alpha$  THEN RETURN  $hivalue$  // Schnitt
15          $lovalue = hivalue - 1$  // neues Nullfenster
16       END
17     RETURN  $hivalue$ 
18 END

```

Algorithmus 8: Min-Teil des Negascout-Verfahrens

```

1  NegaScout(position  $K$ ; integer  $\alpha, \beta$ )
2  BEGIN
3    integer  $j, w, t, lovalue, hivalue$ ;
4    determine successor positions  $K.1, \dots, K.w$ 
5    IF LEAF( $K$ ) THEN RETURN  $g(K)$ 
6     $lovalue = \alpha$ 
7     $hivalue = \beta$ 
8    FOR  $j = 1$  TO  $w$  DO
9      BEGIN
10        $t = -\text{NegaScout}(K.j, -hivalue, -lovalue)$ ;
11       IF  $t > lovalue$  AND  $t < \beta$  AND  $j > 1$  AND  $depth < d - 1$  THEN
12          $t = -\text{NegaScout}(K.j, -\beta, -t)$  // Wiederholungssuche
13          $lovalue = \max(lovalue, t)$ ;
14         IF  $lovalue \geq \beta$  THEN RETURN  $lovalue$  // Schnitt
15          $hivalue = lovalue + 1$  // neues Nullfenster
16       END
17     RETURN  $lovalue$ 
18 END

```

Algorithmus 9: Negascout-Verfahren (Negamaxversion)

Weise kann die Wiederholungssuche sofort mit optimierten Parametern gestartet werden. Gerade aufgrund dieser Kombination von Nullfenster- und Wiederholungssuche fällt die verbessernde Wirkung beim Negascout-Verfahren erheblich deutlicher aus, als beim $\alpha\beta$ -Algorithmus (siehe Algorithmus 10).

```

1  F-NegaScout(position  $K$ ; integer  $\alpha, \beta$ )
2  BEGIN
3    integer  $j, w, t, lovalue, hivalue$ ;
4    determine successor positions  $K.1, \dots, K.w$ 
5    IF LEAF( $K$ ) THEN RETURN  $g(K)$ 
6     $lovalue = -\infty$ 
7     $hivalue = \beta$ 
8    FOR  $j = 1$  TO  $w$  DO
9      BEGIN
10        $t = -\text{F-NegaScout}(K.j, -hivalue, -\max(lovalue, \alpha))$ ;
11       IF  $t > \max(lovalue, \alpha)$  AND  $t < \beta$  AND  $j > 1$  AND  $depth < d - 2$  THEN
12          $t = -\text{F-NegaScout}(K.j, -\beta, -t)$  // Wiederholungssuche
13          $lovalue = \max(lovalue, t)$ ;
14         IF  $lovalue \geq \beta$  THEN RETURN  $lovalue$  // Schnitt
15          $hivalue = \max(hivalue, t)$  // neues Nullfenster
16       END
17     RETURN  $lovalue$ 
18   END

```

Algorithmus 10: F-Negascoutverfahren (Negamaxversion)

Ganz analog zur bereits vorgestellten F-Verbesserung eines $\alpha\beta$ -Algorithmus, wird in Zeile 6 $lovalue$ mit $-\infty$ initialisiert. Wiederum gibt es Standardanpassungen (Zeile 10 und 15), damit für die Suchfensterparameter kein $lovalue$, der kleiner als α ist, benutzt wird. Dies trifft auch auf die erste Modifikation in Zeile 11 zu ($t > \max(lovalue, \alpha)$). Die zweite Modifikation $depth < d - 2$ berücksichtigt die Tatsache, dass aufgrund der Initialisierung von $lovalue$ auf der drittuntersten Ebene mit $-\infty$, keine Wiederholungssuche mehr erforderlich ist. Das F-NegaScout-Verfahren wird auch oft als NegaScout-Verfahren an sich bezeichnet.

L-Verbesserung:

Last move improvement läßt sich dadurch implementieren, dass im letzten Knoten keine Wiederholungssuche stattfindet. Eine Nullfenstersuche erfolgt sowieso bereits. Es sei noch zu beachten, dass die L-Verbesserung nicht rekursiv anwendbar ist.

TL-Verbesserung:

Die TL-Verbesserung geht mit der Nullsuche über alle Kinder hinweg, auch wenn ein besseres Kind gefunden wurde. Dominiert nur das eine Kind, so ist der beste Zug bestimmt. Denn gibt es nur einen Zug, der besser als der aktuelle ist, dann

ist dieser mit Sicherheit auch gleich der beste. Dominieren mehrere Kinder kann eine Wiederholungssuche eingeleitet werden, sobald der Referenzwert ein zweites Mal übertroffen wird. Dies hat nichts mehr mit „last move improvement“ zu tun, das Ziel ist aber das gleiche geblieben: Ist Dominanz entdeckt, so interessiert nur diese und nicht ihre Stärke.

Die Nullfenstersuche des ersten dominanten Kindes ergibt eine untere Schranke, die als zusätzliche Beschleunigung bei der weiteren Nullfenstersuche ausgenutzt wird. Auch bei der Wiederholungssuche, wird der Wert des ersten Unterbaumes mit der unteren Schranke des zweiten verglichen. Ist letzterer nämlich größer, steht dieser bisher als Favorit fest.

Auch dieses Verfahren ist nicht rekursiv anwendbar, weshalb nur F-Nega-Scout-Aufrufe stattfinden. „Letztlich basiert die TL-Verbesserung auf einer Spekulation, bei der nicht von vornherein klar ist, ob sie sich auszahlt“ [REI89].

Verbesserte Informationsaquisition:

Im Gegensatz zum $\alpha\beta$ -Verfahren betrachten die NegaScout-Varianten aufgrund der Wiederholungssuchen ein und denselben Knoten mehrfach. Während der Suchphase mit dem Nullfenster könnten z. B. alle Knotennachfolger abgespeichert werden, welche den Minimaxwert nicht beeinflussen. Dazu können auch Expansionsreihenfolgen gemäß der bei der Nullfenstersuche für MIN-Knoten ermittelten oberen Schranken festgelegt werden.

Die Implementierung dieser Ideen auf allen Ebenen (*Informierten NegaScout-Algorithmus*), benötigt sehr viel Speicherplatz. Es gibt aber die Möglichkeit sich auf die oberste Ebene, wo der Gewinn auch am größten ist, zu beschränken. Dies geschieht in dem *Partiell Informierten NegaScout-Algorithmus*. Das PNS-Verfahren schneidet „in empirischen Untersuchungen deutlich besser als das NegaScout-Verfahren ab“ [REI89].

3.2.2.5 Zustandssuchverfahren

Zustandsorientierte Suchverfahren speichern alle Knoteninhalte ab. Sie können aufgrund des Informationsvorsprunges noch mehr Knotenexpansionen ausschließen als das $\alpha\beta$ -Verfahren. Die Suchleistung wird auf eine quasiparallele Expansion zurückgeführt. Diese kommt aufgrund eines bestenorientierten Suchverfahrens zustande (das $\alpha\beta$ -Verfahren arbeitet wie eine Tiefensuche), die auf den gewonnenen Daten ausgeführt wird. Dabei wird stets der Knoten mit der höchsten Bewertung zuerst expandiert. Die wegen des enorm hohen Speicherplatzbedarfes und aufgrund der mühsamen Verwaltungsaufgaben sehr rechenzeitintensiven, in der Praxis kaum vorkommenden, Zustandssuchverfahren werden hier nicht weiter betrachtet.

3.2.3 Optimierung des Suchverfahrens

Die Effizienz der Suche, die im Wesentlichen von einem $\alpha\beta$ -Gerüst ausgeführt wird, hängt stark von der Anzahl der Schnitte ab. Je weiter oben im Baum ein Schnitt stattfindet, desto größer die Einsparung. Die Zugreihenfolge spielt hinsichtlich der Effizienz eine herausragende Rolle.

Eine Suche zu einer festen Tiefe gefolgt von einer Knotenbewertung ist nachteilig, wenn ein Materialvorteil zu Lasten eines Ausgleiches im nächsten noch nicht sichtbaren Schritt erfolgt. Ein gravierendes Beispiel ist ein Damentausch. Es wird eine variable Suchtiefe motiviert, welche nur *ruhige Stellungen*⁸ bewerten soll. Zu guter Letzt wird eine Möglichkeit angegeben, Bedrohungen abzuschätzen.

3.2.3.1 Iterativ in die Tiefe gehen:

Es kann vorteilhaft sein, den Baum nicht gleich in die volle Tiefe zu expandieren, sondern stattdessen von einer geringen Tiefe ausgehend, diese unter kompletter Neuberechnung sukzessiv zu erhöhen, bis die Zugzeitdauer abgelaufen, oder eine bestimmte Tiefe erreicht worden ist. Dies klingt ineffizient, dem ist aber nicht so:

Als vereinfachende Annahme werde jeder Knoten expandiert, das Spiel höre nicht innerhalb der Tiefenbeschränkung auf und es sei eine Verzweigungstiefe von 40 festgesetzt. Eine iterative Suche bis zur Tiefe i ist eine iterative Suche bis zur Tiefe $i - 1$, gefolgt von einer einfachen Suche bis zur Tiefe i . Im Folgenden werden die Kosten einer einfachen Suche und die zusätzlichen Kosten (verursacht von der iterativen Suche bis zur Tiefe $i - 1$) in Relation gestellt. Als Erstes werden die Kosten der Knotenexpansion verglichen. Bei der einfachen Suche werden $\sum_{j=0}^i 40^j = \lfloor 40^{i+1}/39 \rfloor$ Knoten, bei der iterativen Suche (bis zur Tiefe $i - 1$) höchstens $\sum_{j=0}^{i-1} \lfloor 40^{j+1}/39 \rfloor \leq \lfloor 40^{i+1}/39^2 \rfloor$ Knoten expandiert. Nun wird die Anzahl evaluierter Knoten betrachtet. Bei der einfachen Suche sind es 40^i Knoten, bei der iterativen Variante immerhin $\sum_{j=0}^i 40^j = \lfloor 40^{i+1}/39 \rfloor \geq 40^i$. Das iterative in die Tiefe gehen ist, unabhängig von der Tiefe, um etwa $1/40$ langsamer, wenn man die Knotenexpansionen als Maßstab nimmt. Werden hingegen die teuren Knotenbewertungen betrachtet, so kostet das iterative in die Tiefe gehen, doppelt so viel.

Selbstverständlich müssen aber nicht alle Knoten expandiert (und somit nicht bewertet) werden. Eine überdurchschnittlich hohe Anzahl von Schnitten in Iteration $j + 1$ kann dadurch erzielt werden, daß die Züge, die sich in Iteration j bewährt haben, als Erstes ausprobiert werden. Dies gilt bis zur letzten Iteration. Letztendlich werden beim iterativen Verfahren meist weniger Knoten expandiert.

Dazu kommt noch, dass eine genauere Abschätzung bezüglich der benötigten Rechenzeit möglich ist.

3.2.3.2 Die Zugreihenfolge

Wenn man die, aus Sicht des sich am Zug befindlichen Spielers, vielversprechenderen Züge zuerst expandiert, werden meist schnell gute Züge gefunden. Das Suchfenster kann dann für die nächsten entsprechend verkleinert werden, wodurch die Suche erheblich beschleunigt wird. Selbst wenn die vermeintlich guten Züge die schlechtesten sind, arbeitet das Verfahren zur Minimaxberechnung korrekt, aber deutlich langsamer. Insgesamt handelt es sich aber um eine

⁸ Das sind beispielsweise solche, die nicht über einen schlagenden Zug erreicht wurden.

vorteilhafte Heuristik. Die folgenden Ideen sind im Wesentlichen auch in [MS90] aufgeführt.

Eine von Slagle and Dixon *fixed ordering* genannte Methode sieht vor, alle Kinder eines Knotens einer statischen Bewertung zu unterziehen, und daraus die Expansionsreihenfolge zu ermitteln. Beim *dynamic ordering* ist ein Betrachten eines anderen Zuges aufgrund neu gewonnener Information stets möglich, es entspricht im wesentlichen einer Bestensuche und somit einem Zustandssuchverfahren.

Angenommen ein Knoten hat ein Kind, welches einen Schnitt verursacht, oder zu einer Verbesserung der unteren Schranke führt. Durch Abspeicherung dieser sogenannten *killer moves* in einer entsprechenden *killer table*, können diese im Folgenden in den oft ähnlichen Positionen der gleichen Ebene zuerst probiert werden. Als kleiner Nebeneffekt taucht die identische Brettstellung auf dieser Ebene meist mehrfach auf. Handelt es sich beispielsweise um eine Position, welche sich sechs Halbzüge unter aktuellen Wurzelposition befindet, so gibt es zu jeder Position im Schnitt grob 50 identische allein auf der gleichen Ebene, welche tatsächlich expandiert werden. Der abgespeicherte Zug wird also auf gleiche Stellungen mehrfach angewandt.

Beim oben diskutierten iterativen-in-die-Tiefe-gehen, kann zu jedem Knoten der beste Nachfolger angegeben werden. Um Speicherplatz zu sparen, ist eine Beschränkung auf die Abspeicherung in den oberen Ebenen möglich. Geschickte Zugreihenfolgen in unteren Ebenen resultieren ohnehin bloß in mäßigem Erfolg.

3.2.3.3 „Quiescence“ und der Horizont-Effekt

Die Bewertung einer Position wird im Wesentlichen durch das vorhandene Figurenmaterial bestimmt. Befindet sich das Spiel z. B. mitten in einem Bauerntausch, so kann die Bewertung einen zu guten oder zu schlechten Wert liefern. Probleme oder Chancen hinter dem Suchhorizont bleiben unberücksichtigt. Als Konsequenz ist eine ungünstige Zugwahl denkbar. Um diesen *Horizont-Effekt* zu vermeiden, versucht man eine Evaluierung nur in *stabileren (quiescent)* Stellungen vorzunehmen. Turing bezeichnet solche Positionen als *dead*. In anderen Fällen, wird die Suche noch etwas fortgesetzt. Auf diese Weise wird die Suchtiefe variabel.

3.2.3.4 Variable Suchtiefe

Wie in [MS90] diskutiert, besteht der wesentliche Zweck einer weiteren selektiven Suche variabler Tiefe in der Effizienzsteigerung. Bestimmte interessante Zugfolgen sollen weiter in die Tiefe hinein verfolgt werden. Im Zusammenhang mit der Aufhebung des Horizont-Effektes wird auch von *quiescence search* gesprochen. Näher zu betrachtende Brettstellungen sind solche, wo Figuren eben geschlagen worden sind oder Schach geboten wird.

Es gibt einige Ansätze, wie selektiv weiter gesucht werden soll. Der Vielversprechenste lässt die Suchtiefe der Hauptsuche einfach variabel werden. Dies kann zum Beispiel sehr effektiv dadurch erreicht werden, daß ein Antwortzug auf ein Schach nicht zur Tiefe dazugerechnet wird. Die Laufzeit wird davon nicht

allzu stark betroffen, denn es gibt meist nur wenige Antworten auf ein Schach.

3.2.3.5 Die Null-move Heuristik

Ein Null-move gibt das Zugrecht an den Mitspieler ab, ist also kein legaler Zug. In den meisten Fällen wäre das schlechter, als von seinem Zugrecht Gebrauch zu machen. Die restlichen Fälle werden Zugzwang genannt. Kann der Mitspieler aus seinen zwei Zügen keinen besonderen Vorteil erringen, wird die Position des Spielers als nicht bedroht angesehen. Bei der Berechnung des Null-moves wird meist in der Tiefe gespart. Der Wert eines Null-moves kann in der Regel als untere Schranke für den Wert der aktuellen Position dienen. Handelt es sich um eine Zugzwangposition, wird eventuell ein falscher Minimaxwert an der Wurzel berechnet. Gleiche Folgen kann eine reduzierte Tiefe bei der Berechnung des Null-move-Wertes haben. Bei zu knapper Wahl tritt oft der Horizont-Effekt auf. Deswegen wird der Null-move nicht in Endspielsituationen zum Einsatz kommen. Analog zur Berechnung im Abschnitt 4.2 kostet die Berechnung der Null-move-Bewertung (einer entsprechenden Tiefe) nur etwa 1/40 der eines legalen Zuges. Der Null-move kann z. B. als allererster Zug berechnet werden. Der zurückgelieferte Wert wird wie der eines normalen Kindes betrachtet. Ist er größer als der zu dem Knoten gehörige β -Wert kann ein Cutoff erfolgen. Ist die Suchtiefe des Null-moves dazu noch um z. B. zwei kleiner, so ergibt sich ein Rechenzeitgewinn um den Faktor 60 000. Verursacht er allerdings keinen Cutoff, war die Bemühung umsonst. Es gibt Situationen, in denen der Null-move ausgeschlossen werden sollte. Der Null-move wird nicht zum Einsatz kommen, wenn der Spieler im Schach steht, wenn der vorherige Zug ein Null-move ist, oder es sich um die Wurzel des Baumes handelt. Schließlich soll der Null-move Baumteile ausschließen und nicht den besten Zug ausweisen. Für eine ausführliche Diskussion sei auf [MS90] verwiesen.

3.2.3.6 Spekulative Zugauswahl

Das Minimaxverfahren setzt zwei unfehlbare Spieler voraus, arbeitet dann aber optimal. Jeder muß einen optimalen Zug ziehen, da er so die Bewertung immer sichert. Falls ein Spieler aber nicht alle Züge überblickt, kann eine *gute* Strategie solche Züge benutzen, die mit hoher Erwartung eine große Bewertung sichern, aber mit kleiner Wahrscheinlichkeit sich als schlechter herausstellen (abhängig vom Verhalten des Gegenübers). Es gibt vereinzelt Situationen, in denen selbst ehemalige Schachweltmeister ein Matt in wenigen Zügen übersehen. In [MS90] ist ein Beispiel genannt, indem das Aufsuchen einer solchen Stellung erfolgversprechend ist. Spekulatives Spiel lohnt sich meistens in problematischen Situationen, in denen der beste Zug nicht offensichtlich ist oder der Offensichtliche nicht der Beste. Es setzt einen menschlichen Mitspieler voraus.⁹ Es kann nun neben dem Minimaxwert v_m ein zweiter spekulativer Wert v_s berechnet werden. Aus diesen beiden Werten berechnet sich nun effektive Wert $v_e = \xi v_s + (1 - \xi)v_m$, wobei

⁹ Ein gegnerisches Computerprogramm müsste wenigsten eine deutlich kleinere Suchtiefe benutzen.

$\xi \in [0, 1]$ die je nach Spielsituation erforderliche Gewichtung ist. Mit diesem Wert wird ein Knoten dann endgültig bewertet. Der interessierte Leser findet eine Einführung in [MS90].

3.2.4 Material- und Positionsbewertung

Während das vorherige Kapitel ganz der Suche gewidmet war, geht es hier vielmehr um das Einbinden von Knowhow. Gegenstand aller Betrachtungen ist Schach. Selbst mit dem ausgefeilten NegaScout-Verfahren ist es undenkbar, den Baum in seiner vollen Tiefe zu berücksichtigen. Die Folge der Tiefenbeschränkung ist, dass Brettpositionen erreicht werden, bei denen das Spiel nicht zu Ende ist. Für diese muss eine Stellungsbewertung erfolgen, die ein Skalar zurückliefert. Das Skalar wird von den Suchalgorithmen wie eine Blattbewertung angesehen. Eine Brettstellung mit einer schlichten Zahl zu bewerten, stellt ein schwieriges Unterfangen dar. Meist werden Erfahrungen, Intuition und einfaches Raten benutzt.

Alle Standardschachprogramme benutzen eine Materialbewertung, sie addieren z. B. für jeden Spieler am Zug seine Figuren entsprechend einer festen Gewichtung und subtrahieren das Material des Mitspielers. Dies hat den positiven Nebeneffekt, dass oft sogar Bedrohungen erkannt werden. Wird nämlich zwischen Ebene t und $t + 1$ eine Dame geschlagen, taucht sie in der Materialbewertung von Ebene $t + 1$ nicht mehr auf. Ein weiterer Vorteil der Materialbewertung ist die effiziente Durchführbarkeit. Eine mögliche Bewertung der Figuren ist die folgende: Bauer 1 000, Springer und Läufer 3 000 (manchmal auch 3 250 und 3 500), Turm 5 000, Dame 9 000. Der Läufer wird oft etwas stärker als der Springer bewertet. Die genannten Zahlen treffen den Wert der einzelnen Figuren zumindest in der Startaufstellung recht gut. Je nach Spielsituation verändern sich allerdings die Werte dynamisch. In der Regel kann diesem Aspekt kein Schachprogramm auf effiziente Weise Rechnung tragen. Sicherlich gibt es darüberhinaus auch Bewertungen von positionellen Merkmalen. Positionelle Merkmale haben einen geringeren Einfluss auf die Größe der Gesamtbewertung, als die Materialbewertung. Nur selten übersteigt die Summe dieser „Merkmale den Wert eines Bauern“ [SF95]. Dies erfolgt zum Beispiel, wenn feststeht, dass ein Bauer in wenigen Zügen zur Dame werden kann. Der Grund für diesen geringen Einfluss positioneller Merkmale ist, dass in der Regel kein Bauer geopfert werden soll, um eine taktisch bessere Position zu erhalten. Trotz der ernüchternden Feststellung sollten positionelle Bewertungen sehr genau betrachtet werden. Sie sind in einem gesunden (effizient durchführbaren) Maß erforderlich.

3.2.4.1 Statische Bewertungsfunktionen und Komplexität

Wie in [MS90] beschrieben, erfolgen meist mehrere Bewertungen diverser Eigenschaften einer Stellung, die dann durch entsprechende Konstanten gewichtet zu einer Gesamtbewertung kumuliert werden, man spricht von einer gewichteten Summe. Problematisch sind Eigenschaften, die im Verlauf des Spieles an Wert gewinnen oder verlieren. Der Versuch, mehrere Bewertungsfunktionen über die Zeit zu benutzen, führt meist zu unerwünschten Unstetigkeiten beim Wechsel.

Ein Fehlverhalten (blemish effect) ist in solchen Fällen programmiert. Denkbar ist der Einsatz eines Fuzzysystems, um weichere Übergänge zu ermöglichen.

3.2.4.2 Positionelle und sonstige Bewertungen

Es gibt Richtlinien, an denen man sich bei einer Bewertung orientieren kann. Dennoch ergibt die Bewertung einer Brettstellung bei verschiedenen Schachprogrammen, oft ein unterschiedliches Skalar. Jedes Schachprogramm setzt seine Akzente, jedoch folgen die Bewertungen eines Brettes gewissen Bewertungen, die an den einzelnen Figurtypen festgemacht werden können (Materialbewertungen).

König:

Für den König bietet es sich an, verschiedene Spielphasen zu unterscheiden. Z. B. kann, wenn der Mitspieler nicht mehr als 13 000 Materialpunkte bekommt, dadurch das Endspiel definiert sein, darüberliegende Punktzahlen bilden eine mittlere Phase. Allgemein wird oft das Endspiel so definiert, dass der Gegner von der Materialsomme maximal einen Turm und einen Läufer besitzt, oder beide zusammen das Doppelte. Eine Summierung der eigenen Werte ist also auch möglich. Man beachte, dass so aus Sicht ein und desselben Spielers nur ein Phasensprung stattfinden kann. In der mittleren Phase genießt die Sicherheit des Königs Priorität. Dies fängt bei der Bewertung der Position des Königs bereits an. Je weiter hinten er sich befindet, desto besser (meist linear). Für das Feld des Springers gibt es einen großen Bonus, für das Turm- oder Läuferfeld einen kleineren. Es ist sinnvoll, diese Summe leicht von anderen überlagern zu lassen, um einen König im Brettzentrum zu honorieren, wenn das Endspiel begonnen hat.

Zur Sicherheit gehört im Wesentlichen auch die Struktur der vor dem König befindlichen Bauern. Fehlen welche, oder lassen sie einen direkten Angriff über offene Linien zu, muss ein Abzug erfolgen. Es können weitere Unterschiede hinsichtlich der Art der Lücken erfolgen, z. B. besteht eine konkrete Bedrohung durch eine Figur, oder ist wenigstens ein feindlicher Bauer im Weg. Es kann auch einfach unterschieden werden, ob ein Bauer direkt vor dem König, eine Reihe weiter steht oder gar nicht vorhanden ist.

Die Bewegungsfreiheit des Königs wird ebenfalls oft bewertet. Dabei ist eine mögliche Rochade besonders zu belohnen.

Dame:

Für die Damen sind nicht so viele Gesetzmäßigkeiten gefunden worden, wie etwa für andere Figuren. Immerhin kann auch die Damenbewertung von unterschiedlichen Phasen bestimmt werden. So nachteilig es ist, anfangs mit der Dame zahlreiche Aktivitäten zu veranstalten, so unvermeidlich wird dies in späterer Spielphase sein. Dort sollte eine zentrale Stellung möglichst nahe am gegnerischen König eingenommen werden. Falls der Mitspieler seinen König nicht genügend absichert, kann die Dame an Wert gewinnen.

Turm:

Auf die Bewertung des Turmes haben folgende Eigenschaften Einfluss:

1. Königsnähe
2. Mobilität (insbesondere horizontale)
3. Relative Position zu eigenen Figuren (etwa: Turm steht hinter einem Freibauern)

Läufer:

Sind zwei eigene Läufer vorhanden, so besetzen diese fast ¹⁰ immer Felder unterschiedlicher Farbe, dadurch können prinzipiell alle Felder abgedeckt werden. Dies wird manchmal belohnt, meist wird aber das Vorhandensein nur eines Läufers bestraft. In vielen Fällen ist *ein* Springer dann sogar mehr Wert als *ein* Läufer.

Eine weiteres Maß ist die generelle Mobilität eines einzelnen Läufers, dafür wird die Anzahl der feindlichen Figuren auf Feldern gleicher Farbe abzüglich der eigenen bewertet. Darüber hinaus werden oft Diagonalen darauf überprüft, ob sie versperrt sind.

Springer:

Der Springer soll nicht zu sehr am Rande des Brettes positioniert sein, da so seine Mobilität stark eingeschränkt werden kann. Er sollte sich in der Nähe des gegnerischen Königs aufhalten. Besonders empfehlenswert sind Springer, welche auf der gegnerischen Seite eine Position decken, welche nicht wiederum von einem gegnerischen Bauern gesichert wird. Auch soll der Springer nicht den c7-Bauern, mit dem Zug Sc6 blockieren [SF95].

Bauer:

Um die Bewertung des Bauern machen sich Spieleprogrammierer meist die größten Gedanken. Im Schachprogramm Cray Blitz wurde, wie in [MS90] geschildert, folgende Bewertung vorgesehen:

1. Die zentralen Bauern (Königsbauer und Damenbauer) werden bestraft, wenn sie ihr ursprüngliches Feld nicht verlassen. Insbesondere falls dies auf beide zutrifft, soll eine stärkere Bestrafung erfolgen.
2. Als nächstes wird das Voranschreiten, welches in Abhängigkeit von der Eröffnung taktisch günstig ist belohnt.
3. Bauern unterschiedlicher Parteien, die sich gegenseitig blockieren, werden bestraft, damit mehr freie Reihen bleiben.
4. Die Präsenz von acht Bauern wird bestraft, um wenigstens einen Bauerntausch zu fördern.

¹⁰ Es kann bei einer Verwandlung eines Bauern indirekt Schach geboten werden. Falls sich mit einer Dame eine Pattsituation ergeben würde, ist eine Verwandlung zu einem Läufer denkbar.

5. Isolierte Bauern, zu denen es also in benachbarten Reihen keine Bauern gibt, werden differenziert bestraft. Solche, die keinen gegnerischen Bauern vor sich haben, trifft es am härtesten. Ebenso werden zu weit hervorgezogene Bauern, die quasi isoliert sind, bestraft.
6. Bauern, die keinen Schutz ihrer benachbarten Bauern genießen, werden wie isolierte Bauern bestraft.
7. Freibauern bekommen nur einen Bonus, wenn der Gegner weniger als 13000 Materialpunkte bekommt, wenn also das Endspiel eingetreten ist. Ein Freibauer kann in Endspielen spielentscheidend sein. Eine Bewertung, die sie auf beiden Seiten berücksichtigt und einer Seite einen Bonus erteilt, ist komplex. Insbesondere wird versucht, eine Abschätzung ohne weitere Baumsuche zu tätigen. Nur wenn kein Zweifel an dem Erhalten einer Dame besteht wird ein Bonus gegeben, welcher aber etwas unter der Bewertung einer Dame liegt, immerhin aber mehr als der eines Turmes angesetzt wird (z. B. 750). Ist die Situation unklar, muss eine tiefere Suche erfolgen.
8. Endspiele in denen neben den Königen nur noch Bauern auftauchen, können heutige Programme recht leicht lösen. Dazu braucht *Cray Blitz* keine weitere Baumsuche durchzuführen, die Position kann direkt als gewonnen oder verloren bewertet werden.

Sonstige Bewertungen:

Es sind spezielle Eröffnungsregeln angebracht. Dazu zählen solche, wie „Bestrafen den zweiten Zug einer Figur, bevor nicht jede Figur einmal gezogen wurde“. Ausnahmen gibt es aber auch hier. So müssen nicht nach einer Rochade noch die schützenden Bauern gezogen werden. Dazu wird allgemeines Eröffnungs- und Endspielwissen (z. B. mit Turm) implementiert. Die Eröffnung kann etwa mit dem Wegziehen von vier leichten Figuren aus der Ausgangstellung oder von dreien zuzüglich einer Rochade, erfolgen.

Remisbehandlung:

Remispositionen werden manchmal mit 0 bewertet, hin und wieder aber wie üblich bewertet. Damit soll einem menschlichen Gegner, der versucht, das Remis solange wie möglich hinaus zu zögern, die Möglichkeit offengehalten werden, durch einen Fehler zu verlieren. Das Programm kann auch versuchen, das Remis gänzlich zu vermeiden. Erfolgt eine Remisbewertung mit 0, so können auch Endspiele die Remis ausgehen müssen (wie z. B. König und zwei Springer gegen König) mit Remis bewertet werden.

3.3 Maschinelles Lernen und Evolutionäre Algorithmen

Computer sind auf strengen logischen Prinzipien beruhende Maschinen. Programmierung bedeutet in der Regel die Erstellung eines festgelegten und deterministisch ablaufenden Arbeitsplanes für eine solche Maschine. Die damit verbundene geringe Flexibilität bei der Bearbeitung der ihnen gestellten Aufgaben

ließ schon früh den Gedanken aufkommen, Computer *intelligenter* zu machen. Gemeint waren damit dem menschlichen Verhalten abgeschauten Vorgehensweisen und Eigenschaften. Andersherum betrachtet wäre es interessant zu wissen, wie die *biologische Maschine* „Mensch“ funktioniert bzw. ob dieses Bild ein dem Menschen adäquates Modell ist.

Diese Fragestellungen und Ideen sind kennzeichnende Bestandteile der Forschung im Bereich *Künstliche Intelligenz* (KI). Hier spielen sowohl die technische Anwendbarkeit in flexibler und intelligenter Software, als auch neue Erkenntnisse über die menschliche Informationsverarbeitung im Gehirn eine Rolle. Letzteres wurde zum zentralen Untersuchungsgegenstand des ebenfalls noch relativ jungen Forschungszweiges der *Kognitionswissenschaften* (KW). Zum Kernbereich der Forschung in der Künstlichen Intelligenz zählen *Wissensrepräsentation*, *effizientes Suchen* und *Inferenzmechanismen*, d. h. die Bestimmung von elementaren Problemlöseoperatoren, sowie die Transformation von Repräsentationsebenen (Für einen Überblick siehe [GÖR95]).

Allerdings beschränken sich in beiden Disziplinen die Untersuchungen im Wesentlichen auf die *kognitiven* Fähigkeiten des Menschen wie *Wissen*, *Lernen*, *Visuelle Wahrnehmung* und *Sprache*. Dagegen setzt die erst in den letzten Jahren sich etablierende *Computational Intelligence*-Forschung (CI) breiter an. Der schwer ins Deutsch zu übersetzende Begriff deutet an, dass nicht mehr allein die Erforschung der Funktion der menschlichen Kognition im Mittelpunkt steht, sondern die Frage, wie die maschinellen Vorgänge des Computers, unter Ausnutzung natürlicher Vorbilder, intelligenter ablaufen können. Diese Vorbilder beziehen auch andere Optimierungsverfahren der Natur ein. Die Evolution der Arten beispielsweise hat viele Probleme gelöst, ohne dabei eine vorher eingebaute Strategie zu verwenden. Das menschliche Gehirn und die Fähigkeit des Menschen, mit unscharfen Aussagen klare Anweisungen oder Beschreibungen geben und verstehen zu können, werden in weiteren Teilbereichen der CI erforscht. Die drei Schwerpunkte der Computational Intelligence sind die Gebiete *Evolutionäre Algorithmen*, *Künstliche Neuronale Netze* und *Fuzzy Control*.

Evolutionäre Algorithmen entstanden ebenfalls aus den gleichen Grundmotivationen, wie es auch für die KI beschrieben wurde: Technische Ausnutzung biologischer Vorbilder und Erkenntnisgewinnung über biologische Informationsverarbeitung. Unabhängig voneinander entstanden Verfahren zur Parameteroptimierung mit Hilfe von *Evolutionsstrategien* und zum Verständnis der Informationsverarbeitung auf diskreten Chromosomen durch *Genetische Algorithmen*. Letztere erwiesen sich ebenso als nützlich zur Bearbeitung technischer Fragestellungen.

Ein wesentlicher Antrieb der Forschung mit Evolutionären Algorithmen war die Suche nach Optimierungsverfahren, in der Künstlichen Intelligenz liegt ein Schwerpunkt in der Ausnutzung von Lernverfahren. Lernen aber ist auch als *Suche* darstellbar und Optimierung wiederum ist ebenfalls eine Form von Suche. Computer als flexible Problemlöser stehen im Spannungsfeld von Lernen, Suchen und Optimieren. Diese Begriffe werden nicht immer getrennt voneinander behandelt und gehen in ihren Bedeutungen ineinander über. Daher kann auch im Folgenden nicht immer sauber zwischen den einzelnen Bereichen unterschieden

werden. Suchverfahren wie bspw. das *Hill Climbing*, die im Maschinellen Lernen Verwendung finden, spielen eine ebenso wichtige Rolle in der technischen Optimierung. Dieses Kapitel soll einen Überblick über Möglichkeiten und Ansätze im Maschinellen Lernen vermitteln und die grundsätzlichen Vorgehensweisen bei Evolutionären Algorithmen darstellen. Im nächsten Kapitel wird dann das *Genetische Programmieren*, die Basistechnologie in dieser Projektgruppe, und Evolutionäre Algorithmen ausführlicher behandelt.

3.3.1 Maschinelles Lernen

Das maschinelle Lernen ist ein wichtiges Teilgebiet der Künstlichen Intelligenz und befasst sich mit der Modellierung und Entwicklung von Verfahren, mit denen ein System sich selbst verändern und damit verbessern kann (*Selbstmodifikation*). Das System stellt aufgrund von Beispielen und Beobachtungen Hypothesen auf und überprüft diese. Dadurch kann das System automatisch Regeln erwerben oder bereits vorhandene verbessern. Zur Generierung und Überprüfung der Hypothesen gibt es sowohl numerische als auch logik-orientierte Lernverfahren. Das maschinelle Lernen wird zur Analyse von Daten, zur Entdeckung von Regelmäßigkeiten in Datenbanken, zur Optimierung von Systemverhalten und zur Unterstützung des Wissenserwerbs genutzt.

Der Begriff des *Maschinellen Lernens* wurde erstmals von Samuel 1959 [SAM67] geprägt und interessanterweise bereits im Sinne von automatischer bzw. selbständiger Programmierung benutzt. Allerdings erwies sich dieser Wunsch als bei weitem zu kompliziert, sodass Maschinelles Lernen in den kommenden Jahren eher als das benutzt wurde, was Tom Mitchell 1997 in seinem Buch *Machine Learning* [MIT97] allgemeiner ausdrückte: „[Machine Learning] is the study of computer algorithms that improve automatically through experience“.

Friedberg beschäftigte sich ebenfalls 1958/59 mit dem Problem der selbständigen Programmierung von Computern. In [FDN59, FRI58] beschreibt er zwei grundsätzlich verschiedene Vorgehensweisen, einem Computer zu vermitteln, wie er zum Beispiel menschliche Sprache verstehen oder übersetzen kann, an neue mathematische Problemstellungen herangehen soll oder eine Firma zu leiten hat. Die erste Idee erläuterte er mit den Worten „tell a machine precisely how to go about doing them“ und beschrieb damit das Vorgehen wissensbasierter Systeme, also der Befragung von Experten und der statischen Implementierung der Ergebnisse im Programmcode. Dieser Idee gingen die KI-Forscher in den 60er und 70er Jahren hauptsächlich nach, wobei sich herausstellte, dass dieses Vorgehen wenig mit Lernen im Sinn einer Adaptation an eine Situation zu tun hat. Ein wissensbasiertes System kann sich nicht eigenständig verändern und nur mit großem Aufwand auf ähnliche oder gar neue Probleme angepasst werden. Daraufhin wurde im Laufe der 70er Jahre die Idee des Maschinellen Lernens (Friedberg: „[tell a computer] precisely how to learn“) wieder aufgegriffen. Nach den ersten Erfolgen in den frühen 80er Jahren verrichten heutige Systeme, die auf dem Paradigma des Maschinellen Lernens beruhen, in der Industrie als Grundlage von Robotersteuerungen bestimmte Aufgaben wesentlich exakter als sie ein Mensch hätte erledigen können. Sie werden für Vorhersagen im Wetterdienst genauso wie

an der Börse verwendet (siehe auch Projektgruppe 350, Vorhersagemethoden für den Finanzmarkt) und helfen bei der Muster- und Sprachkennung.

Algorithmen des Maschinellen Lernens sollen dazu in der Lage sein, gegebene Situationen zu „beurteilen“, zuzuordnen oder zu kategorisieren. Die Leistungsfähigkeit wird durch die Art des gewählten Lernvorganges in der jeweiligen Situation bestimmt. Dies kann durch Kategorisierungsverfahren, Anpassung interner Strukturen in einer Trainingsphase oder Inferenzmechanismen basierend auf logischen Kalkülen geschehen. Neben den gemeinsamen *charakteristischen Merkmalen* gibt es *Unterschiede in der Darstellung*, verschiedene *Suchstrategien* und *Lernvorgänge*, die im Folgenden kurz dargestellt werden.

3.3.1.1 Charakteristische Merkmale

Maschinelles Lernen ist die Suche nach einer möglichst guten Lösung zu einem Problem in einem Suchraum. Für den Prozess des Maschinellen Lernens ist es wichtig zu verstehen, wie ein System entwickelt wird, wie es lernt und wie das Gelernte überprüft werden kann.

Der Suchraum wird durch einen Lernbereich (*Learning Domain*) charakterisiert. Das ist die Menge aller Merkmale (*Features, Inputs*), die zu identifizieren und den entsprechenden Ergebnissen (*Classes, Outputs*) zuzuordnen sind. Natürlich sollten diese in einer Beziehung zueinander stehen, wie es zum Beispiel bei meteorologischen Daten (Temperatur, Luftdruck und -feuchtigkeit) aufeinanderfolgender Tage der Fall ist.

Gelernt bzw. trainiert wird anhand von Trainingsmengen (*Training Sets*), also Mengen von Beispielen aus der Learning Domain (Paare von Ein- und Ausgaben). Die vollbrachte Lernleistung eines Systems wird danach mit Hilfe einer Testmenge (*Test Set*) bestimmt, wiederum eine Menge von Beispielen aus der Learning Domain, die allerdings nicht zuvor schon im Training Set vorkamen. Die Leistung eines Systems wird durch die Fähigkeit bestimmt, Gelerntes auf neue, unbekannte Beispiele anzuwenden, das bisherige Wissen so zu generalisieren, sodass nicht nur bekannten Eingaben korrekte Ausgaben zugeordnet werden können.

3.3.1.2 Repräsentationen

Um ein Problem von einem Lernalgorithmus lösen zu lassen, muss zu Beginn festgelegt werden, wie das Problem repräsentiert wird, in welcher Form der Vorgang des Lernens mit dieser Darstellung arbeitet und wie das produzierte Ergebnis interpretiert wird.

Ein einfaches Beispiel zur Problemrepräsentation ist die Nullstellen-Suche bei Funktionen: Das Problem kann bei polynomieller Darstellung der Funktion durch einen Vektor beschrieben werden, der die Koeffizienten beinhaltet. Eine Lösung wird im eindimensionalen Fall durch eine einzige (reelle) Zahl repräsentiert.

Desweiteren muss festgelegt werden, in welcher Form gelernt werden soll bzw. in welcher Form etwas Gelerntes dargestellt wird (*concept description language*). Hier gibt es eine große Anzahl von Konzepten, die sich nicht nur im Vorgehen, sondern auch in ihrer Komplexität teilweise erheblich voneinander

Waffe	Wert	Maßnahme (wenn zutreffend)
Pistole	JA/NEIN	Schnell reagieren
Handgranate	JA/NEIN	Schnell reagieren
Hand in der Tasche	JA/NEIN	Langsam reagieren

Tabelle 1: Banküberfall

unterscheiden. So gibt es Systeme, die mit einfachen boolschen Verknüpfungen auskommen. Andere verwenden Listen mit disjunkten Merkmalen, orientieren sich an Klasseneinteilungen, bilden Durchschnittsanalysen, stellen sich in Form von Neuronalen Netzen dar oder – wie im Fall der Genetischen Programmierung – lernen Computerprogramme.

Als letzter Punkt ist noch die Frage der Ergebnisinterpretation zu klären. Zum Beispiel ist ein Neuronales Netz ein Konzept, das bei der Interpretation erhebliche Schwierigkeiten bereitet, da die enthaltenen Werte Gewichte zwischen den Netzelementen festlegen und keine direkte symbolische Bedeutung haben. Ein solches Netz kann normalerweise nur danach beurteilt werden, wie gut es im Training und Test arbeitet, ob es also die erwarteten Ausgaben produziert. Fuzzy-Systeme hingegen verfolgen gerade das Ziel „unscharf“ formulierte und sich überschneidende Regeln zu verarbeiten. Ebenso sind kombinationen aus den Ansätzen denkbar.

Für die folgende Übersicht der Repräsentationsmöglichkeiten wird als Beispiel ein System beschrieben, dass einem Bankangestellten helfen soll, sich bei einem Überfall korrekt zu verhalten. Die Eingangsgröße ist die Art der Bewaffnung des Bankräubers mit den Werten *Pistole*, *Handgranate* und *Hand in der Tasche*. Klare Aussagen kann das System für alle drei Einzelwerte geben, allerdings ist die Frage, wie es reagiert, wenn der Bankräuber mit *Pistole und Handgranate* sein Anliegen vorträgt.

Boolean-Repräsentation:

In einem System, dass vollständig auf der Boolean-Repräsentation beruht, werden Merkmale mit JA (das Merkmal ist gegeben bzw. trifft zu) oder NEIN (das Merkmal ist nicht gegeben) verwaltet, und gelernte Konzepte werden als Konjunktionen oder Disjunktionen der Merkmale abgebildet. Stellt sich also zum Beispiel die Frage, ob der Bankangestellte das Geld schnell oder langsam herausrücken soll, so hängt es davon ab, mit welcher Waffe er bedroht wird.

Bei einer sichtbaren Waffe sollte der Bankangestellte somit schnell sein, ist die Waffe in der Jackentasche versteckt, darf er sich ruhig Zeit lassen. Natürlich kann es bei dieser Auslegung bzw. bei diesen gelernten Konzepten passieren, dass der Bankangestellte erschossen wird, was (hoffentlich) dazu führen würde, dass das vorhandene Konzept geändert wird ...

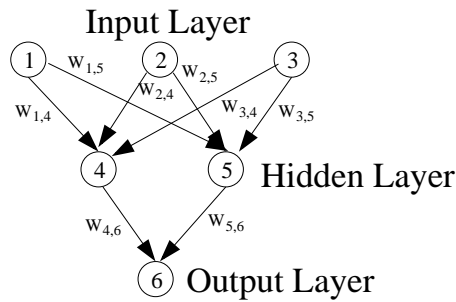


Abbildung 4: Neuronales Netz

Threshold-Repräsentation:

Richtet sich ein System nach der Threshold-Repräsentation, so wird ein Ausgangswert in Abhängigkeit eines gegebenen *Schwellwertes* (Threshold) bestimmt. Eine boolsche Ausgangsvariable hat zum Beispiel nur dann den Wert JA, wenn der entscheidende Wert größer als 0,5 ist. Im Banküberfall-Beispiel könnte die Entscheidung für ein schnelles oder langsames Herausgeben des Geldes zum Beispiel zusätzlich davon abhängen, wie gefährlich der Bankräuber aussieht, oder aber auch mit wievielen Waffen er gleichzeitig droht.

Neuronale Netze sind ein Beispiel für die Verwendung einer Threshold-Repräsentation, wobei die Neuronen eines solchen Netzes Threshold-Einheiten bilden (siehe Abbildung 4). Künstliche Neuronale Netzwerke sind Lernalgorithmen, deren Aufbau und Funktionsweise dem menschlichen Gehirn nachempfunden sind.

Das menschliche Gehirn besteht aus schätzungsweise 10 bis 100 Milliarden Neuronen, die durch Axons, Dendriten und Synapsen miteinander vernetzt sind. Dabei besitzt jedes Neuron ungefähr 1000 bis 10000 direkte Verbindungen zu anderen Neuronen, auf denen je circa 1000 Impulse pro Sekunde übertragen werden können. Diese Operationen können gleichzeitig an verschiedenen Stellen des Neuronennetzwerkes ablaufen. Jedes Neuron besitzt zu jedem Zeitpunkt ein bestimmtes Aktivierungspotential, welches durch die von anderen Neuronen eingehenden Impulse verstärkt oder abgeschwächt/gehemmt werden kann. Wenn das Aktivierungspotential erreicht wird, sendet das Neuron selbst entsprechende Impulse an seine Nachbarn. Die Stärke der Verbindungen zwischen den Neuronen kann sich je nach Beanspruchung im Laufe der Zeit ändern. Das im menschlichen Gehirn gespeicherte Wissen wird letztendlich durch die Gesamtheit der Aktivierungspotentiale aller Neuronen und der zwischen ihnen bestehenden Verbindungsstärken dargestellt.

Neuronale Netzwerke modellieren diese Struktur durch bewertete Graphen. Dabei entsprechen die Knoten den Neuronen und die Kanten den Verbindungen, wobei die Kantenbewertung der Verbindungsstärke entspricht. Die Knoten befinden sich zu jedem Zeitpunkt in einem Zustand, der einem bestimmten Wertebereich entstammt. Dieser Wert entspricht der Summe der anliegenden Kantenengewichte. Beim Überschreiten eines vorher definierten Schwellwertes wird ein Wert an die benachbarten Knoten angelegt. Wenn die beispielsweise die Additi-

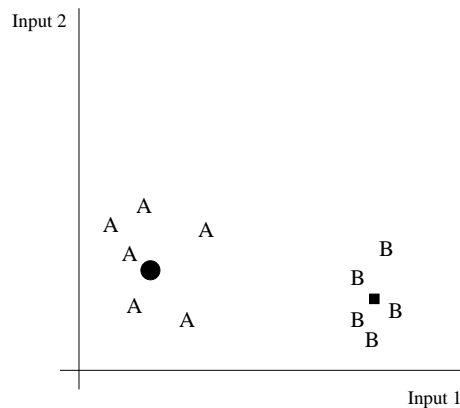


Abbildung 5: Klassifikation mit fallbasierter Repräsentation

on aller Eingabewerte eines Neurons oberhalb eines vorgegebenen Schwellwerts liegt, so wird dieser Wert an ein Neuron der nächsten Ebene weitergeleitet, ansonsten wird nicht „geschaltet“.

Neuronale Netzwerke werden nicht „fest programmiert“, sondern werden mittels Testdaten trainiert. Dieses Training kann entweder überwacht werden oder aber ganz ohne äußere Beeinflussung ablaufen. Die Topologie der neuronalen Netzwerke bleibt in der Regel während des Lernens unverändert, während sich ihre Verhaltensmuster durch die Verbindungsstärken (Kantenbewertungen) ergeben.

Fallbasierte Repräsentation:

Bei dieser Darstellungsart wird das Gelernte in Form einer Klasseneinteilung verwaltet. Ob ein Eingangswert zu einer bestimmten Klasse gehört, kann auf verschiedene Arten bestimmt werden, von denen hier das *Averaging*, also die Klassifizierung anhand einer Durchschnittsbildung, kurz erläutert wird. Abbildung 3.3.1.2 veranschaulicht das Vorgehen.

Eine neue Eingabe wird daraufhin getestet, ob sie näher am Schwerpunkt der Klasse A (Punkt) oder näher am Schwerpunkt der Klasse B (Rechteck) liegt und dann der jeweilige Klasse zugeordnet, worauf sich der Schwellwert verschiebt.

Repräsentation mit Bäumen:

Die Konzepte werden bei dieser Darstellungsart in Form eines Entscheidungsbaumes dargestellt. Jeder innere Knoten stellt dabei ein Merkmal dar, jede Kante einen möglichen Wert des Ausgangsknotens und jedes Blatt repräsentiert eine vollständige Klassifikation. Der Algorithmus *ID3* von Quinlan basiert darauf und ist (neben vielen Variationen) ein oft benutzter Algorithmus¹¹.

¹¹ Eine Einführung zur Repräsentation mit Bäumen findet sich in [qui90]

3.3.1.3 Suchstrategien

Maschinelles Lernen ist oft als Suche darstellbar, nämlich der Suche eines geeigneten Outputs zum gegebenen Input. Daher spielen Suchalgorithmen oder Optimierungsstrategien, die den Suchraum geeignet durchlaufen, eine entscheidende Rolle. Dabei gibt es verschiedene Vorgehensweisen beim Suchen, die teilweise auch vermischt sein können, und die im folgenden kurz vorgestellt werden.

Blind Search:

Die blinde Suche verwendet keinerlei Information über mögliche Strukturen des Suchraumes. Der Suchraum wird ohne Rücksichtnahme auf „Abkürzungen“, Wiederholungen oder dergleichen durchlaufen. Bekannte Algorithmen sind zum Beispiel die Tiefen- und die Breitensuche, welche Bäume in einer festgelegten Reihenfolge absuchen.

Solche auch als *vollständig* (exhaustive) bezeichneten Algorithmen sind nicht effizient, da sie alle Knoten auf dem Weg zur gewünschten Lösung überprüfen.

Hill Climbing:

Die Suche mittels *Hill Climbing* wird mit einer beliebigen Lösung des Suchraumes initialisiert. Danach wird eine neue Lösung in einer vorgegebenen Umgebung zur bisherigen Lösung gewählt und mit ihr verglichen. Im Falle einer Verbesserung wird die neue Lösung beibehalten, ansonsten verworfen. Dies wird so oft wiederholt, bis ein Abbruchkriterium erfüllt ist. Dabei wird immer nur eine mögliche Lösung gleichzeitig betrachtet. Bei einigen Algorithmen steht der Pfad bereits mit der Initial-Lösung fest, sie verfolgen ihn also in deterministischen Schritten.

Beam Search:

Beam Search geht nicht von einem Punkt im Suchraum, sondern von einer Menge von möglichen Lösungen aus. Diese Punkte werden durch eine *Bewertungsmatrix* bestimmt und Strahl (*Beam*) genannt, alle anderen werden verworfen. Dieses Vorgehen ähnelt der Suche bei Genetischen Algorithmen bzw. Genetischer Programmierung. Hier wird nur von einer *Population* (anstelle des Strahls) gesprochen und die Bewertungsmatrix wird *Fitnessfunktion* genannt. Die Ansätze unterscheiden sich allerdings in der Wahl der Suchoperatoren.

3.3.1.4 Lernvorgänge

Für den eigentlichen Lernalgorithmus kann im Wesentlichen zwischen drei Ansätzen unterschieden werden:

Überwachtes Lernen (Supervised Learning):

Diese Methode vergleicht den vom Lernsystem erzeugten Ausgabewert mit dem gewünschten Wert. Das Ziel besteht somit im Erlernen von Paaren aus Ein- und Ausgangswerten. Entsprechend der Abweichung von der Vorgabe können „Bestrafungen“ oder „Belohnungen“ verteilt werden, die zu einer entsprechenden Strukturveränderung in der Repräsentation des Lernalgorithmus führen. Anwen-

dung findet das überwachte Lernen zum Beispiel in Gesichts- und Bilderkennung.

Unüberwachtes Lernen (Unsupervised Learning):

Bei diesem Vorgehen wird dem Algorithmus kein Ziel vorgegeben, sondern die Aufgabe besteht darin, in den Eingabewerten Strukturen zu finden, also beispielsweise diese zu klassifizieren. Dabei sorgt der Algorithmus selbständig für die Einteilung der Eingabewerte.

Lernen durch Verstärkung (Reinforcement Learning):

Das Verstärkungslernen liegt zwischen dem überwachten und unüberwachten Lernen. Zwar werden hier Überprüfungskriterien an die erzeugte Ausgabe angelegt, aber diese sind unspezifischer als normalerweise bei überwachtem Lernen üblich. Diese Beurteilung wirkt sich wiederum auf die Struktur der Repräsentation aus. Anwendungsfelder für das unüberwachte Lernen und das Verstärkungslernen liegen z.B. im Bereich des *Data Mining*, wo es darum geht, in großen Datenmengen herauszufinden, ob es dort überhaupt Strukturen oder Klassifizierungen gibt.

3.3.2 Biologische Grundlagen zu Evolutionären Algorithmen

Ähnlich wie die Künstlichen Neuronalen Netze beruhen die Evolutionären Algorithmen auf einem Vorbild aus der Natur, in diesem Fall der von Darwin postulierten *Evolutionstheorie*. Seitdem die grundlegenden Mechanismen bekannt sind, durch die die Vererbung und Anpassung von Organismen realisiert werden, ist eine Ausnutzung des Prinzips von Reproduktion und Selektion möglich geworden. Erst dadurch wurde klar, *wo* und *wie* die Informationen über das Aussehen (*Phänotyp*) eines Individuums gespeichert werden, und dass gezielte Veränderungen dieser Information zu einer Veränderung des Organismus und somit zu dessen Tauglichkeit in seiner Umwelt führen. Anpassung findet durch Veränderung der Erbinformation (*Genotyp*) statt.

Auch wenn bei Evolutionären Algorithmen die Motivation nicht immer im tieferen Verständnis biologischer Informationsverarbeitung, sondern in deren ingenieurwissenschaftlicher Anwendung besteht, hilft die Kenntnis des biologischen Vorbildes die algorithmischen Vorgänge besser nachvollziehen zu können. Die in diesem Zusammenhang notwendigen Grundbegriffe werden in diesem Abschnitt erläutert.

3.3.2.1 Evolution

Charles Darwin postulierte 1859 in seinem Buch *On the Origin of Species by Means of Natural Selection* [DAR59], dass die heutige Artenvielfalt der Natur nicht durch einen einmaligen Schöpfungsakt Gottes entstanden sei, sondern dass sich Organismen und Lebewesen in einem ständigen Prozess der Anpassung an die Umweltbedingungen verändern. Das ist durch die Weitergabe und Veränderlichkeit erblicher Eigenschaften möglich. Zufällige Veränderungen der Erbinformation sorgen für Diversität und die Umweltbedingungen bestimmen (*selektieren*), welche Individuen überleben und ihr Erbmaterial weitergeben können.

Einen solcher evolutionären Prozess in einer Population von Individuen hat folgende Voraussetzungen:

1. Ein Individuum einer Population muss in der Lage sein, sich zu reproduzieren.
2. Durch die Reproduktion wird auch die Erbinformation weitergegeben.
3. Ein reproduziertes Individuum muss seinem Original nicht unbedingt vollständig gleichen.
4. Die Individuen stehen in Konkurrenz zueinander, da die für die Reproduktion notwendigen Ressourcen begrenzt sind.

Da nur die angepassten Individuen dauerhaft eine hohe Überlebenswahrscheinlichkeit haben, wird sich die Population im Laufe von Generationen nicht bloß verändern, sondern weiterentwickeln und bezüglich der Umwelt optimieren. Der Grad der Anpassung einzelner Individuen wird dabei auch als *Tauglichkeit* oder *Fitness* bezeichnet.

3.3.2.2 Genetik

Der Bauplan eines Individuums (Organismus) steht im Erbmolekül DNS (Desoxyribonucleinsäure). Die Interpretation (*Expression*) der auf diesem Molekül kodierten Information bestimmt die Entwicklung des *Phänotyps* (*Ontogenese*). Das Genom, welches beim Menschen aus 46 Chromosomen besteht, ist die Gesamtheit aller Erbinformationen, die in jeder Zelle als Kopie abgelegt wird. Dabei entwickelt sich in einem Organismus nicht jede Zelle gleich, sondern spezialisiert sich im Laufe des Wachstums auf eine bestimmte Aufgabe (z. B. Muskel-, Nerven- oder Knochenzellen). Im Laufe der Evolution wurden einige Stellen des genetischen Kodes nicht mehr benötigt. Diese Stellen (*Introns*) werden bei der Expression der DNS durch Marker erkannt und übersprungen. Erstaunlicherweise treten in einigen evolutionären Algorithmen – und insbesondere bei genetischer Programmierung – die gleichen Phänomene auf. Es ist allerdings – sowohl bei der künstlichen, als auch bei der natürlichen Evolution – nicht geklärt, ob diese überflüssige Information nicht doch andere sinnvolle Aufgaben hat.

3.3.2.3 Aufbau der DNS

Die DNS stellt eine Ansammlung von Nukleotiden Basenpaaren dar. Die menschliche DNS besteht zum Beispiel aus 2,8 Milliarden dieser Paare. Es gibt vier verschiedene Basen, die in der DNS vorkommen und durch ihre Anfangsbuchstaben repräsentiert werden: (A)denin, (C)ytosin, (G)uanin und (T)hymine.

Die DNS besteht aus zwei zu einer Doppel-Helix ineinander gedrehten Strängen (siehe Abbildung 3.3.2.3). Der Begriff Basenpaar bezieht sich dabei auf zwei Basen, die durch eine Wasserstoffbrücke verbunden sind und in gegenüberliegenden Strängen liegen. Von 16 möglichen Kombinationen der Basenpaare sind lediglich die Bindungen von A-T und C-G (bzw. andersherum) stabil. Die Natur hat an

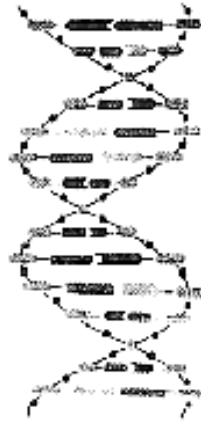


Abbildung 6: Doppelhelix der DNS

dieser Stelle einen wirksamen Schutz gegen Reproduktionsfehler (*Mutation*) eingebaut, da sich von Adenin auf einem der beiden Stränge automatisch auf das komplementäre Thymin auf dem anderen Strang schließen lässt. Zur Reproduktion wäre also auch nur einer der beiden Stränge ausreichend.

Ein *Codon* stellt eine Gruppe von 3 Basenpaaren (*Triplett*) dar, die eine Aminosäure kodieren. Auch hier besteht ein Schutz vor Reproduktionsfehlern. Die 64 möglichen Kodierungen eines Codons werden nur zu 20 verschiedenen Aminosäuren übersetzt. Somit wird eine Aminosäure durchschnittlich mit 3 verschiedenen Kodierungen beschrieben. Zum Beispiel stehen die Kombinationen CAG und CAA beide für die Aminosäure Glutamin.

Aminosäuren sind Bestandteile von Proteinen, die lebenswichtige Aufgaben in einem Organismus wahrnehmen. Proteine sind zum Beispiel für den Transport von Sauerstoff in den Körper und dessen Speicherung verantwortlich, leiten Nervenimpulse an Nervenzellen weiter, übernehmen wesentliche Teile der Immunabwehr und spielen auch bei der Reproduktion der DNS eine wichtige Rolle. Als katalysatorisch wirkende *Enzyme* beschleunigen sie chemische Vorgänge in den Zellen. Wichtig sind Proteine bei der Expression des Genoms in den Phänotyp. Hierbei findet keine direkte Übersetzung statt, sondern aus den Gensequenzen der DNS werden zunächst nur Proteine hergestellt. Diese steuern dann sowohl den weiteren Ablauf der Genexpression, wie auch die Vorgänge, die zum Aufbau des Genotyps führen. Der Weg vom Geno- zum Phänotyp führt dabei über so viele Schritte, dass es schwierig ist, eine eindeutige Beziehung der Gene zum Organismus herzustellen, sondern nur zu den Proteinen, die im Zusammenspiel mit anderen Proteinen die einzelnen Aufgaben übernehmen.

3.3.3 Evolutionäre Algorithmen

Evolutionäre Algorithmen ist der Sammelbegriff für eine ganze Klasse von Algorithmen, die sich alle in unterschiedlicher Art und Weise am Vorbild der natürlichen Evolution orientieren. Die wichtigsten, und im weiteren Verlauf näher

erläuterten, Vertreter sind die *Genetischen Algorithmen* (GA), die *Evolutionssstrategien* (ES) und die *Genetische Programmierung* (GP). Dem letzten Punkt ist, aufgrund der Relevanz für diese Projektgruppe, das ganze folgende Kapitel gewidmet.

Ein Blick in die Natur zeigt, dass die Evolution in der Lage ist, biologische Meisterleistungen zu vollbringen. Der menschliche Organismus ist nur ein herausragendes Beispiel dafür. Allerdings wird immer noch darüber diskutiert, ob Evolution tatsächlich als ein Optimierer bezeichnet werden kann oder nicht.

Die Evolution als Vorbild für Optimierungs- und Adaptationsprozesse zu wählen liegt nahe, sieht man sich die durch diesen Prozess hervorgebrachten Organismen an. Es sind hochkomplexe Systeme, die in der Lage sind, schwierige Aufgaben zu bewältigen, um ihre Systemintegrität zu erhalten und somit ihr Überleben in ihrer Umwelt – teilweise sogar unter wechselnden Bedingungen – zu sichern. Für die Informatik und auch für andere Wissenschaften ist es von Bedeutung, ob das Kopieren bzw. das Übertragen der evolutionären Idee auf den Computer dabei helfen kann, Maschinen für den täglichen Einsatz zu optimieren oder neue Ansätze für schwer lösbare Probleme zu liefern. Evolutionäre Algorithmen zeigen, dass dies möglich sein kann. Basierend auf einfachen Methoden wie Vererbung (*Conservation*) und Mutation (*Innovation*) – weisen Evolutionäre Algorithmen ein Potenzial zur Lösung komplexer Fragestellungen auf. Diese Verfahren sind in umgekehrter Richtung ebenso interessant für die Abstraktion der natürlichen Evolutionsvorgänge und die Erforschung der allgemeinen Bedingungen evolutiver Systeme durch Simulation.

Evolutionäre Algorithmen bedienen sich einer *genetischen Repräsentation*. Die zu lernende oder zu optimierende Information wird in einem *Genom* oder *Chromosom* kodiert. Die Bezeichnung ist hier nicht einheitlich festgelegt und steht lediglich für die Datenstruktur, die die Erbinformation enthält. Ein solches Genom kann man sich beispielsweise als eine Ansammlung von Bits, die in einer fest zu definierenden Kodierung zu interpretieren und zu bewerten sind, vorstellen. Dieses Genom wird in jedem Evolutionsschritt interpretiert und bezüglich seines Verhaltens in einer „Umwelt“ (meistens die zu lösende Aufgabe) bewertet. Diese dadurch bestimmte *Fitness* legt die Überlebens- und Reproduktionsfähigkeit des Individuums für den nächsten Durchlauf der Evolutionsschleife fest. Anhand der Fitnesswerte wird *selektiert*. Normalerweise arbeiten Evolutionäre Algorithmen mit Hilfe einer Population von möglichen Lösungen (Beam Search), wobei die Wahl der Populationsgröße u.U. einen entscheidenden Faktor für den Erfolg darstellt.

Die Anpassung der Individuen an die Umwelt erfolgt durch die Anwendung *genetischer Operatoren* auf die Genome der Individuen. Die wichtigsten Operatoren sind *Reproduktion*, *Mutation* und *Crossover*¹². Diese Operatoren sorgen für die Vervielfältigung und zufällige Veränderung der Genome. Während Mutationen dafür sorgen sollen, andere – evtl. sogar weit entfernte – Stellen des Suchraumes anzuspringen, soll mit Hilfe des Crossovers die Vererbung charak-

¹² Entspricht dem Begriff der *Rekombination* aus der Genetik. In allen Evolutionären Algorithmen wird aber üblicherweise durchgängig von Crossover gesprochen.

teristischer Merkmale erreicht werden. Hiermit sollen möglichst die Bestandteile weitergereicht werden, die als Bausteine (*Building Blocks*) gemeinsam zu einer besseren Lösung führen. Die Anwendung der genetischen Operatoren stellt eine weitere Menge Parameter zur Verfügung, die je nach Problemstellung entscheidend für den Erfolg des Algorithmus sind. Hohe Mutationsraten lassen den EA beispielsweise einer blinden Suche ähnlich werden, wogegen eine zu geringe Mutationsrate zu Stillstand oder einem „Festhängen“ in lokalen Optima führen kann. Deswegen benutzen einige Evolutionäre Algorithmen eine Anpassung der Parameter, z. B. für die Mutation, an den Fortschritt der Lösungssuche.

Der Erfolg Evolutionärer Algorithmen ist stark abhängig von der Wahl der Parameter, der Kodierung und der Bewertungsfunktion. Eine einfache oder gar auf die meisten Problemstellungen zu verallgemeinernde Vorgehensweise ist in der Regel nicht möglich. Die theoretische Erforschung dieser Algorithmen steht erst am Anfang¹³. Einige praktische Probleme lassen sich auf allgemeine und bekannte theoretische Probleme abstrahieren, für die wiederum Vorgehensweisen und Einstellungen bekannt sind. Oft aber ist viel Fingerspitzengefühl, Erfahrung und Zeit zum Experimentieren erforderlich. Das heißt aber nicht, dass die Anwendung Evolutionärer Algorithmen grundsätzlich mit Nachteilen verbunden wäre. Zum einen sind viele Probleme bekannt, deren optimale Lösung durch die kombinatorische Explosion der Lösungswege auf herkömmliche Weise nicht zu berechnen wäre und für welche ein EA gute Approximationen liefern kann. Andererseits führt gerade die immer häufigere Benutzung zu einem besseren Verständnis und somit zu einer immer besseren Anwendbarkeit.

3.3.3.1 Evolutionsstrategien

Im Fall der *Evolutionsstrategien* wird ein Individuum folgendermaßen definiert:

$$\vec{g} = (\vec{p}, \vec{s}) = ((p_1, p_2, \dots, p_n), (s_1, s_2, \dots, s_n)) \text{ mit } p_i, s_i \in \mathbb{R}. \quad (1)$$

Ein Individuum wird somit durch zwei gleichlange Vektoren \vec{p} und \vec{s} repräsentiert, wobei \vec{p} die *Objektparameter* und \vec{s} die *Strategieparameter* enthält. Die Objektparameter enthalten die eigentliche Kodierung des Problems. Sie können z. B. die gesuchten Werte für die optimale Beschaffenheit einer Überschalldüse oder einer Reglereinstellung eines Regelkreises sein. Aus diesem Umfeld der ingenieurtechnischen Probleme stammen die ersten Ideen zu den Evolutionsstrategien von Ingo Rechenberg und Hans-Paul Schwefel in den 60er und 70er Jahren (siehe u.a. [REC73, SCH95]).

Die Strategieparameter werden zur Steuerung des Evolutionsablaufes benötigt. Eine *Mutation* ist die Addition eines zufällig normalverteilten Wertes um den Erwartungswert 0 und mit einer Standardabweichung s . Die Standardabweichung zu jedem Objektparameter p_i wird durch den jeweils zugehörigen Strategieparameter s_i bestimmt. *Rekombinationen* werden bei den Evolutionsstrategien nicht

¹³ Siehe u.a. den Sonderforschungsbereich 531, Design und Management komplexer technischer Prozesse und System mit Methoden der Computational Intelligence, am Lehrstuhl Systemanalyse, Universität Dortmund, <http://sfbCI.informatik.uni-dortmund.de>.

so intensiv genutzt wie die Mutationen. Eine Rekombination ist eine Zusammensetzung *eines* neuen Vektors aus einem oder mehreren anderer Vektoren. Dazu wird an jeder Position i entweder ein gemittelter oder ein zufällig gewählter Wert der beteiligten Vektoren an dieser Position ermittelt.

Das allgemeine Ablaufschema einer Evolutionsstrategie ist:

1. Initialisiere die Population mit μ Vektoren aus zufälligen Werten.
2. Wähle *zufällig* (gleichverteilt) λ Individuen (größer oder gleich der Populationsgröße μ).
3. Wende Rekombination und Mutation an.
4. Bewerte die Individuen.
5. *Selektiere* (aus den neuen oder aus allen Individuen) die λ *besten* Individuen.
6. Wenn Abbruchkriterium nicht erfüllt, beginne wieder mit 2.

Die Evolutionsstrategie lässt sich anhand der Werte für die Zahl der Eltern (μ) und der zu erzeugenden Nachkommen (λ) (wobei immer $\lambda \geq \mu$ gilt) und dem gewählten Selektionsschema charakterisieren. So werden bei einer (3 + 5)-ES (*Plus-Strategie*) 5 Nachkommen aus 3 Eltern erzeugt und nach der Anwendung der genetischen Operatoren die 3 besten Individuen aus allen 8 Individuen selektiert. Dabei ist offensichtlich, dass der Selektionsdruck steigt, je mehr Nachkommen im Verhältnis zu den Eltern erzeugt werden, also je kleiner der Quotient $\frac{\mu}{\lambda}$ wird. Bei *Komma-Strategien* (also hier: (3,5)-ES) hingegen werden die Individuen der nächsten Generation *nur* aus den Nachkommen ermittelt.

Auf diesem Schema basierend lassen sich nun viele Veränderungen und Verbesserungen vorstellen. So ist es beispielsweise möglich, dass auch der Vektor der Strategieparameter evolutiv verändert wird (*Evolution zweiter Ordnung*), um eine Anpassung der Schrittweiten im Suchraum zu erreichen. Ein anderer Vorschlag ist die *Metaevolution*, bei der parallel mehrere Populationen evolviert werden, die – von der Meta-Ebene aus betrachtet – wiederum wie Individuen behandelt, also selektiert und verändert werden können¹⁴.

Wichtig für den Vergleich mit anderen Evolutionären Algorithmen ist es festzuhalten, dass

- ES nicht primär Wert auf eine genaue Äquivalenz zum biologischen Vorbild legen.
- die Selektion der Individuen für die Anwendung von Crossover und Mutation zufällig erfolgt.
- nach der Bewertung nur die besten übernommen werden.

¹⁴ Parallelität wird aber nicht erst durch Metaevolution möglich. Auch „normale“ ES, die auf Populationen basieren, arbeiten bereits *inhärent parallel*.

3.3.3.2 Genetische Algorithmen

Genetische Algorithmen (GA) gehen auf John Holland zurück, erste Arbeiten zu diesem Thema sind Mitte der 70er Jahre parallel zu und unabhängig von den Evolutionsstrategien entstanden. In ihrer Struktur können Genetische Algorithmen als Grundlage der Genetischen Programmierung verstanden werden. Im Gegensatz zu den Evolutionsstrategien sind die GAS näher am Vorbild der Genetik orientiert und dienten John Holland auch zunächst als abstraktes Modell biologischer Informationsverarbeitung.

Genetische Algorithmen kennzeichnen sich durch zwei entscheidende Merkmale. Zum einen ist das Chromosom als Darstellungstyp auf einen binären String fester Länge fixiert und zum anderen wird als Vererbungsoperator das Crossover sehr häufig angewandt (im Normalfall entstehen etwa 95% aller Individuen einer Population entweder durch Crossover oder einfaches Kopieren).

Der Standard-GA verwendet Chromosomen der Form

$$\vec{x} = (x_1, x_2, \dots, x_n) \in M^n = \{0, 1\}^n, \quad (2)$$

also binäre Vektoren x aus der n -elementigen Grundmenge M . Die x_i werden auch als *Gene* bezeichnet, die Grundmenge M als Menge der *Allele*.

Die Festlegung auf eine bestimmte Länge bzw. Anzahl der Bits des Chromosoms lässt erkennen, dass die Kodierung des Problems weniger intuitiv als bei einer ES ist. Während eine ES eine *kompakte* Kodierung in reellen Zahlen vornimmt, ist die Kodierung bei GAS *breiter*. Jedes Bit oder eine Gruppe von Bits spiegelt ein bestimmtes Merkmal der Lösung wieder. Das Kodieren bzw. das Auffinden oder die Erstellung einer sinnvollen Kodierung ist eine der entscheidenden Aufgaben beim Design eines GA.¹⁵

Der kanonische GA lässt sich mit folgendem Ablauf darstellen:

1. Initialisiere die Population mit zufälligen Binärvektoren.
2. Interpretiere und bewerte die Individuen (Fitness)¹⁶.
3. Selektiere Individuen zur Reproduktion und Veränderung.
4. Führe die genetischen Operatoren auf den Chromosomen aus.
5. Entscheide über die Zusammensetzung der nächsten Generation und fahre mit Punkt 2 fort, bis das Abbruchkriterium erfüllt ist.

¹⁵ Darüber hinaus sind auch andere *diskrete* Kodierungen denkbar. Beispielsweise schlägt Jacob in [JAC97] eine der Genetik noch näher kommende dreiwertige Kodierung mit polyploiden Chromosomensätzen vor.

¹⁶ Strenggenommen unterscheidet man bei GA zwischen *Bewertung* und *Fitness*. Die Bewertung ist die Interpretation des Individuums (z. B. Funktionswert der zu optimierenden Funktion), welche nicht notwendig mit dessen Fitness identisch sein muss (es könnten beispielsweise „Bestrafungen“ für zu niedrige Ergebnisse erfolgen). Der Einfachheit halber gehen wir aber im Folgenden von einer Identität der Begriffe aus.

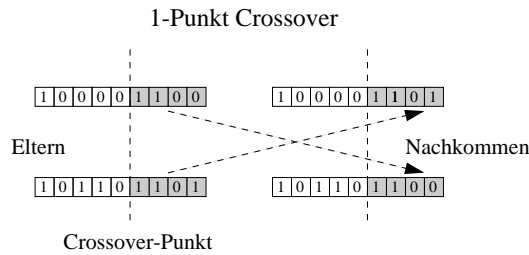


Abbildung 7: Crossover

Der Ablauf eines GA unterscheidet sich von dem der ES vor allen Dingen in der Selektion. Während bei einer ES per Zufall entschieden wird, welche Individuen reproduziert und verändert werden, werden dazu in einem GA nur Individuen mit einer entsprechenden Fitness verwendet. Das bedeutet z. B. bei einer *fitnessproportionalen Selektion* eine Auswahlwahrscheinlichkeit, die in Abhängigkeit zur ermittelten Fitness steht. Dabei wird das sogenannte *Roulette-Prinzip* angewendet: Anschaulich teilt man jedem Individuum der vorhandenen Population ein Tortenstück auf einer Scheibe zu. Die Größe des Tortenstücks fällt proportional zum Fitnesswert des Individuums aus. Danach wird die Scheibe gedreht, das Individuum ausgewählt, auf das die Kugel fällt, und in die nächste Population übernommen. Darüberhinaus sind natürlich viele Verbesserungen oder auch ganz andere Selektionsmöglichkeiten denkbar.

Anschließend werden auf Individuen der neuen Population Crossover und Mutation angewendet. Das Ein-Punkt-Crossover ist schematisch in Abbildung 7 dargestellt.

Bezogen auf eine zufällig gewählte Position werden einfach die Teilketten nach dieser Position ausgetauscht, sodass zwei neue Individuen entstehen. Dieses Vorgehen lässt sich auf 2 bis n-Punkt-Crossover erweitern, indem immer abwechselnd zwischen zwei zufällig ermittelten Positionen getauscht wird oder nicht.

Bei der Mutation wird lediglich an jeder Position des Chromosoms zufällig entschieden, ob das Bit invertiert wird oder nicht.

Wurden diese Operatoren angewendet, muss noch entschieden werden, wieviele davon die neue Generation für den nächsten Durchlauf der Evolutionsschleife bilden. Im *Generational* GA bilden entweder alle veränderten Nachkommen oder aber ein großer Teil davon die Population für den nächsten Durchlauf. Da die Populationsgröße konstant bleibt, können z. B. die besten Individuen der Elterngeneration in die neue Generation übernommen werden. So ist auch für den Erhalt guter Lösungen gesorgt. Im entgegengesetzten Fall – dem *Steady-State* GA – kann von einem Generationenbegriff kaum mehr gesprochen werden. Nur eine sehr geringe Zahl oder auch nur 1 Individuum werden reproduziert und verändert. Dieses ersetzt anschliessend das schlechteste Individuum der Elterngeneration.

Auch bei den GA werden viele Variationen und Verbesserungsvorschläge dis-

kutiert¹⁷. Es gilt aber hier ebenso, dass noch keine einheitliche Theorie über die Anwendbarkeit und die Erfolgswahrscheinlichkeit Genetischer Algorithmen existiert. Eine Möglichkeit dazu sind das von John Holland entwickelte *Schema-Theorem* [HOL75] und Dave Goldbergs *Building-Block-Hypothese* ([GOL89]). *Building Blocks* kann man sich als Bausteine vorstellen. So wie ein Kind einen Turm aus einzelnen Bausteinen zusammensetzt, stellt man sich vor, besteht auch eine gute Lösung eines GA aus kleineren funktionellen Einheiten. Um diese These zu erhärten, muss gezeigt werden, wie solche Blöcke aussehen, und unter welchen Bedingungen sie sich am besten durchsetzen können. Dazu dient das *Schema-Theorem*. Beides wird in den folgenden Abschnitten kurz vorgestellt und dient bei den Betrachtungen zur Genetischen Programmierung als Ausgangspunkt.

3.3.3.3 Schemata

Für Schemata wird die Grundmenge M um das Variablensymbol $\#$ („don’t care“) erweitert. Es kann die Werte 0 oder 1 annehmen. Ein Schema ist somit ein Vektor \vec{x} , für den gilt:

$$\vec{x} = (x_1, x_2, \dots, x_n) \in M^n = \{0, 1, \#\}^n. \quad (3)$$

Beispielsweise ist $h = (1, 0, \#, \#, 1, 0)$ ein Schema der Länge $n = 6$. Mathematisch betrachtet definiert es einen *Unterraum* des M^6 . Allgemein definiert ein Schema der Länge n jeweils einen Unterraum¹⁸ des M^n .

Ein Chromosom \vec{x} enthält keine Variablensymbole und wird als *Instanz* eines Schemas bezeichnet. Da es für den M^n genau 3^n Schemata gibt, stellt jedes Chromosom (x_1, \dots, x_n) eine Instanz von 2^n Schemata dar und liegt damit gleichzeitig in 2^n Unterräumen des M^n . Eine Population von p Chromosomen repräsentiert daher zwischen 2^n und $p * 2^n$ Schemata. Goldberg argumentiert darauf aufbauend, dass bei einer solchen Population näherungsweise $p^3 * (O(p^3))$ Schemata indirekt verarbeitet werden.¹⁹ Dies dient den GA-Theoretikern zur Begründung der Suchmächtigkeit der Genetischen Algorithmen. Es findet nicht nur eine *direkte* Suche im Lösungsraum statt, sondern gleichzeitig auch eine *indirekte*, die parallel die Güte von $O(p^3)$ Unterräumen überprüft. Goldberg bezeichnet dies als *impliziten Parallelismus*.

3.3.3.4 Schema-Theorem

In Bezug auf Populationen von Individuen, die einem Evolutionsprozess unterliegen und der Optimierung dienen, stellt sich die Frage, wie die *Überlebenschancen* von *Funktionsblöcken* – aus denen die Individuen bestehen – beschrieben werden können. Ist sichergestellt, dass gewonnene Informationen nicht wieder

¹⁷ So sind z. B. weitere aus der Genetik entlehnte Operatoren, wie *Inversion*, *Deletion* etc., vorstellbar (Siehe [JAC97]).

¹⁸ Anschaulicher und unter Verlust der Exaktheit verwendet man stattdessen oft den Begriff der *Hyperebene*.

¹⁹ Für die genaue Herleitung der Abschätzung siehe [GOL89].

zerstört werden? Welche Eigenschaften haben gute Funktionsblöcke? Durch die Beschreibung von Funktionsblöcken als Schemata lassen sich nun quantitative Abschätzungen formulieren, deren grundsätzliche Idee hier vorgestellt werden soll. Für eine genaue Herleitung siehe vor allem [GOL89].

Das Schema-Theorem in seiner formalen Definition lautet:

$$m(h, t + 1) \geq m(h, t) * \frac{f(h)}{f_{\text{mittel}}} * \left(1 - p_c * \frac{L(h)}{n - 1} - p_m * o(h) \right) \quad (4)$$

mit

$m(h, t)$	Instanzen eines Schemas h in einer Population zum Zeitpunkt t .
$f(h)$:	Durchschnittliche Fitness aller Instanzen von h zur Zeit t .
f_{mittel} :	Durchschnittliche Fitness der <i>gesamten</i> Population zur Zeit t .
p_c, p_m :	Wahrscheinlichkeit eines Crossovers oder einer Mutation, bezogen auf ein <i>Chromosom</i> .
$o(h)$:	<i>Ordnung</i> eines Schemas h : Anzahl der Positionen, die nicht das Variablensymbol $\#$ enthalten.
$L(h)$:	<i>Definierende Länge</i> eines Schemas h : Abstand zwischen der ersten und letzten von $\#$ verschiedenen Position. Der Abstand ist 0, bei weniger als zwei Positionen ungleich $\#$.
n	Absolute Länge des Schemas.

Diese Formel macht eine Aussage über die Anzahl der Instanzen eines Schemas im nächsten Zeitschritt $t + 1$: Diese Anzahl hängt proportional ab vom *Selektionsoperator* $\frac{f(h)}{f_{\text{mittel}}}$ abzüglich der Wirkungen des Crossovers $p_c * \frac{L(h)}{n-1}$ und der Mutation $p_m * o(h)$. Der negative Einfluss von Crossover und Mutation hängt dabei auch noch von den Eigenschaften der Schemata bezüglich ihrer definierenden Länge und Ordnung ab. So wirkt beispielsweise der Crossover-Operator um so weniger destruktiv, je geringer die relative Länge $\frac{L(h)}{n-1}$ ist.

Qualitativ formuliert drückt diese Formel die Vermutung aus, dass Schemata mit

- überdurchschnittlicher Fitness
- kurzer definierender Länge
- und niedriger Ordnung

in den Folgegenerationen gesteigerte Chancen haben, sich zu reproduzieren, bzw. bei überdurchschnittlich hoher Fitness sogar exponentiell häufiger zum Zuge kommen. Ein GA, so die Vermutung der Theoretiker, gibt somit den in diesem Sinne besseren Schemata eine höhere Chance zu überleben. Wenn dies tatsächlich der Fall ist, hat das auch Konsequenzen für die Überlebensfähigkeit von Instanzen der Schemata – also den eigentlichen Individuen – und führt zur Building-Block-Hypothese.

3.3.3.5 Building-Block-Hypothese

Eine indirekte Schlussfolgerung aus dem Schema-Theorem besagt, dass es sinnvoll ist, inhaltlich zusammengehörende Informationen nicht getrennt auf dem Chromosom abzulegen, sondern in kompakten Segmenten zu kodieren. Solche zusammengehörenden Blöcke können als *Building Blocks* zum Zusammensetzen größerer Blöcke dienen. So werden aus guten (Teil-) Lösungen wie in einem Spielzeugbaukasten immer bessere Lösungen konstruiert.

Diese *Building-Block-Hypothese* legt eine automatische Verbesserung der Individuen, sobald nur genügend sinnvolle Blöcke existieren, nahe. An dieser Stelle ist zu bemerken, dass das Schema-Theorem und die Building-Block-Hypothese zu den umstrittensten Fragen der *Evolutionary-Computation-Forschung* (EC) zählt. Es ist weder erwiesen, dass die durch Formel (4) ausgedrückte Vermutung stimmt, noch, wie ein in diesem Sinne „gutes“ Individuum mathematisch exakt zu definieren ist. Letzteres wäre für einen praktischen Nutzen in der Konstruktion von Optimierungsverfahren mit Hilfe von GA's erforderlich. Es geht nicht aus dem Schema-Theorem hervor, wie „fit“ und wie „kurz“ ein Individuum sein muss, um vom Selektionsprozess bevorzugt zu werden, gesetzt den Fall, die grundlegende Vermutung stimmt. Das Schema-Theorem ist also nicht konstruktiv, sondern beschreibt in mathematischer Formulierung eine Vermutung, warum Optimierungsvorgänge mit Genetischen Algorithmen überhaupt funktionieren.²⁰

3.4 Genetisches Programmieren

Genetisches Programmieren ist die Übertragung des allgemeinen Vorgehens der Evolutionären Algorithmen auf die Verknüpfung von Symbolen zu einem Programm. Seit der Entwicklung der ersten Computer gibt es Überlegungen dazu, wie man die Aufgabe des *Programmierens* dem Computer selbst überlassen kann. Programmieren ist im Allgemeinen nicht trivial und es wird noch viel Zeit vergehen, bis ein Computer sich selbständig auf völlig neue Problemsituationen einstellen kann. Noch ist Programmierung reine Handarbeit, auch wenn etliche Programm-Assistenten einem Programmierer in verschiedenen Entwicklungsumgebungen sicherlich schon einiges an Arbeit abnehmen. Allerdings hat dieses Vorgehen wenig mit Genetischer Programmierung zu tun. Dabei wird lediglich mit zuvor angefertigten Schablonen ein bekanntes, parametrisierbares Vorgehen (zum Beispiel Datenbank-Anbindung oder GUI-Erstellung) unterstützt.

Die folgenden Abschnitte (3.4.1–3.4.4) stellen das Genetische Programmieren als einen evolutionären Algorithmus – so wie diese im letzten Kapitel dargestellt wurden – vor. Die letzten Abschnitte dieses Kapitels (3.4.5–3.4.7) gehen dann genauer auf spezielle Probleme ein, die sich bei GP stellen, und zeigen Verbesserungs- und Veränderungsmöglichkeiten auf.

²⁰ Neuere Untersuchungen weisen sogar auf einen Fehler im Schema-Theorem hin (siehe [MEN97]).

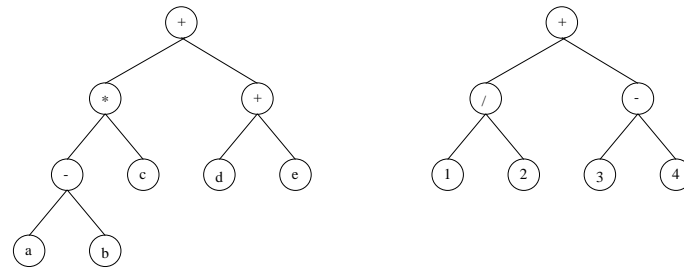


Abbildung 8: Baumstrukturen

3.4.1 Repräsentation

Im Genetischen Programmieren werden drei Repräsentationsmöglichkeiten mittlerweile am häufigsten benutzt: Programme werden als *Baum*, *Graph* oder in *linearer* Form repräsentiert. Dabei wird sehr oft die von Koza in [koz92] vorgestellte Baumstruktur verwendet. Natürlich sind aber auch hier wieder Vermischungen dieser Strukturen oder ganz neue Vorschläge möglich.

Im Gegensatz zu Genetischen Algorithmen ist die Länge eines GP-Genoms nicht festgelegt, sondern variabel. Dadurch ist es dem Genom möglich, sich der Komplexität des Problems anzupassen. Andererseits steigt dadurch die Gefahr der Entstehung „unsinnigen“ oder überflüssigen Codes. Dennoch ist der Flexibilitätsgewinn gegenüber Genetischen Algorithmen ein wichtiger Vorteil. Das Ziel bei GP ist die Erkundung des Suchraums der möglichen *Programme*, die ein bestimmtes Problem lösen können, und nicht der Suchraum der richtigen Parameter einer Funktion oder Aufgabenstellung. Deswegen wäre es ungünstig, die Programmgröße von vorneherein zu fixieren.

Das Genom als Grundlage des Phänoms kann dabei verschiedene Methoden und Operationen enthalten, wie zum Beispiel arithmetische Funktionen (+, *), mathematische Funktionen (sin, exp), boolesche Funktionen (AND, OR), Fallunterscheidungen (IF, SWITCH), Schleifen (FOR, WHILE)²¹ und andere anwendungsspezifische Funktionen. Ebenso lassen sich Terminalzeichen kodieren, also Variablen oder Konstanten. In den folgenden Abschnitten wird die Menge F als die Menge aller erlaubten Funktionen und die Menge T als die Menge aller erlaubten Terminale angesehen. Die Reihenfolge der Abarbeitung der zufällig zusammengesetzten Befehle wird durch die gewählte Struktur determiniert.

3.4.1.1 Baumbasierte Genetische Programmierung

Als Evolutionärer Algorithmus sucht auch die Genetische Programmierung parallel mit einer Population nach möglichen Programmen. Initialisiert wird ein Anfangsprogramm zufällig. Dabei stellt sich bei diesem Ansatz jedes Programm

²¹ Schleifen werden bei der Genetischen Programmierung nur selten verwendet, da sie bestimmten Nebenbedingungen genügen müssen, um nicht zu Endlosschleifen zu werden. Eine Möglichkeit der Verwendung von Schleifen zeigen wir für unser Schachproblem in 77.

als ein Baum dar, der in der Wurzel ein Element der Menge F hat. Dadurch wird ermöglicht, dass Programme hierarchisch aufgebaut sind, und sich dynamisch weiterentwickeln können. Befindet man sich in einem Knoten, der mit einer Funktion f_i beschriftet ist, so gehen $z(f_i)$ Kanten von diesem Knoten aus, wobei z eine Funktion ist, die die Anzahl der Parameter der Funktion spezifiziert. Wird ein Knoten mit einem Element t_i aus T beschriftet, so stellt er sich als Blatt dar und somit ein vorläufiges Ende der jeweiligen übergeordneten Funktion. Die Individuen der Initial-Population können auf verschiedene Arten erzeugt werden. Die *Full-Methode* erzeugt ausgewogene Bäume, bei denen alle Blätter den gleichen Abstand zur Wurzel aufweisen. Die *Grow-Methode* erlaubt hingegen auch Blätter mit einer geringeren Tiefe als die vorgegebene maximale Tiefe. Verfahren, die die beiden genannten Methoden kombinieren, werden häufig verwendet. Dabei werden Bäume mit verschiedenen Höhen erzeugt, die durch die Full- und die Grow-Methode erstellt werden. Um die Diversität nicht schon im Ausgangspunkt zu beeinträchtigen, wird bei vielen Ansätzen der Genetischen Programmierung darauf geachtet, dass keine Duplikate vorkommen, die besonders bei kleinen Ausgangsbäumen oder kleinen Mengen T und F mit wenigen Elementen gehäuft auftreten. Genauso sollte darauf geachtet werden, dass alle Initial-Individuen einen möglichst ähnlichen Fitness-Wert aufweisen, da ansonsten die besseren Individuen bereits nach wenigen Generationen dominieren und somit die Divergenz der Population senken. Allerdings können wenige Ausreißer oftmals zu sehr guten Ergebnissen führen.

3.4.1.2 Lineare Genetische Programmierung

Diese Art der Darstellung stellt Programme in Form einer linearen Sequenz dar. Sie wird häufig verwendet, wenn man auf Programmiersprachen wie LISP²² verzichten möchte. Anstelle dessen wird zum Beispiel – auch aus Effizienzgründen – C(++) verwendet. Im Gegensatz zur Baum-Darstellung werden die Genetischen Operationen gleichmäßig auf allen Abschnitten des Programms ausgeführt.²³

3.4.1.3 Graphbasierte Genetische Programmierung

Es gibt verschiedene Arten der *Graphbasierten Genetischen Programmierung*, von denen hier nur das PADO-System ([TV95]) erläutert wird, das von Teller und Veloso entwickelt wurde.

PADO benutzt einen gerichteten Graphen, der aus N Knoten besteht, von denen jeder einzelne N ausgehende Kanten besitzen darf. Jeder Knoten besteht aus einem Aktions- und einem Verzweigungs-Teil. Die Knoten können auch verschiedene, spezielle Aufgaben übernehmen:

²² LISP ist die Standardsprache bei der Verwendung der Baum-Darstellung, da sie dieses Konzept einfach unterstützt.

²³ Zumindest bei vollständigen Bäumen finden sich 50% alle Knoten auf der untersten Ebene und werden somit statistisch am häufigsten bei der Verwendung der Genetischen Operatoren selektiert. Anschaulich ist klar, dass diese Operationen nur geringe Auswirkungen auf das Programm haben. Bei der linearen Programmierung werden hingegen „wichtige“ und „unwichtige“ Abschnitte gleichermaßen ausgewählt.

1. Start-Knoten
2. End-Knoten
3. Knoten, der ein Unterprogramm aufruft
4. Knoten, der ein Unterprogramm einer Bibliothek aufruft

Dabei stehen die Unterprogramme einer Bibliothek allen Knoten zur Verfügung, während „normale“ Unterprogramme nur dem aufrufenden Knoten zuzuordnen sind. Die spezielle Darstellung erlaubt es (im Gegensatz zur Baum-Darstellung), auf einfache Art Schleifen zu implementieren. Auf der anderen Seite ist es relativ kompliziert, Crossover und andere Operatoren zu realisieren.

3.4.2 Fitness

Die Fitness eines Individuums ist das Maß, das bestimmt, wie gut das Individuum die gestellte Aufgabe gelernt hat, also z. B. anhand einer gegebenen Eingabe eine korrekte Ausgabe vorherzusagen. Der Fitnesswert eines Individuums entscheidet über dessen „Überleben“. Eine höhere Fitness erhöht gleichzeitig auch die Chance, in der nächsten Generation aufzutauchen. Fitnessfunktionen können in verschiedenen Formen auftreten, die im Folgenden vorgestellt werden.

Die *Raw Fitness* lässt sich aus dem jeweiligen Problem herleiten. Betrachtet man das sogenannte *Ant Problem*, bei dem eine Ameise auf einem Feld nach Futterstücken suchen soll, so lässt sich die Raw Fitness als Anzahl der gefundenen Futterstücke interpretieren. Die Raw Fitness wird anhand von *Fitness Cases* bestimmt, die im eben genannten Beispiel aus der Menge des gefundenen Futters bestehen, zusätzlich aber zum Beispiel auch die dafür zurückgelegte Strecke umfassen können. Dabei sollten die gewählten Fitness Cases natürlich repräsentativ sein, um als Bewertung herangezogen werden zu können. Die Raw Fitness wird oft als Abweichung von einem optimalen Ergebnis angegeben, die sich als Summe über alle Fitness Cases bildet oder als Abstandsmaß mittels der Quadratwurzel bestimmt wird. Je nach Definition kann die Raw Fitness entweder als kleiner oder als großer Wert besser sein.

$$r(i, t) = \sum_{j=1}^{N_e} S(i, j) - C(j) \quad (5)$$

Dabei bezeichnet $r(i, t)$ die i -te Rawfitness, sofern es mehrere gibt, in der t -ten Generation. N_e ist die Anzahl der betrachteten Fitnesscases. S ist die Bewertungsfunktion des Individuums und C gibt die tatsächliche Fitness an. Die *standardisierte Fitness* definiert die Raw Fitness so, dass immer ein kleinerer Wert ein besseres Individuum ausmacht. Dadurch ergeben sich zwei Fälle:

$s(i, t) = r(i, t)$, falls kleinere Werte der Raw Fitness besser sind.

$s(i, t) = r_{max} - r(i, t)$, falls größere Werte der Raw Fitness besser sind.

r_{max} stellt dabei die maximale Rawfitness eines Individuums dar. Das führt allerdings dazu, dass gute Individuen eine Fitness erhalten, die immer näher

an Null herankommt. Dadurch bedingt sind verschiedene gute Individuen kaum noch zu unterscheiden.

Die *Adjusted Fitness* umgeht dieses Problem, indem die standardisierte Fitness auf einen Wert zwischen 0 und 1 abgebildet wird, wobei ein Wert von 1 die beste Fitness angibt.

$$a(i, t) = \frac{1}{1 + s(i, t)} \quad (6)$$

Für eine proportionale Selektion von Individuen müssen alle Werte der Adjusted Fitness Werte einer Population mittels der *normalisierten Fitness* bearbeitet werden. Dabei wird die Adjusted Fitness des betrachteten Individuums durch die Summe aller Adjusted Fitness Werte geteilt.

$$n(i, t) = \frac{a(i, t)}{\sum_{k=1}^M a(k, t)} \quad (7)$$

M gibt die Anzahl der Fitnesstypen an. Neben der soeben beschriebenen Fitness-Bewertung, die unter anderem die Korrektheit einer Lösung widerspiegelt, lässt sich auch die Effizienz eines erstellten Programms in die Fitness mit einbeziehen.

3.4.3 Selektionsmechanismen

Nachdem die Fitness-Bewertung der Individuen erfolgt ist, muss entschieden werden, ob und in welcher Form diese weiterverwendet werden. Dieses Vorgehen wird Selektion genannt. Es gibt verschiedene Arten der Selektion, die im Folgenden vorgestellt werden.

Die *fitnessproportionale Selektion* wurde bereits bei den Genetischen Algorithmen anhand des Roulette-Prinzips erläutert. Die Wahrscheinlichkeit p_i , dass ein Individuum i in die nächste Generation übernommen wird, ist dabei durch

$$p_i = \frac{f_i}{\sum_j f_i} \quad (8)$$

gegeben, wobei f_i die Fitness eines Individuums i darstellt.

Truncation bzw. (μ, λ) -Selektion²⁴ bezeichnet ein Vorgehen, bei dem μ Eltern λ Kinder erzeugen, von denen wiederum die μ besten ausgewählt werden, die in die nächste Generation gelangen, die Eltern selbst werden dabei verworfen. Bei der $(\mu + \lambda)$ -Selektion haben auch die Eltern die Möglichkeit, in die Folgegeneration übernommen zu werden.

Die *Ranking-Selektion* basiert auf der Reihenfolge der Fitness-Werte der Individuen. Die Selektions-Wahrscheinlichkeit wird dabei durch eine Funktion in

²⁴ Hier wird die Terminologie der Selektionsmechanismen bei den Evolutionsstrategien verwendet (siehe auch Abschnitt 3.3.3.1).

Abhängigkeit der Position innerhalb der Reihenfolge bestimmt. Lineares Ranking wird durch eine lineare Funktion ermöglicht:

$$p_i = \frac{1}{N} [p^- + (p^+ - p^-) \frac{i-1}{N-1}]. \quad (9)$$

Dabei steht $\frac{p^-}{N}$ für die Wahrscheinlichkeit, dass das „schlechteste“ (gemessen am Fitness-Wert) Individuum und $\frac{p^+}{N}$ für die Wahrscheinlichkeit, dass das „beste“ Individuum selektiert wird. Für exponentielles Ranking wird die Funktion

$$p_i = \frac{c-1}{c^N-1} c^N - 1; 0 < c < 1 \quad (10)$$

benutzt.

Die *Tournament-Selektion* arbeitet nicht auf der gesamten Population, sondern nur auf einer kleinen Anzahl von Individuen. Diese Individuen treten in einem *Turnier* gegeneinander an. Dabei werden die schlechteren Individuen ersetzt. Bei einer Größe von zwei wird das schlechte Individuum durch eine Mutation des besseren ersetzt. Der Vorteil dieser Selektionsmethode liegt darin, dass sie parallel ausgeführt werden kann und somit sehr schnell ist.

3.4.4 Der Standard-GP-Algorithmus

3.4.4.1 Vorbereitung

Um einen Algorithmus zu implementieren, müssen folgende Schritte zuvor durchdacht werden:

1. Definition der Menge aller Terminalzeichen.
2. Definition der Menge aller erlaubten Funktionen.
3. Definition der Fitness-Funktion.
4. Festlegung von Angaben wie Populations-Größe, Maximale Größe der Individuen, Crossover-Wahrscheinlichkeit, Selektions-Methode und Abbruchkriterium.

Danach kann mit der Implementierung des eigentlichen Algorithmus begonnen werden.

3.4.4.2 Generationenbasierter Algorithmus

Dieser Algorithmus kann als *Standard-Algorithmus* betrachtet werden. Es werden klar voneinander getrennte Generationen betrachtet. Jede Generation umfasst eine Population von Individuen (Programmen), wobei eine neue Generation aus der bestehenden generiert wird und diese dann ersetzt. Das Vorgehen dabei wird folgend beschrieben:

1. Initialisierung der ersten Population.
2. Bewertung aller Individuen mittels der Fitness-Funktion.
3. Solange die neue Population nicht vollständig ist, werden die Schritte 4–6 wiederholt.
4. Selektion eines oder mehrerer Individuen mittels des Selektions-Algorithmus.
5. Ausführung der Genetischen Operatoren auf die selektierten Individuen.
6. Die so generierten Individuen in die neue Population einfügen.
7. Wenn das Abbruchkriterium nicht erreicht ist, wiederhole die Schritte 3–7 mit der neuen Population.
8. Ausgabe des besten Individuums.

3.4.4.3 *Steady-State-Algorithmus*

Der *Steady-State-Algorithmus* wird auch *Tournament-Algorithmus* genannt und arbeitet mit der Tournament-Selektion. Es gibt keine Generationen, stattdessen werden die Individuen ständig verändert und in die vorhandene Population eingefügt.

1. Initialisierung der ersten Population.
2. Auswahl einer Menge von Individuen des 'Wettkampfes'.
3. Berechnung der Fitness-Werte der Individuen.
4. Selektion der „Gewinner“.
5. Anwendung der Operatoren auf die Gewinner.
6. Ersetzen der Verlierer durch die veränderten Gewinner aus 5.
7. Sofern das Abbruchkriterium noch nicht erfüllt ist, wiederhole die Schritte 2–7.
8. Ausgabe des besten Individuums.

Ein zusammenfassender Überblick findet sich in Abbildung 9 auf der nächsten Seite.

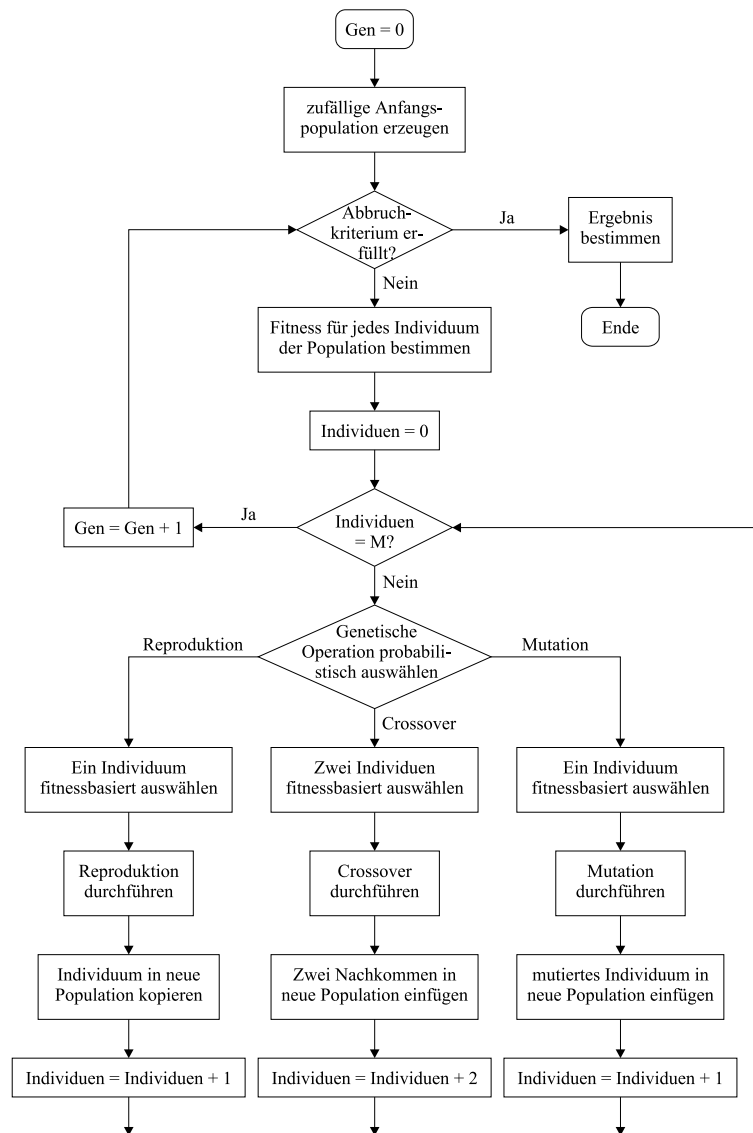


Abbildung 9: Flussdiagramm eines GP-Algorithmus'

3.4.5 Crossover genauer betrachtet

Ein vertiefender Einblick in das *Genetische Programmieren* (GP) geht der Frage nach, was während des Evolutionsvorganges eigentlich geschieht. Es ist ja nicht von vornherein klar, warum und unter welchen Bedingungen GP dazu in der Lage ist, Programme zu produzieren, die die ihnen gestellten Aufgaben lösen können. Dabei spielen der Crossover-Operator und die während eines GP-Laufes entstehenden Introns eine zentrale Rolle. Die Funktionalität der GP-Individuen hängt entscheidend davon ab, wie sie während der künstlichen Evolution weiterverarbeitet werden, und welche Seiteneffekte die genetischen Operatoren haben. Langfristige Verbesserungen der Fitness sind nur möglich, wenn gute Programme nicht durch die Anwendungen von Crossover und Mutation zerstört werden.

Während bei Introns – in einem GP-Lauf entstehende wirkungslose Befehle – untersucht wird, weshalb sie überhaupt entstehen und wie sie am besten zu vermeiden sind, ist der Crossover-Operator eine der treibenden Kräfte in einer künstlichen Evolution mit GP. Zu erkennen ist das z. B. an den sog. *Koza-Tableaus*, einer tabellenförmigen Darstellung aller wichtigen Parameter für einen GP-Lauf, in denen häufig die Crossover-Wahrscheinlichkeit auf 90%, die Mutationswahrscheinlichkeit hingegen auf 5% festgelegt wird. Eine Begründung hierfür liegt in der Analogie zum biologischen Vorbild: Die Natur investiert einen bemerkenswerten Aufwand in den Vorgang der Rekombination.

Die qualitative Vorstellung der Wirkungsweise des Crossovers führt zu der Vermutung, dass bessere Lösungen im Suchprozess eher durch das Zusammenfügen bisheriger guter Lösungen (*building blocks*) entstehen, und nicht lediglich durch kleine Veränderungen im Genom. Diese Sichtweise gilt es nun, theoretisch zu untermauern.²⁵ Es stellt sich die Frage, ob das Crossover tatsächlich besser und anders arbeitet als die Mutation, oder ob es sich lediglich um eine *Makromutation* handelt. Banzhaf et. al. [BNKF98] vermuten: „GP works faster than systems just based on mutations [...], because good building blocks get combined into ever larger and better building blocks to form better individuals.“ Daraus lassen sich dann Schlussfolgerungen für die Verbesserung des Crossovers ziehen. Um festzustellen, wie der Crossover-Operator arbeitet, greift man auf theoretische Arbeiten im GA-Bereich zurück und versucht, diese auf GP anzuwenden.

Die theoretische Untersuchung des Verhaltens von GP gestaltet sich relativ schwierig. Ziel ist es, eine quantitative Vorstellung davon zu bekommen, wie erfolgreiche Programme aussehen und entstehen, und welchen Einfluss die genetischen Operatoren auf diesen Prozess nehmen. Die auf Zufallsmechanismen basierende Arbeitsweise der Operatoren ermöglicht eine breitgestreute Suche im Lösungsraum. Im Gegensatz z. B. zu einfachen Hill-Climbing-Verfahren zeichnet sich GP dadurch aus, an die unterschiedlichsten Stellen des Suchraumes springen zu können, ohne nur einem Pfad zu einer (sub-)optimalen Lösung zu folgen. Gerade dadurch aber werden quantitative Abschätzungen des Verhaltens erschwert.

Desweiteren handelt es sich bei GP um ein Paradigma, das Kombinationen

²⁵ Ein Beweis für die Wirksamkeit des Crossovers findet sich in einer theoretischen Untersuchung von Jansen und Wegener in [JW98].

von *Befehlen* zu einem *Programm* zusammenfügt. Gesucht ist somit ein komplexer Lösungsvektor in einem sehr grossen Suchraum. Zu dieser kombinatorischen Komplexität kommt noch die Unterschiedlichkeit der Auswirkung der Operatoren zu verschiedenen Zeitpunkten des Evolutionslaufes. Um nicht in einem lokalen Optimum steckenzubleiben, sind große Veränderungen wünschenswert. Die Verbesserung einzelner guter Programme erfordert dagegen ein eher kleinschrittiges Suchen, damit die gefundene Näherung nicht wieder zerstört wird.

Deshalb findet die Anwendung der aus der Theorie der *Genetischen Algorithmen* (GA) bekannten Abschätzungen für die Charakteristika erfolgreicher Programmindividuen nur begrenzt statt und führt nur zu Vermutungen. In Kombination mit empirischen Untersuchungen der Ergebnisse von Evolutionsläufen lassen sich zwar keine endgültigen Antworten auf die anfangs gestellten Fragen finden, aber Hinweise auf Verbesserungsmöglichkeiten der Operatoren. Vermutungen aufgrund theoretischer Annahmen können somit durch Experimente evident gemacht oder verworfen werden. Verbesserungen lassen sich dann dort anbringen, wo das Verhalten der Operatoren aus dieser Kombination von theoretischer Vorhersage und experimenteller Überprüfung klarer geworden ist.

Auch die Untersuchung zu Entstehung und Umgang mit Introns steht in unmittelbarem Zusammenhang zur Betrachtung der Theorie der Genetischen Algorithmen und der Funktionsweise des Crossovers.

Nach der genaueren Betrachtung von Crossover in diesem Abschnitt und Introns im folgenden Abschnitt 3.4.6 folgt eine Zusammenfassung von Variationen und Verbesserungsmöglichkeiten von GP in Abschnitt 3.4.7. Diese stichwortartige Zusammenstellung soll als Anregung dienen und Einblick in aktuelle Forschungsfragen geben.

Eine gute Übersicht über alle bisher erwähnten Punkte findet sich in [BNKF98]. Dort finden sich nicht nur die Ansätze von John Koza, einem der Begründer dieses Gebietes, sondern auch Vergleiche mit anderen Arbeiten. Legt man Wert auf detaillierte Beispiele und ausführliche Erklärungen zu Koza's Ansätzen, so sind seine Bücher ([KOZ92, KOZ94, KABK97]) die geeignete Wahl. Einblicke in die verschiedenen Möglichkeiten, GP zu verändern und zu verbessern, geben vor allen Dingen aktuelle Veröffentlichungen aus Tagungen und Zeitschriften. Hinweise finden sich in diesem Text an den entsprechenden Stellen.

3.4.5.1 Übertragung des Schema-Theorems

Trotz seiner Strittigkeit bildet das Schema-Theorem einen guten Ausgangspunkt für eine mathematisch basierte Untersuchung von GP. Die Variabilität der Größe des Genoms und die Möglichkeit des genetischen Materials, beliebig in diesem Genom zu wandern, erschweren allerdings eine theoretische Untersuchung. Es gibt unterschiedliche Ansätze, das Schema-Theorem für GP und insbesondere den Crossover-Operator nutzbar zu machen. Das Ziel ist es, zu verifizieren, ob die von dieser Theorie vorhergesagten Strukturen entstehen und das Verhalten „guter“ Programme begründen. Ein vernünftiges Verhalten wäre demnach in einer Struktur aus kleinen und „fitten“ Building Blocks begründet.

Nach einer informalen Beschreibung von Koza (in [KOZ92]), in der eine Über-

einstimmung der Funktionsweise des Crossover in GP mit den Aussagen des Schema-Theorems argumentativ begründet wird, gab es einige Arbeiten, die diesen Nachweis formal zu führen versuchten. Als Beispiel sei hier die Abschätzung von O'Reilly (aus [oo94, oo95]) genannt:

$$E[m(h, t + 1)] \geq m(h, t) * \frac{f(h)}{f_{\text{mittel}}} * \left(1 - p_c P_d(H, t)\right) \quad (11)$$

mit

$E[m(h, t + 1)]$	Erwartungswert der Anzahl der Schemata zum nächsten Zeitpunkt $t + 1$.
$m(h, t)$	Instanzen eines Schemas h in einer Population zum Zeitpunkt t .
$f(h)$:	Durchschnittliche Fitness aller Instanzen von h zur Zeit t .
f_{mittel} :	Durchschnittliche Fitness der <i>gesamten</i> Population zur Zeit t .
p_c :	Wahrscheinlichkeit eines Crossovers.
P_d :	Maximale Zerstörungswahrscheinlichkeit.

Diese Quantifizierung der Erwartung der Entwicklung der Instanzen $E[m(h, t)]$ bezieht sich nur auf das Crossover und arbeitet mit der maximalen Zerstörungswahrscheinlichkeit (*maximum probability of disruption*) $P_d(H, t)$. Die Merkmale eines Schemas und die Begriffe *Ordnung* und *Länge* wurden auf Baum-GP übertragen. Die Wahrscheinlichkeit des Auseinanderreißen eines Schema-Baumes hängt von dessen Größe und Form ab. Problematisch bei dieser Messung ist die Konservativität der Formel. Durch die Verwendung der *maximalen* Wahrscheinlichkeit P_d sind brauchbare Aussagen über das durchschnittliche Verhalten „guter“ Schemata in Baum-Form kaum zu erwarten. Ausserdem ist P_d von der Größe des Baumes abhängig, variiert also während eines Laufes erheblich und erschwert eine Vergleichbarkeit der Individuen.

Mit ähnlichen Problemen sind auch die anderen Schema-Theoreme für GP behaftet. Insgesamt gesehen ist keines davon in der Lage, die gewünschten Voraussagen über die Existenz und das Verhalten kompakter und „fitter“ Blöcke in den GP-Individuen und ihre Auswirkungen auf die Fitnesswerte zu treffen.

Gedankenexperimente zeigen, dass, unter der Voraussetzung der Existenz von Building Blocks, der Crossover-Operator tendenziell eher eine zerstörerische Wirkung hat. Eine einfache Erklärung hierfür ist die immer größer werdende Wahrscheinlichkeit, dass zusammenhängende Teile des Individuums zertrennt werden, je größer diese sind. Da sich nach Annahme der Building-Block-Hypothese eine Lösung der gestellten Aufgabe aus vielen suboptimalen Lösungen zusammensetzt, steigt mit Zunahme der Qualität des Individuums seine Zerstörungswahrscheinlichkeit. Auch die weiterhin positive Wirkung guter Blöcke in neuen Kontexten – Situationen, wie sie durch Crossover in der Regel gegeben sind – ist nicht immer garantiert.

Das Crossover kann aber auch konstruktive Wirkung haben. Nur mit einem

solchen Operator ist die Zusammensetzung „guter“ Blöcke zu besseren Lösungen überhaupt möglich. Wahrscheinlich handelt es sich um ein dynamisches Gleichgewicht aus diesen beiden Faktoren, die sich im Laufe der Evolution gegenseitig ergänzen. Auch das Argument der vermehrten Reproduktion guter Teillösungen und der damit verbundenen höheren Wahrscheinlichkeit, gute neue Lösungen zu finden, ist noch nicht quantitativ erfasst und gesichert.²⁶ Hier gibt es ebenfalls eine Balance zwischen dem positiven Einfluss der Reproduktion und der zerstörerischen Wirkung des Crossovers. Die Analyse mittels des Schema-Theorems ist somit noch nicht abgeschlossen und liefert für GP keine befriedigende Aussage über die Wahrscheinlichkeit der Evolution besserer Lösungen des Ausgangsproblems.

3.4.5.2 Empirische Ergebnisse

Die Möglichkeit, auf empirischem Weg Aufschluss über diese Wahrscheinlichkeit zu erhalten, brachte ebenfalls nicht die gewünschte Eindeutigkeit in der Antwort. Es ist weder erwiesen, dass GP durch Crossover prinzipiell besser ist als andere Verfahren des Maschinellen Lernens, noch konnte die Aussage bestätigt werden, GP sei nur eine Form eines Hill-Climbing-Verfahrens. Das lässt aber nicht so sehr auf Nachteile dieses Verfahrens schließen, als vielmehr auf die Kontextsensitivität vieler Lern- und Optimierungsaufgaben. Die Abwesenheit allgemeiner Aussagen, die in fast allen Situationen zutreffend sind, bedeutet nicht automatisch die Unbrauchbarkeit des Verfahrens. Sollte sich sogar herausstellen, dass das Crossover lediglich eine Makromutation des Genoms erzeugt, bedeutet das nicht, dass es deswegen nicht für Lernaufgaben zu gebrauchen sei. Die Wichtigkeit des Crossovers für GP ist in jedem Fall gegeben. Nur ist dessen Funktionsweise noch nicht völlig geklärt.

Die Handhabung und das Verständnis der Vorgänge benötigen daher zusätzlich einen Blick in die gegebene Problemstellung. Auch grundsätzliche Verbesserungsmöglichkeiten des Crossovers sind durch das Fehlen endgültiger und allgemeiner Aussagen nicht ausgeschlossen. Daher folgt nach diesem Einblick in den Versuch einer theoretischen Beschreibung ein Blick auf das biologische Vorbild und ein Überblick über einige Verbesserungsmöglichkeiten.

3.4.5.3 Ein Blick in die Biologie

Einige Besonderheiten des Crossover-Mechanismus in der Natur – bei Biologen ist meist der Begriff *Rekombination* geläufiger – macht die Unterschiede zum Crossover in der künstlichen Evolution deutlich:

- *Rekombination bei Lebewesen findet nur innerhalb der gleichen Spezies statt. Lebewesen betreiben sogar einen nicht unerheblichen Aufwand, um die Art des potentiellen Vermehrungspartners herauszubekommen. Durch*

²⁶ Blöcke mit hoher Fitness müssen keine höhere Fitness des Individuums hervorrufen. In Baum-GP kann es z. B. sein, dass der entsprechende Teilbaum gar nicht durchlaufen wird. Oder die Wirkung guter Blöcke wird durch negative Auswirkungen anderer Blöcke aufgehoben.

die Ähnlichkeit beider Genpools bleiben Basisfunktionen nach dem Rekombinationsvorgang erhalten.

In Standard-GP Systemen können Crossover zwischen beliebigen Individuen durchgeführt werden, ohne Rücksicht auf Verwandtschaftsgrade oder Abstammungsverhältnisse. Auch die Funktionalität beider am Crossover beteiligten Programme wird nicht berücksichtigt. So gibt es zum Beispiel keinen Grund dafür zu glauben, dass GP-Individuen der ersten Generation alle zur selben Spezies gehören, da sie zufällig erzeugt sind.

- *Biologischer Crossover beachtet die Semantik der zu tauschenden Chromosomabschnitte. Nur Gene gleicher Funktionalität werden gegeneinander getauscht. Anschaulich ausgedrückt: Das Gen für die Augenfarbe wird nicht gegen das Gen, das die Körpergröße kodiert, getauscht.*

Crossover bei GP ist nicht kontext-sensitiv. Die Funktionalität eines Programmtails im neuen Genom wird nicht beachtet.

- *In der Natur ist eine Rekombination homolog, d. h. Rekombinationen finden nur zwischen gleichen Chromosomen oder solchen mit zumindest sehr ähnlichen Basenpaaren statt.*

Der GP-Crossover sorgt eher für die Zerstörung sinnvoller Teilbereiche, als für deren Erhaltung. Wohingegen die Biologie Wert darauf legt, Gen-Funktionalität zu erhalten.

Insgesamt lässt sich festhalten, dass das Crossover in Standard-GP gänzlich unkontrolliert und ohne die Beachtung von Randbedingungen durchgeführt wird. Im Gegensatz zur Natur, die mit großem Aufwand dafür gesorgt hat, dass das Crossover nicht lediglich eine Makromutation mit großem Unsicherheitsfaktor ist.

3.4.5.4 Verbesserungsmöglichkeiten

In einer besseren Kontrolle des GP-Crossovers liegen Potentiale für eine Verbesserung dieses Operators, von denen einige im Folgenden kurz vorgestellt werden.

Brood Recombination:

Die Vorgehensweise bei einer Population, aus der zwei Individuen eine „Brut“ von N Nachkommen erzeugen (vorgeschlagen von Tackett in [TAC94]):

1. Nimm zwei Individuen für ein Elternpaar aus der Population.
2. Führe das Crossover N mal durch und erzeuge N mal zwei Nachkommen. Dabei bezeichnet N die Brutgröße.
3. Werte die Fitness für alle Individuen aus und sortiere alle Nachkommen in der „Brut“ nach ihrer Fitness. Die beiden besten Individuen sind die Nachkommen der Eltern, alle anderen werden entfernt.

Die Idee für diese Arbeitsweise stammt aus der Beobachtung, dass in der Natur oft eine viel größere Zahl an Nachkommen erzeugt wird, als wahrscheinlich überleben werden. Unterstellt man, dass Individuen mit höheren Fitnesswerten auch bessere Überlebenschancen haben, so kann man diese besseren Individuen zur weiteren Verarbeitung selektieren.²⁷ Der Nachteil dieser GP-Variante liegt in der Vielzahl der durchzuführenden Fitnessauswertungen. Ist dies schon bei Standard-GP-Verfahren ein Flaschenhals, der einen großen Teil der Rechenleistung verbraucht und der Verbesserung bedarf, wird der Rechenbedarf hier sogar noch stark erhöht. Allerdings fällt das bei Tacketts Ansatz kaum ins Gewicht, da er die Fitnesssevaluierung der Nachkommen modifiziert hat. In seinem Algorithmus wird die Fitness nicht mehr über sämtliche Fitness-Cases ermittelt, sondern nur über einen kleinen Teil davon. Es genügt, wenn die Güte der Programme annähernd bestimmt wird.

Tatsächlich konnte Tackett für einige Anwendungen Geschwindigkeitsverbesserungen seines Ansatzes gegenüber Standard-GP zeigen, ohne dass der erhöhte Aufwand für die vermehrten Fitnesssevaluationen zu sehr ins Gewicht fällt. Die schnellere Entstehung sinnvoller Individuen lässt vermuten, dass die Brood Recombination nicht so zerstörerisch auf das Genom wirkt und somit größere Zusammensetzungen von Building Blocks möglich sind. Der Suchraum scheint mit dieser Variante effizienter durchsucht zu werden.²⁸

Zerstörungsvermeidung zusammengehöriger Blöcke:

Eine weitere Möglichkeit, Nachteile des Crossovers auszugleichen, liegt darin, Building Blocks zu erkennen und als zusammenhängende Bereiche zu kennzeichnen. Dies sollte zur Erhaltung gut funktionierender Blöcke und einer Steigerung der Fitness der Individuen führen.

Untersuchungen haben gezeigt, dass sich zwar die Existenz von Building Blocks nicht im strengen Sinne beweisen lässt, aber dass es Codeblöcke gibt, die

- zusammengehörig erscheinen.
- über Eigenschaften verfügen, die sie mit Hilfe von Heuristiken oder Lernalgorithmen als zusammengehörig erkennen lassen.
- eine bessere GP-Performance erzeugen, wenn sie als Blöcke erhalten bleiben.

Dies versuchen nun verschiedene Verfahren, die man als *intelligent* oder *smart* bezeichnet, da sie im Laufe der Evolution *lernen* sollen, welche Teile zusammenhängende Blöcke sind. Das Lernen kann z. B. über zusätzliche Informationen des Crossover-Operators über die Ausführung der Individuen erfolgen²⁹ oder

²⁷ An dieser Stelle hakt die Naturanalogie ein wenig: Das Überleben einer Brut hängt oft auch von anderen Faktoren ab. In diesem Fall liefert die Überlegung aber einen Mechanismus, der schlechte Nachkommen durch Crossover-Fehler wieder ausmerzt.

²⁸ Sicher ist, dass ein größerer Teil des Suchraums durchlaufen wird: In diesem Fall werden $G * (P * N)$ Individuen getestet, statt $G * P$ im Normalfall. (G : Generationen).

²⁹ Die Literaturangaben zu den folgenden Ansätzen finden sich in [BNKF98], Kap. 6.5.2.

durch den Teilbäumen zugeordnete *performance values*, die eine Abschätzung zulassen, ob es sich um einen zusammengehörenden Block handelt.

Eine Möglichkeit, das Erkennen der Building Blocks als *emergenten* Effekt im Laufe der künstlichen Evolution entstehen zu lassen, stellt der Ansatz mit *Explicit Defined Introns* (EDI) von Nordin et. al. dar. Hier werden beim Baum-GP *Intron-Knoten* – Programmbefehle ohne weitere Auswirkungen auf den Ablauf – zwischen die „normalen“ Befehls-Knoten gesetzt. Diese Introns stellen natürliche Zahlen dar und werden in einem Vektor zusammengefasst, der ebenso wie der Programm-Baum, evolviert wird. Der Crossover-Operator wird so angepasst, dass die Wahrscheinlichkeit eines Crossovers proportional ist zu den Werten in den Intron-Knoten. Diese Werte werden so evolviert, dass sie dort am geringsten sind, wo ein Crossover zusammengehörige Teilbäume zerstören würde. Angelehnt an das biologische Vorbild bekommt GP die Möglichkeit, Markierungen auszubilden, die die Gene des Chromosoms begrenzen.

Mit allen diesen Ansätzen konnte die Performance gegenüber Standard-GP in unterschiedlichem Maße verbessert werden.

Homologes Crossover:

Banzhaf et. al. schlagen in [BNKF98] einen Mechanismus vor, der beachtet, dass in der Natur Rekombinationen immer *homolog* sind, also nur zwischen ähnlichen Chromosomen und Genen ähnlicher Funktion stattfinden und nur geringe Veränderungen hervorrufen.³⁰ Ähnlichkeit bedeutet in diesem Zusammenhang sowohl *strukturelle*, als auch *funktionale* Ähnlichkeit. Der vorgeschlagene Crossover-Operator bestimmt zunächst die strukturelle Ähnlichkeit der beiden für das Crossover vorgesehenen Individuen mit Hilfe der *Edit-Distanz*, welche in der Lage ist, Strukturen variabler Länge zu vergleichen. Die Überprüfung funktioneller Ähnlichkeit erfolgt mit dem Vergleich von Fitness-Werten für eine kleine Anzahl der Fitness-Cases.

Der Algorithmus sucht sich mit diesen beiden Ähnlichkeitskriterien je einen Teilbaum der beiden Elternindividuen für das Crossover heraus. Die strukturellen und funktionalen Distanzmaße zwischen den Teilbäumen der beiden Programme werden dann mit einer Wahrscheinlichkeitsfunktion ausgewählt.

Mit diesem informalen Überblick über einige der Verbesserungsmöglichkeiten des Crossoveroperators wird im Folgenden das Problem der Introns diskutiert, und gezeigt, wie sie mit dem Crossover, aber auch mit den anderen Operatoren im Zusammenhang stehen.

3.4.6 Introns genauer betrachtet

Unter Introns versteht man Code, der keine Auswirkung auf den Programmablauf hat, wie z. B.:

³⁰ Ein anderes vor kurzem vorgestelltes Verfahren für homologes Crossover bei linearem GP findet sich in [FCBN99]. Es arbeitet mit festgelegten Stellen, an denen das Crossover in beiden Elterngenomen stattfindet.

$$a = a + 0 \quad (12)$$

$$b = b * 1 \quad (13)$$

Im Gegensatz zu den zur Verbesserung des Crossovers künstlich eingeführten Introns, entstehen solche wirkungslosen Befehle in der Regel von alleine. Je nach Phase des Evolutionslaufes haben Introns sowohl positive, als auch negative Auswirkungen. Positiv ist, dass durch Introns zusammenhängende Programmblöcke vor der Zerstörung durch Crossover geschützt werden können, da die Wahrscheinlichkeit eines Crossover in dem zu schützenden Teilsegment mit zunehmender Codegröße sinkt. Während dies ein Vorteil in den frühen und mittleren Phasen des Laufes ist, kann sich diese Entwicklung nachteilig gegen Ende der Evolution auswirken. Introns vermehren sich stärker – mitunter sogar exponentiell – und es entsteht eine überproportional große Menge Code, sog. *Bloats*, der die Laufzeit der Ausführung des Individuums stark verlangsamen kann.

Daraus ergeben sich Fragen für die Forschung: Wie und warum entstehen Introns? Warum vermehren sie sich exponentiell? Wie kann man ihre Vorteile ausnutzen, ohne ihre Nachteile in Kauf nehmen zu müssen? Interessanterweise gibt es Analogien in der Biologie. Auch hier ist die Rolle der Introns in der natürlichen DNA nicht ganz klar. Nach einem kurzen Blick auf das, was man von den biologischen Introns lernen kann, wird aufgezeigt, wie die obigen Fragen bearbeitet werden können.

3.4.6.1 Introns in der Biologie

Auch in der natürlichen DNA gibt es Basensequenzen, die keine Auswirkungen auf die genetischen Mechanismen haben und keine Information codieren. Im Gegensatz zu den *Exons* werden aus ihnen keine Proteine oder Polypeptide hergestellt. Gründe für die Existenz dieser Sequenzen kann es verschiedene geben. So ist es z. B. möglich, dass es sich um inzwischen wertlos gewordene Informationen handelt, die im Laufe der Evolution, der auch der genetische Code unterliegt, ihren Sinn verloren hat. Es ist nicht automatisch gewährleistet, dass die Teile der DNA, die nicht mehr gebraucht werden, im Laufe der Zeit entfernt werden. Einige Biologen vermuten, dass in diesen Introns ein Potenzial liegt, um bei drastischen Umweltveränderungen flexibel reagieren zu können. Bisher sinnloser Code kann dann als Grundlage neuer Variationen des Lebewesens dienen. Weiterhin können die biologischen Introns Schutz vor fehlgeschlagenen Rekombinationen sein, die zusammengehörende Gene zerstören würden.

Interessanterweise werden Introns während des Prozesses der Expression der DNA in Proteine „überlesen“, sodass sie nicht der Effektivität der Ausführung des genetischen Codes hinderlich sind. Sie kommen nur auf dem Genotyp vor. Introns haben also eine indirekte Auswirkung auf den Organismus, obwohl ihr eigentlicher Code wirkungslos ist.

3.4.6.2 Introns in GP

Definition:

Introns in einem GP-Code haben folgende Eigenschaften:

- Sie entstehen als emergentes Phänomen durch den Prozess der künstlichen Evolution von Strukturen variabler Länge.
- Ein Intron beeinflusst die Überlebensfähigkeit eines Individuums nicht direkt.

Es ist noch einmal zu betonen, dass es sich hierbei nicht um die zu Optimierungszwecken künstlich hinzugefügten Introns handelt.

Die Entstehung von Introns:

Zur quantitativen Beschreibung der Entstehung von Introns werden zunächst einige wichtige Begriffe eingeführt:

- *Effektive Fitness:* Die effektive Fitness berücksichtigt zusätzlich zur eigenen Fitness auch die der Nachkommen des Individuums.
- *Komplexität:* Länge oder Größe eines Programmes oder Blocks in einer für dessen Art natürlichen Maßangabe. So könnte z. B. bei einem Programm in Baumdarstellung die Tiefe des Baumes oder die Anzahl der Knoten gemessen werden.
- *Absolute Komplexität:* Gesamte Größe eines Programmes oder Blocks.
- *Effektive Komplexität:* Länge oder Größe der *aktiven* Teile eines Programmes oder Blockes im Gegensatz zu den Intron-Anteilen.

Die effektive Fitness lässt sich, sowohl für den Crossover-Operator als auch für andere Operatoren, berechnen, indem eine dem Schema-Theorem angelehnte Gleichung umgestellt wird.³¹ Die Differenz zwischen effektiver und aktueller Fitness gibt nun ein Maß dafür an, wie stark der destruktive Einfluss des genetischen Operators ist, unabhängig von der gewählten Fitnessfunktion. Aus diesen Gleichungen lässt sich ebenso die grundlegende Dynamik der Introns in einem GP-Lauf zeigen:

Die aktuelle Fitness wird durch das Verhältnis von effektiver und absoluter Komplexität, ausgedrückt durch einen Quotienten, beeinflusst (stark vereinfacht):

$$Fitness * \frac{effektive\ Komplexität}{absolute\ Komplexität} \quad (14)$$

³¹ Siehe auch [BNKF98], Kap. 7.8–7.10.

Dieser Quotient ist zu Beginn eines Laufes niedrig. Beide Maße unterscheiden sich kaum, d. h. es sind wenige Introns vorhanden. In dieser Situation genügt es, die Fitness des Individuums zu erhöhen, um eine bessere Fitness zu erhalten. Erst durch die allmähliche Zunahme der absoluten Komplexität – verursacht durch Vermehrung der Introns – wird die Steigerung der Fitness immer schwieriger (da der Wert des Quotienten sinkt). Eine Zeitlang ist die Zunahme der Introns sogar positiv, da sie vor den destruktiven Eigenschaften der Operatoren schützt. Aber durch die Formeln kann gezeigt werden, dass das Wachstum der Introns nicht aufzuhalten ist und am Ende sogar exponentiell steigt und den Bloat verursacht.

Schlussfolgerungen:

Aus den theoretischen Überlegungen zu Introns können Schlussfolgerungen gezogen werden, die in einigen der im Folgenden erwähnten Verbesserungen für Standard-GP berücksichtigt werden:

- Veränderungen des Crossover-Operators haben einen Effekt auf die Entstehung der Introns. Je effektiver das Crossover funktioniert, desto weniger Introns werden für den Schutz des Codes gebraucht und desto weniger ist die Gefahr des Bloats am Ende des Laufes.
- Die Sparsamkeit des Codes hat auch Einfluss auf die Introns. Eine Kontrolle der Größe der Individuen und das Verbot der Überschreitung einer Grenze verhindert die exponentielle Ausbreitung von Introns.
- Introns entstehen vermehrt, wenn ein maximaler Fitnesswert erreicht ist. Eine sich im Evolutionsverlauf ändernde Fitnessfunktion könnte diesen Effekt verhindern.

3.4.7 Variationen und Verbesserungen in GP

Variationen und Verbesserungen des Standard-GP Ansatzes sind in vielfältiger Weise denkbar. Die Variationen können sich entweder auf eine der drei üblichen Genome – Baum, Linear, Graph – beziehen oder eine neue Genomart vorschlagen. Beim baum-basierten GP-Ansatz lassen sich vor allen Dingen Veränderungen der genetischen Operatoren Crossover und Mutation denken. Mutationen beispielsweise können ganze Teilbäume löschen oder einzelne Knoten. Kurz vorgestellt werden im Folgenden zwei Variationsmöglichkeiten für linearen GP: Machine Language GP, welches direkt maschinensprachliche Befehle verwendet, und Developmental GP, in welchem eine Unterscheidung zwischen Geno- und Phänotyp eingeführt wird.

Verbesserungen von GP können drei Gruppen zugeordnet werden: Verbesserungen in der Geschwindigkeit, der Evolvierbarkeit und der Suchmächtigkeit der Individuen. Hervorzuheben sind hier die Ansätze zur Parallelisierung mit Hilfe einer Aufteilung in Subpopulationen (auch *demes*) und die Modularisierung von Programmen, hier gezeigt am Beispiel von *Automatic Defined Functions* (ADF).

3.4.7.1 Machine Code GP

Computer sind in der Regel auf ihrer untersten Stufe *Registermaschinen*. Die Operationen verarbeiten die Informationen im Austausch zwischen Speicher und (Prozessor-) Registern. Auf Registerebene zu programmieren ist für den menschlichen Anwender zu detailliert, meistens intransparent und daher fehleranfällig. Trotzdem sind Programme in Maschinensprache kompakt im Speicherplatzverbrauch und effizient in der Ausführungszeit. Dieser Geschwindigkeitsvorteil ist sicher einer der Hauptgründe, GP mit Maschinencode-Anweisungen operieren zu lassen bzw. sogar direkt binären Maschinencode zu benutzen. Darüberhinaus gibt es aber noch einige andere Gründe, die für einen solchen Versuch sprechen. So könnten Computer programmiert werden, für die es noch keine höhersprachlichen Tools gibt. Ausserdem ergibt sich ein weiterer Effizienzvorteil für binären Maschinencode: Es wird keine Interpreter mehr benötigt, dessen Aufgabe gerade die Übersetzung in binäre Maschinenbefehle wäre. Die Geschwindigkeitsverbesserungen durch Maschinen Code GP liegen (laut [BNKF98]) zwischen Faktor 60 und 200.

Grundsätzlich hat man die Wahl zwischen einem normalen GP-Ansatz, der in den Knoten Befehle der jeweiligen Maschinensprache enthält, und der direkten Verwendung binärer Maschinensprache. Ein Problem stellt sich beim Crossover: Es dürfen keine unsinnigen Verknüpfungen vorgenommen werden. Operationen müssen mit den gleichen Datentypen arbeiten und andere Bedingungen der Syntax müssen eingehalten werden. Eine Lösung dazu bietet *Strongly Typed GP* (STGP) [MON93]. STGP arbeitet mit Maschinenbefehlen in den Knoten eines Baum-GP. Zusätzlich gibt es Knoten, die die Ausführungsordnung (beispielsweise Tiefensuche) markieren. Diese darf durch den Crossover nicht zerstört werden.

Alle weiteren Ansätze, die auf Maschinencode basieren, haben diese beiden Bedingungen – Wahl der Repräsentation und Schutz der Syntax vor Crossover – zu berücksichtigen.

3.4.7.2 DGP: Genotyp-Phänotyp-Mapping

In Standard-GP gibt es keine Unterscheidung zwischen dem Suchraum und dem Lösungsraum. Programme, die der Evolutionsprozess aus dem Suchraum der möglichen Programme auswählt, werden direkt als Lösung interpretiert. Mit biologischen Metaphern gesprochen, ist Genotyp gleich Phänotyp. Der entscheidende Nachteil bei dieser Konstellation liegt in dem Zwang, dass evolvierte Programme in jedem Fall syntaktisch korrekt sein müssen. Das bindet den Evolutionsprozess an festgelegte Bedingungen. Weiterhin ist nicht sicher, dass nicht auch in syntaktisch inkorrekten Programmen korrekte Teile enthalten sind. Statistisch gesehen gibt es wesentlich mehr inkorrekte Programme als korrekte. Somit ist es nicht unwahrscheinlich, dass auch hier Potenzial für Lösungen liegt.

Um diesen Nachteil auszugleichen, wurde *Developmental Genetic Programming* (DGP) für lineares GP entwickelt³². Dieser Ansatz stellt eine Abbildung aus dem Genotyp-Raum in den Phänotyp-Raum zur Verfügung. Die Genotypen sind

³² (Erstmals vorgestellt in [BAN94])

einfache Binärsequenzen, die beispielsweise mit Standard-GA Operatoren bearbeitet werden können. Diese Binärsequenzen werden in Teilabschnitte, sog. *Codons*, aufgeteilt, und den Symbolen des Phänotyp-Raums zugewiesen. So können z. B. die Sequenzen 00, 01 und 10 für die Symbole a , b und $+$ stehen. Die Aufgabe von DGP ist es nun, anhand einer Grammatik³³ festzustellen, ob die richtigen Symbolsequenzen benutzt wurden und in welcher Reihenfolge. Bei unkorrektem Gebrauch wird ein korrektes Symbol, das der falschen Sequenz am nächsten liegt, gesucht.

Ein weiterer Vorteil dieser Methode wird an den verschiedenen grossen Dimensionen des Such- und des Lösungsraumes deutlich. Der Suchraum ist höher dimensioniert, birgt also auch mehr potenzielle Lösungen, als der Lösungsraum.

3.4.7.3 Parallelisierung: Demes

Da Evolutionäre Algorithmen im Allgemeinen populationsbasiert sind, liegt eine Geschwindigkeitsverbesserung durch Aufteilung in Subpopulationen, sog. *Demes*, nahe.³⁴ Diese können nun auf parallele Prozessorknoten in Parallelrechnern aufgeteilt und getrennt voneinander evolviert werden. Damit die Entwicklungen nicht völlig isoliert voneinander ablaufen, findet nach jedem Evolutionschritt eine Migration von Individuen zwischen den Subpopulationen statt. Dieser Austausch soll zum einen einzelne Demes vor Stagnationen in lokalen Minima schützen, darf aber andererseits die Diversitätserhaltung in der Gesamtpopulation nicht gefährden.

Die einzelnen Ansätze zur Parallelisierung unterscheiden sich hauptsächlich darin, wie diese Subpopulationen untereinander vernetzt sind und wie sie einen Teil der Individuen austauschen. So schlagen Andre und Koza in [KABK97] eine vollständig verknüpfte quadratische Matrix vor, wogegen Brameier und Banzhaf³⁵ eine Ringstruktur bevorzugen, in der jeder Knoten genau einen Vorgänger und einen Nachfolger hat. Durch den geringeren Grad der Vernetzung soll eine eigenständigere Entwicklung der Demes und eine höhere Vielfalt der Individuen erreicht werden.

3.4.7.4 Modularisierung: ADFs

Eine Möglichkeit zur Modularisierung von GP sind die von Koza in [KOZ94] vorgeschlagenen *Automatic Defined Functions* (ADFs) für Baum-GP. Dazu wird ein Baum in zwei Äste eingeteilt: Den *Result-Producing Branch* (enthält die eigentliche Funktionalität und wird während der Fitnesssevaluation ausgewertet) und den *Function-Defining Branch* (enthält die Funktionsdeklaration und -definition). Im Deklarationsteil des Baumes wird, wie bei Funktionen in C oder Pascal, ein Funktionsname, eine Argumentliste und ein Auswertungsteil festgelegt. Der Result-Producing Branch kann nun den Namen der deklarierten Funktion aufrufen und beliebig verwenden.

³³ DGP verwendet eine LALR(1) Grammatik

³⁴ Zu verteilten GAS siehe [TAN89]

³⁵ Siehe [BB99], aufbauend auf dem *Stepping Stone Model* von Kimura.

Ein Individuum kann beliebig viele ADFs enthalten. Damit die Evolution funktioniert, muss der Crossover-Operator angepasst werden. Dieser entscheidet zu Beginn, in welchem der beiden Zweige ein Crossover durchgeführt wird. Rekombination findet somit nur zwischen Zweigen gleichen Typs statt. Damit ist es auch möglich, modularisierte Programme zu evolvieren.

3.5 Schach und Maschinelles Lernen

Im Jahr 1986 hat S. Skiena im ICCA Journal einen Überblick über damalige Anwendungen des maschinellen Lernens auf dem Gebiet des Computerschachs veröffentlicht [SKI86], die ihn eher pessimistisch stimmten. Zehn Jahre später nimmt Johannes Fürnkranz diesen Artikel zum Anlass, um die Fortschritte, die seit dieser Zeit gemacht worden sind, in [FÜR96] zusammenzufassen.

Das erste Gebiet, das Fürnkranz betrachtet, ist die Klassifizierung von Endspielen. Dabei werden Positionen von einem bestimmten Schachendspiel in die Kategorien *gewonnen* und *nicht gewonnen* eingeteilt. Ein induktiver Lernalgorithmus benutzt diese Positionen, um aus ihnen allgemeine Regeln zur Klassifizierung von Endspielen abzuleiten. Oft wird nur eine Teilmenge der Positionen – die *Trainingsmenge* – dazu benutzt, die Regeln zu folgern. Die restlichen Positionen – die *Testmenge* – werden dann mittels der gelernten Regeln klassifiziert und das Ergebnis mit der korrekten Klassifizierung verglichen. Dadurch erhält man ein Qualitätsmerkmal für die erzeugten Regeln und somit für den Lernalgorithmus.

Als Beispiele für diese Art maschinellen Lernens nennt Fürnkranz u. a. die Experimente von Quinlan zum Lernen von Regeln für das Endspiel *König und Turm gegen König und Springer* [QUI83], bei denen der dort benutzte ID3-Algorithmus mit Hilfe von weniger als 10% der möglichen Positionen einen Entscheidungsbaum erstellen konnte, der nur 2 Fehler bei einer Testmenge von 10 000 zufällig ausgewählten Positionen machte. Allerdings war dieses Ergebnis nur durch die richtige Wahl der Attribute möglich, und tatsächlich ist das größte Problem bei der Klassifizierung von Endspielen, eine geeignete Repräsentation für die Schachpositionen zu finden.

Der DUCE-Algorithmus von Muggleton [MUG90] kann dem Benutzer selbstständig Konzepte auf hoher Ebene vorschlagen. DUCE sucht nach allgemeinen Mustern in der Regelsammlung (die anfangs nur aus einer Menge von Regeln besteht, die alle jeweils eine spezielle Brettposition beschreiben) und versucht, deren Größe zu verkleinern, indem es gefundene Muster durch neue ersetzt.

Eine weitere angewandte Technik ist die des erklärungsbasierten Lernens (*Explanation-Based Learning*, EBL), dessen Grundidee ist, eine Erläuterung für ein gegebenes Beispiel zu finden, die dann verallgemeinert werden kann. Die verallgemeinerte Erläuterung beinhaltet dann eine Regel, mit der ähnliche Beispiele klassifiziert werden können. Als Beispiel ist vor allem Tadepallis Weiterentwicklung, das *lockere erklärungsbasierte Lernen*, das in Kapitel 3.5.2 ausführlicher besprochen wird, zu nennen.

Beim fallbasierten Urteilen (*Case-Based Reasoning*, CBR) wird ein neues Problem dadurch zu lösen versucht, indem das CBR-System sich an ähnliche, vorher

gelöste Probleme „erinnert“ und die damaligen Lösungen der neuen Situation anpasst. Durch das Speichern der gefundenen Lösungen für den zukünftigen Gebrauch verbessert das System ständig seine Problemlösungsmöglichkeiten, d. h. es „lernt“.

Die Idee des Lernens einer Bewertungsfunktion, bzw. der Verbesserung einer Bewertungsfunktion durch einen Lernprozess ist schon relativ alt: Seit dem berühmten Dame-Programm von Samuel([SAM67]) aus dem Jahre 1959 ist die Idee, dass man Spiele von Experten oder Spiele des Programms gegen menschliche Gegner oder sich selbst dazu nutzt, um die Gewichtung von Parametern einer Bewertungsfunktion zu optimieren, weit verbreitet. Z. B. hat Tunstall-Pedoe in [TP91] versucht, die Gewichtungen mittels Genetischer Algorithmen zu optimieren.³⁶ Der Algorithmus verwaltet eine Menge von Gewichtsvektoren, die *Generation* genannt werden. Die nächste Generation wird dadurch berechnet, dass zufällig einige Gewichte eines Mitgliedes der vorhergehenden Generation geändert werden (*Mutation*) bzw. dass einige Gewichte zwischen zwei Mitgliedern getauscht werden (*Kreuzung*). Die Wahrscheinlichkeit, dass Mitglieder einer Generation für diese Operationen ausgewählt (*selektiert*) werden (und dass sie in der nächsten Generation überleben werden) ist proportional zu ihrer *Fitness*. Die Fitness einer bestimmten Parametermenge wurde mit der Prozentzahl der Testpositionen, für die die Bewertungsfunktion bei einer zufälligen Auswahl von 500 Positionen aus 389 Großmeisterspielen denselben Zug wählt wie der Großmeister, veranschlagt. Das Programm entwickelte eine Menge von Gewichten, die sich nicht schlechter verhielten, als diejenigen, die der Autor des Programmes manuell festsetzte, aber die trotzdem wenig Übereinstimmung mit letzteren zeigten. Der Nachteil, der für die Vermeidung lokaler Optima in Kauf genommen werden muss, ist die Ineffizienz der genetischen Algorithmen. Es gibt daher verschiedene Versuche mit neuronalen Netzwerken, Bewertungsfunktionen zu optimieren. Z. B. hat van Tiggelen bei seinen Experimenten herausgefunden, dass die Optimierung mit neuronalen Netzen – in seinem Testgebiet – nicht nur effizienter ist, sondern auch zu besseren Resultaten als die Benutzung genetischer Algorithmen führt [TIG91]. Thruns „NeuroChess“ [THR95] – das in Kapitel 3.5.4 ausführlich besprochen wird – gehört ebenso in diese Kategorie wie Tesausos bekanntes Backgammon-Programm [TES92].

3.5.1 Methoden „roher Gewalt“

Die Methoden, die mit „roher Gewalt“ arbeiten, beruhen darauf, dass sie eine Vorschau auf die möglichen zukünftigen Spielentwicklungen berechnen und diese zu einer geschickten Auswahl von Zügen benutzen. Sie betrachten also den Entscheidungsbaum, der aus allen möglichen, d. h. regelgerechten, Zügen besteht, und versuchen, einen Pfad zu finden, der unabhängig von den Zügen des Gegenspielers zu einem Sieg führt. Bei den meisten (interessanten) Spielen, wie

³⁶ Dieser Ansatz unterscheidet sich von dem dieser Projektgruppe vor allem dadurch, dass hier nur Parameter für eine gegebene Bewertungsfunktion optimiert werden. Unser Ziel dagegen ist es, einen Schach-Algorithmus suchen, der Bewertungsfunktionen als Bausteine nutzt.

z. B. Schach, ist der Entscheidungsbaum allerdings so immens groß, dass man nicht den ganzen Entscheidungsbaum durchsuchen kann, sondern nach einer gewissen Suchtiefe abbrechen muss. Deshalb muss man sich mit Zwischenzielen zufrieden geben. Diese Zwischenziele bestehen darin, Spielsituationen zu erreichen, die für einen selbst vielversprechender sind, als für den Gegner. Um die erreichbaren Spielsituationen, also die Knoten des Entscheidungsbaumes, miteinander vergleichen zu können, müssen diese daher eine Bewertung bekommen. Es gibt verschiedene Bewertungsverfahren für Spielbäume³⁷; das am häufigsten verwendete ist das *Minimax-Verfahren*: Die Hauptidee besteht darin, dass der Wert eines Knotens als Maximum der Minima der Nachfolgeknoten berechnet wird. Das Minimax-Verfahren ist eine pessimistische Entscheidungsregel, die auf den ungünstigsten Fall eingestellt ist [SCH97]. Ein ähnliches Verfahren ist das *Negamax-Verfahren*³⁸, bei dem ein Knoten das Maximum der negierten Werte der Nachfolgeknoten als Bewertung zugewiesen bekommt [KÜM92]. Der Rechenaufwand für dieses Verfahren kann deutlich reduziert werden, wenn z. B. mit Hilfe des *Alpha-Beta-Verfahrens* (*alpha-beta-pruning*) nur Knoten betrachtet werden, die auch wirklich einen Einfluss auf die Bewertung darüberliegender Knoten ausüben.³⁹

Die meisten heutigen Schachprogramme arbeiten mit diesen Methoden „roher Gewalt“, die auch beträchtliche Erfolge vorzuweisen haben: So besiegte der Schachcomputer *Deep Thought* 1988 den Großmeister Bent Larsen und das Nachfolgemodell *Deep Blue* gewann 1996 eine Turnierpartie gegen den amtierenden Weltmeister Garri Kasparow.

Für die Künstliche Intelligenz sind diese Erfolge aber beinahe wertlos, da sie nur auf purer Rechenpower und eben „roher Gewalt“ anstatt auf „intelligenten Verfahren“ basieren.

3.5.2 Durch approximiertes Wissen ermöglichte Planung in Spielen

In Systemen, die erklärungsbasiertes Lernen verwenden, führt das Lernen von vollständigem und korrektem Wissen zu der Produktion von vollständigen und korrekten Erklärungen für die Zielvorstellung, welche aber oft zu komplex und damit schwer zu handhaben sind. Daher werden im Erklärungsprozess oft Vereinfachungen und Approximationen benutzt, die aber Ungenauigkeiten und Fehler sowohl im gelernten Wissen als auch in den erzeugten Plänen mit sich bringen. Ein wissensbasierter Planer sollte deshalb mit nicht-perfektem Wissen klarkommen und darf daher auch einige Fehler machen, diese Fehler sollten sich aber mit verbessertem Wissen verringern. Um das Lernen überhaupt rechtfertigen zu können, sollte natürlich die Planung mit Wissen schneller sein, als die ohne Wissen. Außerdem sollten die Pläne des Gegners berücksichtigt werden.

In [TAD89] verwendet Prasad Tadepalli die Methode des *lockeren erklärungs-basierten Lernens* (*Lazy Explanation-Based Learning*, LEBL), bei der aufgrund von gelerntem Wissen verschiedene Züge für beide Spieler untersucht werden und

³⁷ Diese werden ausführlicher in 3.2 besprochen.

³⁸ Oft wird es auch als *Negamax-Verfahren* bezeichnet.

³⁹ Eine ausführlichere Erklärung der hier erwähnten Verfahren finden sich auch in [KAI90].

eine Menge von Makros extrahiert wird. Diese Makros, die er *optimistische Pläne* bzw. *O-Pläne* nennt, bestehen aus einem Ziel und einer Abfolge von Zügen, wobei jeder Zug von der schwächsten Bedingung angeführt wird, die erfüllt sein muss, damit der Rest der Zugfolge anwendbar ist. O-Pläne sind in dem Sinne optimistisch, dass angenommen wird, dass die gegnerischen Züge entweder den Erfolg des O-Plans nicht beeinflussen, oder aber selbst Teil des O-Plans sind.

Bei der durch Wissen ermöglichten Planung (*Knowledge Enabled Planning*, KEP) wird die Suche nach Lösungen auf die Pfade des Entscheidungsbaumes beschränkt, die das verfügbare Wissen – das in Form von Makros oder O-Plänen vorliegt – vorschlägt. Um mehr als einen möglichen gegnerischen Zug in Erwägung zu ziehen, werden O-Pläne zu komplizierteren Suchbäumen, *C-Pläne* genannt, kombiniert.

LEBL erzeugt vielversprechende C-Pläne für jeden Spieler und testet sie gegeneinander, indem es den Entscheidungsbaum ausdehnt und neu bewertet. Dieser Prozess endet, wenn ein C-Plan eines Spielers nicht mehr von einem gegnerischen C-Plan mit begrenzter Komplexität geschlagen werden kann.

Die anfangs gestellten Forderungen an einen wissensbasierten Planer werden durch LEBL folgendermaßen erfüllt: Die Tatsache, dass die O-Pläne nicht perfekt sind, wird dadurch berücksichtigt, dass sie kombiniert und auf Wechselwirkungen getestet werden. Je mehr Wissen LEBL gelernt hat, desto weniger Fehler macht es, indem es erschöpfender sucht. Mit den nötigen O-Plänen sucht LEBL mindestens eine Größenordnung weniger Knoten als ein auf Alpha-Beta-Suche basierendes Programm, und berücksichtigt zusätzlich die Anwesenheit eines aktiven Gegenspielers.

3.5.3 Das musterbasierte Schachprogramm von Levinson

Eine der menschlichen Wahrnehmung des Schachspieles nachempfundene Methode benutzt Robert Arlen Levinson für sein selbstlernendes, musterbasiertes Schachprogramm [LEV89].

Levinson setzt sich dafür das Ziel, ein Schachprogramm zu entwickeln,

- das mehr auf „zusammenhängenden Urteilen“ (associative-reasoning) basiert, als auf Suche. (Wobei die Möglichkeit der Suche allerdings nicht ganz unbeachtet bleibt.)
- das wichtige zusammenhängende Muster alleine entdeckt und deren Wichtigkeit erlernt.
- das selbstlernend ist, d. h. es lernt, indem es selbst Schach spielt.

Das schachspezifische Wissen des Systems ist auf die Spielregeln, eine Graph-Repräsentation für Brettpositionen und Muster sowie eine Methode für die Erstellung dieser Muster, beschränkt.

Jede Schachposition wird durch einen gerichteten bewerteten Graph⁴⁰ dargestellt, wobei die Knoten sowohl die Spielsteine, als auch unbesetzte Felder um

40 Für diese Graphen wird ebenfalls die Bezeichnung *Muster* benutzt.

die Könige herum, darstellen. Die Kanten repräsentieren Angriffs- und Verteidigungsbeziehungen zwischen den Steinen. Die Graphen werden in einer halbgeordneten Hierarchie mit der *Teilgraph von*-Relation gespeichert. *Teilgraph von* ist hierbei gleichbedeutend mit *allgemeiner als*. An dem einen Ende der Hierarchie sind daher einfache, allgemeine Muster wie z. B. *weißer Läufer kann schwarzen Bauer schlagen* und an dem anderen Ende sind komplizierte, spezielle Muster, die ganze Stellungen beschreiben. Die Graphen werden vom Programm selbst erzeugt und verwaltet. Für jeden Graph wird seine Struktur, Zeiger auf seine direkten Vorgänger und Nachfolger in der Halbordnung und ein Gewicht zwischen 0 und 1, das die Stärke des Musters bewertet, gespeichert. (Dabei steht 1 für Sieg, 0,5 für Remis und 0 für Niederlage.) Eine Position wird als arithmetisches Mittel der Gewichte der direkten Vorgänger ihres Graphen – bzw. als ihr eigenes Gewicht, falls ihr Graph schon existiert – bewertet. Für eine gegebene Position wählt das Programm einen legalen Zug, der den Gegner in die Position mit der kleinsten Bewertung zwingt.

Das Lernmodell ist eine Kombination aus *Zeitunterschied-Methoden* (TD) mit einem dynamischen booleschen Merkmal (Erzeugen/Löschen), wobei letzteres die An- oder Abwesenheit eines gegebenen Musters in der aktuellen Position anzeigt. Das TD-Lernen wird für die Stellungsbewertung benutzt. Die Idee dabei ist, die Gewichte jeweils so zu aktualisieren, dass die Bewertung einer Position mehr mit der Bewertung der folgenden Position übereinstimmt.

Wenn ein neues Muster gespeichert wird, sucht das Programm zuerst in den bereits vorhandenen Mustern das dazu ähnlichste Muster und fügt einen maximalen, gemeinsamen Teilgraph als neues Muster ein. Diese Prozedur wird dabei rekursiv wiederholt. Zum Schluß wird dem Muster ein Gewicht, das dem Durchschnittsgewicht seiner Eltern entspricht, zugewiesen.

Eine gut gewählte Menge einer kleinen Anzahl von Mustern könnte einen ganzen Entscheidungsbaum ersetzen. Es werden Techniken entwickelt, um aus einem Entscheidungsbaum eine minimale Anzahl von Mustern zu erzeugen, die das optimale Verhalten, das am Entscheidungsbaum abzulesen ist, nachahmen. (Eine solche Menge von Mustern muß existieren, da die Menge schlimmstenfalls aus den Positionen selbst bestehen würde.)

Bei der praktischen Ausführung begann das Programm mit fast zufälligen Zügen und hatte wenig Schwierigkeiten Schach-Prinzipien wie *Schlagen (oder Angreifen) eines höherwertigen Steines*, *Schlagen (oder Angreifen) eines ungedeckten Steines*, *Decken ungeschützter Steine* usw. zu lernen. Insgesamt ist die Spielstärke allerdings noch verbesserungsbedürftig, es besteht jedoch die Hoffnung, dass das Programm in Zukunft einen Punkt erreicht, an dem es Spieler zu schlagen vermag, die es zur Zeit noch mühelos besiegen.

3.5.4 Thruns NeuroChess

Das Programm *NeuroChess* [THR95] von Sebastian Thrun lernt das Schachspiel durch die Betrachtung der Endergebnisse von Partien. Der zentrale Lernmechanismus ist ein erklärungsbasiertes neuronales Netzwerk (*Explanation-Based Neural Network*). Wie auch Tesauros Backgammon-Programm ([TES92]) kon-

struiert NeuroChess eine Bewertungsfunktion mit Hilfe von *Zeitunterschied-Methoden* (TD).

TD-Methoden werden oft eingesetzt, um Funktionen zu lernen, die den Ausgang von Partien voraussagen, und daher als Stellungsbewertung geeignet sind. Die Aufgabe von TD(0), der Variante von TD, die hier benutzt wird, ist, eine Bewertungsfunktion V zu finden, die Schachstellungen bezüglich ihrer Güte anordnet. Diese Bewertungsfunktion wird in NeuroChess durch ein neuronales Netzwerk repräsentiert.

Da in so komplexen Gebieten wie Schach rein induktive Lerntechniken wegen der enormen Trainingszeiten praktisch nicht anwendbar sind, verwendet Thrun erklärungs-basierte Methoden, die mit weniger Trainingsdaten Spielsituationen genauer verallgemeinern können. Das erklärungs-basierte neuronale Netzwerk benutzt Wissen über das Schachspiel, welches durch ein gesondertes neuronales Netzwerk – dem *Schachmodell* M – repräsentiert wird. Dieses besteht aus 175 Eingabeeinheiten, 165 verborgenen Einheiten und 175 Ausgabeeinheiten und wird mittels einer Datenbank, die 120 000 von Großmeistern gespielte Schachpartien enthält, trainiert. Danach lernt NeuroChess eine Bewertungsfunktion V , die durch ein neuronales Netzwerk mit 175 Eingabeeinheiten, 0 bis 80 verborgenen Einheiten und einer Ausgabeeinheit, repräsentiert wird. Um die Spielstärke von NeuroChess abschätzen zu können und somit die durch das Lernen erzielten Fortschritte zu messen, spielt es in gleichmäßigen Zeitabständen gegen das Programm *GNU-Chess*, aus dem auch der in NeuroChess verwendete Suchmechanismus übernommen wurde.

Als Ergebnis bleibt festzuhalten, dass NeuroChess meistens sehr schlechte Eröffnungen spielt, die oft für ein Remis oder eine Niederlage verantwortlich sind. Da TD die Werte vom Ende des Spiels bis zum Anfang verbreitet, sind die schlechten Eröffnungen allerdings auch nicht sehr verwunderlich. Obwohl NeuroChess es einige hundert Male gelungen ist, *GNU-Chess* zu schlagen, ist es insgesamt gesehen doch noch ziemlich schlecht, verglichen mit *GNU-Chess* oder menschlichen Schachspielern.

NeuroChess steht noch zwei grundlegenden Problemen gegenüber: Als erstes ist dieses die beschränkte Trainingszeit, wobei erwartet wird, dass eine hervorragende Beherrschung des Schachspiels auch übermäßig viel Trainingszeit voraussetzt. Das zweite Problem ist der Trade-off zwischen Wissen und Suche, d. h. die Zeit, die in die Suche investiert wird, steht nicht mehr für die Bewertung von Schachstellungen zur Verfügung, und umgekehrt.

3.6 Bewertung von Spielstärke im Schach

3.6.1 Bewertbarkeit von Spielstärke

Offensichtlich gibt es verschiedene Heransgehensweisen an die Bewertung von „Güte von Schachspiel“:

- Anhand der Züge:

Es erscheint sehr schwer, eine objektive Qualität von Schachzügen festzustellen. Man müsste ein Orakel, etwa den Schachweltmeister befragen. Das

Verstehen des Schachspiels unterliegt überdies noch einer steten Evolution, und Schachweltmeister zu werden, hat nur derjenige eine Chance, der eine Lücke im gängigen Schachverständnis auftut, nach der eigentlich gute Stellungen schlecht bewertet werden. Allerdings Spielstärke auf sehr niedrigem Niveau ließe sich derart bewerten. Je näher sich das Schachkönnen des Prüflings dem des Weltmeister-Orakels näherte, desto unsicherer fiel letzteren Urteil.

- Anhand der Ergebnisse

Nach dem Ergebnis zu verfahren, ist in vielen Sportarten verbreitet und recht naheliegend. Im Schach jedoch verfiel man erst spät darauf. Es tat sich immer wieder die Frage auf, was Schach denn nun eigentlich sei: Ein Spiel? Ein Sport? Eine Kunst? Eine Wissenschaft? In der heutigen Zeit gilt Schach mehr denn je als Sport, und darum sollte es legitim sein, eine Bewertungsmaxime aus der Sportwelt anzubringen. Aber was für Ergebnisse werden gewertet? Turniere können gewertet werden (wie auch im Tennis), aber es gibt eine Vielzahl verschiedener Turnierformen: Rundenturniere (jeder gegen jeden), KO-Turniere, und Schweizer-System-Turniere. Außerdem gibt es über mehrere Partien angelegte Wettkämpfe zwischen genau zwei Spielern. Die Auffassung, die Güte von Schach sei feststellbar anhand der Ergebnisse, hat sich weitgehend durchgesetzt, obwohl ästhetische Aspekte dabei nicht berücksichtigt werden.

- Anhand von Sekundärmerkmalen der Partien:

Als beispielhafte Sekundärmerkmalswertung sei das Torverhältnis im Fußball genannt. Mögliche Kriterien im Schach könnten beispielsweise sein:

- Die Länge der Partie in Zügen
- Die Materialsituation zum Zeitpunkt des Matts. Für eine ausgeglichene Materialsituation sollte der Mattsetzende einen Bonus bekommen, weil im Fall von Beraubungssiegen das Matt irgendwann zufällig eintreten muss.

Ein Spieler, dessen Maxime lautete, langsam aber sicher vorzugehen, würde bei derartigen Bewertungsregeln benachteiligt. Weiterhin gelten an dieser Bewertung auch die gleichen Kritikpunkte wie an der Wertung anhand der Qualität der Züge.

Da es auf anderen Partieattributen als dem Resultat beruhenden Bewertungen also offensichtlich an Objektivität mangelte, verwundert es nicht, dass die weltweite Schachgemeinde seit Jahrzehnten die Spielstärkebewertung anhand der Turnierergebnisse präferiert. Das weitestverbreitete solche Bewertungsverfahren, das *Elo-System*, soll im nächsten Abschnitt vorgestellt werden.

3.6.2 Die Elo-Bewertung

Die Elo-Bewertung verwendet als Eingangsgrößen ausschließlich die Resultate gespielter Schachpartien. Ihre Namensgebung geht zurück auf den ungarischen

Mathematiker Arpad Elo. Seine Arbeit „The ratings of chess players past and present“ ([ELO78]) erschien 1978, aber schon seit 1960 wird das darin vorgestellte System von der United States Chess Federation und seit 1970 von der Federation Internationale Des Echecs (FIDE) angewandt. Das Verfahren fand weiten Zuspruch und wird mittlerweile auch in Zusammenhang mit Go, Backgammon, Scrabble, Fußball (US College Soccer), Tischtennis, Squash und neuerdings diversen Internetspielereien benutzt. Eine hilfreiche Einführung in die praktischen wie mathematischen Grundlagen des Elo-Systems findet der Leser in [BLE98].

In seinem Kern ist das Elo-Bewertungsverfahren ein „Paired Comparison“-Verfahren aus der quantitativen Psychologie. „Paired Comparison“ in die Sprache der Mathematik überführt heißt nichts anderes als „vollständige Ordnung“, wo alle Elemente paarweise vergleichbar sind.

Auf der Menge der Schachspieler soll eine irreflexible Ordnung hergestellt werden. Essentielle Eigenschaft einer solchen Ordnung ist die Transitivität, die in realen Beobachtungen aber regelmäßig verletzt wird. Irreflexivität impliziert fernerhin Antisymmetrie, aber im Schach gibt es Remisen. Beide Probleme werden dadurch behoben, dass man nicht mit einzelnen Spielausgängen arbeitet, sondern mit Erwartungswerten von Spielausgängen.

Die gewählte Bewertung der Schachspieler soll aber nicht nur eine Aussage der Form „Spieler A ist besser als Spieler B“ machen, sondern sie quantifizieren „Spieler A ist dreimal so gut wie Spieler B“.

Ein Modell, dass sich als Grundgerüst für ein geeignetes Bewertungssystem eignet, ist das „Thurstone Case V“-Modell ([THU94]), welches auf der Standardnormalverteilung beruht.

$$\Phi(v_i - v_j) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{(v_i - v_j)} e^{-\frac{1}{2}t^2} dt$$

$v_i - v_j$ bezeichnet dabei den Erwartungswert eines Gewinns von Spielers i über Spieler j .

Insbesondere lässt sich im Elo-System die Gewinnerwartung einer Spielpaarung aus der Differenz der Elo-Zahlen berechnen.

Eine praktische Implementierung eines Paired-Comparison-Verfahrens muß allerdings weitere Gesichtspunkte berücksichtigen: Bislang wurde von idealen, in ihrem Spielverhalten unveränderlichen Spielern ausgegangen. In der Realität hingegen findet man Phänomene wie Tagesform, Angstgegner und Alterung.

Vor allem des letzteren Punktes (und auch russischer Wunderkinder) wegen ist eine Update-Funktionalität auf der Basis weniger, unlängst gespielter Partien wichtig.

Diese aktuellen Partien sind es, die das Performance-Rating R_p beschreibt:

$$R_p = R_c + D(p)$$

Dabei beschreibt R_c die Durchschnittsbewertung der Gegner und der Summand $D(p)$ das prozentuale Spielergebnis aus den gespielten Partien.

Elo difference	Expected score	Elo difference	Expected score
0	0.50	140	0.76
20	0.53	160	0.79
40	0.58	180	0.82
60	0.62	200	0.84
80	0.66	300	0.93
100	0.69	400	0.97
120	0.73

Tabelle 2: Gewinnwahrscheinlichkeit in Abhängigkeit der Elo-Differenz

Neu in die Menge der zu Bewertenden eintretende Spieler werden als Spezialfall direkt mit ihrem Performance-Rating bewertet.

Für alle anderen Spieler erfolgt ein Update auf ihre bisherige Bewertung. Die Lernrate wähle man dabei mit Bedacht.

Im Elo-System wird es als „beliebig“ betrachtet, wie oft ein Update erfolgt. Die Konkatenation dieser Aktualisierungen ist nicht kommutativ, aber die Abweichungen werden als marginal hingenommen. In der Praxis wird das Update am Ende eines Turniers, das sich beispielsweise über sieben Spiele erstrecken kann, vorgenommen.

In Tabelle 2 wird die Gewinnwahrscheinlichkeit eines Spielers A über Spieler B in Abhängigkeit von der Elo-Differenz aufgezeigt. Diese Werte sollen lediglich einen Eindruck geben und können auch mithilfe der obigen Formel von Thurstone berechnet werden. Der Leser möge jedoch im Gedächtnis behalten, dass eine Elo-Differenz von 400 Punkten mit einer Gewinnwahrscheinlichkeit von 97% für den besserbewerteten Spieler einem Klassenunterschied gleichkommt. Garry Kasparov als weltbesten Spieler hält derzeit (1.7.2000) eine Elo-Zahl von 2849.

Auf der Basis der Gewinnerwartung wird die neue Elo-Bewertung berechnet:

$$R_n = R_o + C * (S - S_e)$$

wobei:

R_n = neue Elo-Bewertung

R_o = alte Elo-Bewertung

S = Punktausbeute

S_e = erwartete Punktausbeute

C = Konstante

C bezeichnet die maximale Wertungsänderung pro ausgetragener Partie und beeinflusst damit wesentlich die Dynamik des Systems. Im Schach pflegt man diesen Wert auf 30 zu setzen.

4 Entwicklungsumgebung und Werkzeuge

In den folgenden Abschnitten werden die Entwicklungsumgebung und Werkzeuge beschrieben, die wir für die Realisierung des Projektes benutzt haben. Da eine Einteilung oft nicht eindeutig möglich ist, wird nicht zwischen Entwicklungsumgebung und Werkzeug unterschieden. Es wird beschrieben, warum wir welche Soft- und Hardware eingesetzt haben; teilweise wird deren Funktionsweise beschrieben. Wie die Ziele unseres Projektes mit diesen Werkzeugen realisiert werden, wird im Kapitel 6 beschrieben.

4.1 Rechner

Uns steht für das Projekt ein Rechnerpool bestückt mit Sun Ultra 1 und Sun Ultra 10 Rechnern zur Verfügung. Als Betriebssystem ist Solaris 2.6 vorhanden. Die Softwareumgebung ist auf allen Rechnern identisch, so dass es einfach möglich ist, Teile des Projektes auf mehreren Rechnern gleichzeitig zu starten und parallel ablaufen zu lassen. Neben privaten Verzeichnissen für die Mitglieder der Projektgruppe existiert ein allgemein zugängliches Verzeichnis für gemeinsam benutzte Daten.

4.2 SYSGP

4.2.1 Einführung

SYSGP ist eine in C++ implementierte Klassenbibliothek, welche zur Umsetzung von Methoden der Genetischen Programmierung in C++ genutzt werden kann. SYSGP wurde am Lehrstuhl 11 des Fachbereichs Informatik der Universität Dortmund entwickelt. Der folgenden Beschreibung liegt die Version 3.0 dieser Klassenbibliothek zugrunde.

Die wichtigsten Bestandteile von SYSGP seien hier einführend vorgestellt. SYSGP stellt Klassen zur Verfügung, mit deren Hilfe die Individuen und deren Programme (sowohl lineare als auch Bäume und Graphen) modelliert werden können. Die einer Evolution unterzogenen Individuen können in einem den Pool verkörpernden Objekt zusammengefasst werden. Es existieren darüber hinaus Klassen, welche die Umgebung (Register, Speicher, Stack) simulieren, in der die Programme der Individuen ausgeführt werden können. Die Ausführung der Programme wird durch Interpreterobjekte vorgenommen. SYSGP beinhaltet Interpreterklassen für jeden der drei Programmtypen. Daneben stellt SYSGP Methoden zur Verfügung, welche die Genetischen Operatoren (Crossover, Mutation) realisieren, sowie weitere zur Durchführung der Selektion von Individuen aus dem Pool. Eine tiefergehende Darstellung der wichtigsten in SYSGP enthaltenen Klassen und Methoden erfolgt in Abschnitt 4.2.2. Es sollte jedoch auch nach dieser ersten, oberflächlichen Beschreibung von SYSGP deutlich geworden sein, welche Möglichkeiten SYSGP einem Entwickler von auf Genetischer Programmierung basierenden Programmen bietet.

Ein solcher Entwickler, welcher mit Hilfe der Genetischen Programmierung und unter Rückgriff auf SYSGP ein Programm generieren möchte, kann dies tun,

indem er folgende Komponenten realisiert:

- eine Strukturdatei
- das Hauptprogramm
- eine Funktion zur Bewertung der Fitness der einzelnen Individuen

In der *Strukturdatei* werden zum einen allgemeine Parameter zur Evolution festgelegt, so z.B. die Größe des zu betrachtenden Pools, die Anzahl der gewünschten Generationen sowie Mutations- und Crossover-Wahrscheinlichkeiten. Zum anderen wird in der Strukturdatei festgelegt, wie viele Programme ein zu erzeugendes Individuum besitzen kann, welcher Art (linear, Baum, Graph) und Größe das Programm sein soll sowie welche Operationen im jeweiligen Programm genutzt werden können. Außerdem können neben den von SYSGP vorgesehenen Parametern weitere programmspezifische Parameter angegeben werden, welche vom Hauptprogramm ausgewertet werden. Dies kann der Name einer Datei sein, in welche eine Ausgabe erfolgen soll. Der Aufbau einer Strukturdatei sowie die von SYSGP vorgesehenen Parameter werden detailliert in Abschnitt 4.2.6 erläutert.

Das *Hauptprogramm* bildet den Kern der Entwicklung. Im Hauptprogramm müssen die für die Evolution notwendigen Objekte instanziiert und initialisiert werden. Darunter fällt neben dem Einlesen der in der Strukturdatei gemachten Angaben das Erzeugen des Environments, das Definieren der zu nutzenden Operationen sowie das Anlegen eines Pools von Individuen, auf dem die Evolution durchgeführt werden soll. Darüberhinaus wird im Hauptprogramm die Art und Weise festgelegt, in welcher die Evolution vollzogen werden soll. Dazu wird auf die von SYSGP bereitgestellten genetischen Operationen und Selektionsmethoden zurückgegriffen. Diesen Teil des Hauptprogrammes bezeichnet man als Evolutionsschleife. Der Prozess der Evolution kann über im Hauptprogramm verankerte Ausgabebefehle beobachtet werden. Ferner kann hier eine erste Anwendung des gefundenen „Siegers“ der Evolution auf das zugrundeliegende Problemfeld erfolgen. Eine genauere Beschreibung des Hauptprogrammes befindet sich in Abschnitt 4.2.7.

Schließlich muss eine *Fitnessfunktion* zur Bewertung der erzeugten Individuen erstellt werden. Die Bewertung spiegelt die Fähigkeit des Individuums wieder, die zugrundeliegende Aufgabe zu lösen. Diese Fitnessfunktion wird herangezogen, wenn es während der Evolution zur Selektion von Individuen kommt. Eine geschickte Wahl der Fitnessfunktion ist von entscheidender Bedeutung für den Erfolg der Evolution, da hier die Beziehung zwischen den (zufällig erzeugten) Individuen und dem Problemfeld hergestellt wird. Der Implementierung der Fitnessfunktion kommt folglich bezüglich des Gelingens des Gesamtprogramms eine bedeutsame Rolle zu. Auf die Fitnessfunktion wird ebenfalls in Abschnitt 4.2.7 genauer eingegangen.

SYSGP bildet zusammen mit der Fitnessfunktion quasi den GP-Werkzeugkasten, aus dem sich der Entwickler bedienen kann. Mit Hilfe der zur Verfügung gestellten Mittel kann genetisch programmiert werden, ohne sich um die tieferliegenden Details – abgesehen von der Implementierung der Fitnessfunktion –

zu kümmern. Das Programmieren einer Fitnessfunktion kann dem Entwickler wegen deren engem Bezug zum Problemfeld nicht abgenommen werden. Die Verwendung einer Strukturdatei ermöglicht es, das Verhalten des Programmes sowie der Fitnessfunktion in verschiedenen Umgebungen zu analysieren. Es ist ohne Schwierigkeiten möglich, das gleiche Hauptprogramm für variierende Poolgrößen, für Individuen mit linearen Programmen oder für Individuen mit Programmen basierend auf Bäumen oder Graphen auszuführen. Hierzu ist lediglich eine Änderung der Strukturdatei erforderlich und nicht etwa das Editieren und Neucompilieren des Quelltextes.

4.2.2 Zusammenspiel der sysgp-Komponenten

Dieser Abschnitt behandelt die wichtigsten in SYSGP enthaltenen Klassen und Methoden, um einen Einblick auf die Konzeption von SYSGP zu ermöglichen. Im einzelnen sind dies:

- SYSGP_Env
- SYSGP_Ind
- SYSGP_InterpretInd
- SYSGP_Op
- SYSGP_Opset
- SYSGP_Pool
- SYSGP_PrgGraph
- SYSGP_PrgLin
- SYSGP_PrgTree
- SYSGP_Struc

Die Klassen werden im folgenden einzeln beschrieben, ihre Bedeutung für die Bibliothek SYSGP dargestellt.

An einigen Stellen des Textes wird Bezug genommen auf die Klassen SYSGP_Object und SYSGP_Interpret. Diese Klassen beinhalten keine nennenswerte Funktionalität und werden deswegen im Gegensatz zu den oben genannten Klassen nicht explizit beschrieben. Ihre Bedeutung besteht vielmehr in der „Schnittstellenspezifikation“. In diesem Sinne ist die Klasse SYSGP_Object Oberklasse der Klassen SYSGP_Ind und SYSGP_IndSet, die Klasse SYSGP_Interpret Oberklasse der Klassen SYSGP_InterpretTree, SYSGP_InterpretLin, SYSGP_InterpretIndSet, SYSGP_InterpretInd sowie SYSGP_InterpretGraph.

Darüber hinaus beinhaltet SYSGP Methoden zur Selektion sowie zur Anwendung der Genetischen Operatoren. Diese sind nicht in Klassen gekapselt sondern liegen als Funktionen vor. Die Beschreibung dieser Methoden erfolgt im Anschluss an die Klassendarstellung in den Abschnitten:

- Selektion (Abschnitt 4.2.4)
- Genetische Operatoren (Abschnitt 4.2.5)

Zunächst soll jedoch das Zusammenwirken der Klassen erläutert werden. In der Klasse `SYSGP_Structure` werden die in der Strukturdatei gemachten Angaben verwaltet und verfügbar gehalten. Muss an einer Stelle beispielsweise in Erfahrung gebracht werden, wie viele Individuen der Entwickler vorgesehen hat, so kann dies über eine Anfrage an die Klasse `SYSGP_Structure` geschehen. Sie kann als Sammelstelle für die angegebenen Parameter verstanden werden.

Die Klasse `SYSGP_Ind` beinhaltet ein einzelnes Individuum, welches an der Evolution beteiligt sein kann. Jedes Individuum beinhaltet ein oder mehrere Programme zur Lösung des zugrundeliegenden Problems. Eine Menge von Individuen kann in der Klasse `SYSGP_Pool` verwaltet werden. Diese beiden Klassen verkörpern gleichsam die Objekte (hier nicht im Sinne der objektorientierten Programmierung zu verstehen), auf welche die Genetischen Operatoren angewandt werden.

Die Programme der Individuen können vom Typ lineares Programm, Baum oder Graph sein. Die Modellierung dieser Programmtypen erfolgt durch die Klassen `SYSGP_PrgLin`, `SYSGP_PrgTree` und `SYSGP_PrgGraph`. Da die Programme nicht nur dargestellt, sondern auch ausgeführt werden müssen, liegt zu jedem Programmtyp ein Interpreter vor (`SYSGP_InterpreterLin`, `SYSGP_InterpreterTree`, `SYSGP_InterpreterGraph`). Da außerhalb der Bibliothek `SYSGP` auf die Programme der Individuen an sich nie direkt sondern nur über die Individuen zugegriffen wird, ist für den Entwickler die Klasse `SYSGP_InterpreterInd` von Bedeutung. Dieser ruft nach Auswertung des Programmtyps eines Individuums den entsprechenden Interpreter auf.

Welche Operationen sollen in einem Programm verwendet werden können? Das Festlegen der Operationen, die einem Programm zur Verfügung stehen sollen, erfolgt in der Strukturdatei, die Einstellungen werden in ein Objekt der Klasse `SYSGP_Struc` übernommen. Eine durchzuführende Operation muss aber natürlich auch definiert werden. Falls der Entwickler nicht mit den in `SYSGP` vorhandenen Operationen auskommt, muss er zusätzlich benötigte selbst implementieren. Die Einbindung der Operationen in die Programme erfolgt über Objekte der Klasse `SYSGP_Op`, welche die definierten Operationen kapseln. Eine Menge von `SYSGP_Op`-Objekten wird in der Klasse `SYSGP_Opset` zusammengefasst.

Anstelle einer Funktion kann als Operation auch ein Unterprogramm genutzt werden, ein **adf-Programm**. Ein solches Programm entspricht im Wesentlichen einem Hauptprogramm, kann also ebenfalls als lineares Programm, Baum oder Graph vorliegen. Dementsprechend ist ein ADF-Programm ein Objekt der Klassen `SYSGP_PrgLin`, `SYSGP_PrgTree` oder `SYSGP_PrgGraph`. Der grundsätzliche Unterschied zu einem Hauptprogramm besteht darin, dass das ADF-Programm den Charakter einer Operation bzw. eines Terminals hat und somit in einem Hauptprogramm als Unterprogramm genutzt werden kann.

Zur Ausführung der Programme ist es erforderlich, dass eine Umgebung vorhanden ist, in der die Programme operieren können. Neben einem Speicher und

Registern wird ein Stack zur Verfügung gestellt. Diese Umgebung wird durch die Klasse `SYSGP_Env` realisiert.

Zusammenfassend werden in einem `SYSGP_Pool`-Objekt die betrachteten Individuen gehalten, welche Instanzen der Klasse `SYSGP_Ind` sind. Die in den Individuen enthaltenen Programme (je nach Typ Objekte der Klassen `SYSGP_PrgLin`, `SYSGP_PrgTree` oder `SYSGP_PrgGraph`) basieren auf `SYSGP_Op`-Objekten, welche die in `SYSGP` enthaltenen oder vom Entwickler zusätzlich definierten Funktionen kapseln. Die Ausführung der Programme eines Individuums erfolgt über einen Interpreter der Klasse `SYSGP_InterpreterInd`. Die zur Ausführung notwendige Umgebung befindet sich in der Klasse `SYSGP_Env`.

Standardmäßig befindet sich der Quellcode einer Klasse in der Headerdatei, die den Namen des Klassenbezeichners ohne das Präfix `SYSGP`. So kann der Quellcode der Klasse `SYSGP_Env` in der Datei `Env.hh` der Bibliothek gefunden werden. Der Quellcode wird in Headerdateien gefasst, um Problemen vorzubeugen, welche in Zusammenhang mit der Verwendung von Templates auftreten können (s. u.).

Bevor nun die Beschreibung der einzelnen Klassen folgt, noch ein Wort zur Implementierung der Bibliothek: Die notwendige Flexibilität, welche notwendig ist, um der Breite von Problemfeldern begegnen zu können, in denen Genetische Programmierung anwendbar ist, wird durch die Verwendung von Templates erreicht. Die dadurch auf den ersten Blick verwirrende Klassen- und Methodensignaturen werden verständlicher, wenn man das Template `<T>` grundsätzlich als den Typ der von den Programmen zu verwendenden Variablen betrachtet. Entsprechend steht das Template `<Op>` dann für Funktionen (Operationen), welche auf Variablen des Typs `<T>` operieren und das Template `<Env>` für eine Umgebung (Klasse `SYSGP_Env`), in der Variablen dieses Typs gehalten werden sollen. Diese Konvention wird innerhalb der Bibliothek weitestgehend eingehalten.

4.2.3 Die sysgp-Klassen

1. `SYSGP_Env<T>`

Die Klasse `SYSGP_Env<T>` modelliert die Umgebung, in welcher ein Programm ausgeführt werden kann. Dazu werden ein Speicher und zwei Registersätze („normale“ und ADF-Register) sowie ein Stack bereitgestellt. Die vorhandenen Methoden erlauben das Lesen und das Schreiben auf Speicher und Registern und stellen die gängigen Stackoperationen zur Verfügung.

Für die Ausführung von Programmen des Typs `Graph` wird in einer Integervariablen `mRunTime` die maximale Laufzeit des Programmes angegeben.

2. `SYSGP_Ind<T>`

Die Klasse `SYSGP_Ind` modelliert ein Individuum. Jedes Individuum kann mehrere Hauptprogramme und mehrere ADF-Programme besitzen. Die Programmtypen dieser Programme sind beliebig. Außerdem besitzt ein Individuum einen Fitnesswert, welcher die Bewertung des Individuums enthält. Die Methoden der Klasse erlauben das zufällige Erzeugen eines

Individuums. Der Prozess des Erzeugens orientiert sich jedoch immer an den in der Strukturdatei gemachten und in das Strukturobjekt eingelesenen Angaben. Darüberhinaus stehen diverse Methoden zum Setzen und Auslesen der Eigenschaften eines Individuums zur Verfügung.

3. `SYSGP_InterpreterInd <T, Env>`

Die Klasse `SYSGP_InterpreterInd` stellt den zur Ausführung (der Programme) der Individuen notwendigen Interpreter. Die Klasse übernimmt die Interpretation jedoch nicht selber, sondern ruft abhängig vom Typ des auszuführenden Programmes Methoden der spezielleren Interpreter `SYSGP_InterpreterLin`, `SYSGP_InterpreterTree` oder `SYSGP_InterpreterGraph` auf. Zur Ausführung des Programmes eines Individuums, wie es beispielsweise zur Bewertung erforderlich ist, wird dem Interpreter das zu betrachtende Individuum übergeben. Anschließend wird die Ausführung der Programme angestoßen, ggf. kann ein Individuum ja mehrere Hauptprogramme aufweisen. Die Resultate, die jedes Hauptprogramm liefert, werden in einem Array abgelegt. Die Inhalte dieses Arrays können schließlich vom Interpreter-Objekt abgefragt werden.

Desweiteren ist es möglich, einzelne ADF-Programme eines Individuums auszuführen. Dabei wird das Resultat direkt als Ergebnis bei Ausführen des Programmes zurückgegeben.

4. `SYSGP_Op <T, Env>`

Die Klasse dient dazu, die von einem Programm nutzbaren Operationen zu erfassen. Unter einer Operation wird in diesem Zusammenhang entweder eine definierte Funktion oder ein ADF-Programm verstanden.

Verkörpert die Operation eine Funktion, so muss diese implementiert und bei Instanziierung des entsprechenden `SYSGP_Op`-Objektes sichtbar (im Sinne eines Gültigkeitsbereiches) sein. Das `SYSGP_Op`-Objekt erhält bei Aufruf des Konstruktors die Funktion als Parameter übergeben. Bei Anwendung der Operation erfolgt dann schlichtweg ein Aufruf dieser Funktion. Bei Instanziierung eines `SYSGP_Op`-Objektes dieser Art entfällt die Angabe des fünften Parameters im Konstruktor. Näheres zur Verwendung von (selbstdefinierten) Funktionen als Operationen kann dem Abschnitt [4.2.8](#) entnommen werden.

Anders verhält es sich, wenn das Objekt Platzhalter für ein ADF-Programm ist. In diesem Fall enthält der fünfte Parameter die Position des ADF-Programmes unter den ADF-Programmen eines Individuums, welches ausgeführt werden soll. Es ist offensichtlich, dass hier nun die Angabe einer Funktion entfallen kann. Daher wird im Konstruktor bei Instanziierung eines `SYSGP_Op`-Objektes meist anstelle der Funktion `NULL` übergeben.

Die Interpreter-Klassen identifizieren ein `SYSGP_Op`-Objekt nicht über seinen Bezeichner, sondern über seinen Namen, welcher im Konstruktor als zweites Argument übergeben wird. Die einem Programm zur Verfügung

stehenden Operationen sind jedoch nicht identisch mit der Menge aller definierten Operationen. Vielmehr muss sich ein Programm auf die ihm laut Strukturdatei zustehenden Operationen beschränken.

5. `SYSGP_Opset <Op>`

Die Klasse `Opset` dient als Container für Objekte der Klasse `SYSGP_Op`. Sie wird nur zur Initialisierung des `SYSGP_Struc` Objektes nötig. `SYSGP_Struc` liest aus diesem Objekt die Funktionen (Operatoren) aus und fügt sie nach den Vorgaben der Strukturdatei in die den ADF- und Hauptprogrammen zur Verfügung stehenden Funktions- und Terminalmengen ein.

6. `SYSGP_Pool <T, IND, Op>`

Die Klasse `SYSGP_Pool` modelliert den Pool, in dem die Evolution stattfindet. Hier werden die Individuen zusammengefasst, auf welche die Genetischen Operatoren angewandt werden können. Neben dieser Funktion als Behälter von Individuen bietet die Klasse `Pool` Methoden zur Verwaltung und Organisation des Pools. So ist es möglich, den Pool mit zufälligen Individuen zu initialisieren. Die Zufälligkeit bewegt sich auch hier wieder in den Bahnen, welche durch die Strukturdatei vorgegeben wurden. Desweiteren können die im Pool vorhandenen Individuen entsprechend ihrer Fitness sortiert werden.

Für die Anwendung der Genetischen Operatoren ist ein weiteres Merkmal eines Pools von Bedeutung: Er beinhaltet ein Array `mPartners`, welches auf Individuen verweist, die an einer Genetischen Operation teilnehmen können. Welche Individuen in diesem Array stehen, hängt von der gewählten Selektionsmethode ab. Zur Verwaltung dieses Arrays stehen ebenfalls Methoden zur Verfügung.

7. `SYSGP_PrgGraph <T, Op>`

Die Klasse `SYSGP_PrgGraph` ist, wie alle Programm-Klassen eine Unterklasse der Klasse `SYSGP_Prg`. Die Graph-Eigenschaften erbt sie von der Klasse `SYSGP_Graph`.

Der Graph besteht aus einer Menge von Knoten, welche miteinander über Kanten verknüpft sind. Es existieren zwei ausgezeichnete Knoten, der Startknoten und der Endknoten. Intern wird die Knotenmenge als Array von Graphknoten (Klasse `SYSGP_GraphNode`) gehalten. Der Startknoten ist der im Array an zweiter Stelle (Position 1) stehende Knoten. Als Endknoten wird der im Array an erster Stelle (Position 0) stehende Knoten betrachtet.

Ein Knoten des Graphen besteht aus zwei Komponenten, einmal dem Inhalt des Knoten, genannt `mObject`, zum anderen dem Array aus Nachfolgeknoten (Adjazenzliste) namens `mChildren`. Das Array `mChildren` beinhaltet die Position eines jeden Nachfolgeknotens im Knotenarray des Graphen. Der Inhalt `mObject` eines Knotens besteht zunächst aus einem Teil, der angibt, welche Operation bei Erreichen des Knotens auszuführen ist, bzw.

welcher Registerinhalt / welche Konstante im Knoten enthalten ist. Diese Information wird vom Interpreter als erstes ausgewertet. Der restliche Teil von `mObject` liefert Informationen bzgl. des Nachfolgeknotens, mit dem das Programm fortgesetzt werden soll. Zum einen ist hier eine Verzweigungskonstante zu finden, zum anderen eine Verzweigungsfunktion. Der Interpreter ruft die Verzweigungsfunktion auf, welche im Allgemeinen aus Verzweigungskonstante und zuvor durch den Interpreter ermittelten Wert ein Ergebnis berechnet (Integer-Wert). Aufgrund dieses Ergebnisses wählt der Interpreter einen Nachfolgeknoten aus.

Die Knoten werden beim Erzeugen des Graphen generiert und initialisiert. Sowohl die Adjazenzliste `mChildren` als auch der Inhalt des Knotens können zufällig gesetzt werden. Dabei werden die in der Strukturdatei gemachten Angaben berücksichtigt.

Bei Interpretation eines Graphen wird mit dem Startknoten begonnen und solange mit einem Nachfolgeknoten weitergemacht, bis entweder der Endknoten erreicht oder die maximale Laufzeit abgelaufen ist. Die Ergebnisse der jeweiligen Knoten werden auf den Stack gelegt. Das Ergebnis der Ausführung ist der Inhalt der Speicherzelle 0.

8. `SYSGP_PrgLin <T, Op>`

Die Klasse `SYSGP_PrgLin` ist, wie alle Programm-Klassen eine Unterklasse der Klasse `SYSGP_Prg`.

Ein lineares Programm besteht aus einer Folge von Befehlen. In `SYSGP` wird ein jeder solcher Befehl in einem Objekt der Klasse `SYSGP_LinPrgNode` abgelegt. Das Programm, also das `SYSGP_PrgLin`-Objekt beinhaltet ein Array von `SYSGP_LinPrgNode`-Objekten. Der wichtigste Bestandteil eines `SYSGP_LinPrgNode`-Objektes ist ein Array, welches die einzelnen Elemente eines Befehls, also den Operator und die beteiligten Argumente (bis auf das erste) enthält. Daneben werden zwei Register geführt. In ein Register (`mResultReg`) wird das Ergebnis geschrieben, das andere (`mFirstReg`) enthält das erste Argument. Der Operator steht immer an Position 0 des Arrays. Es kann sich dabei um eine Funktion oder ein ADF-Programm handeln. Je nach Operator enthält der Rest des Arrays eine Anzahl von Argumenten in Form von Konstanten, Registern oder Terminalen. Handelt es sich bei dem Operator um einen Sprungbefehl, so enthält das erste Register `mFirstReg` die Zieladresse. Es ist zu beachten, dass im linearen Programm nur Sprünge erlaubt sind, die Richtung Programmende verlaufen.

Die Befehlszeilen können beim Anlegen eines linearen Programmes zufällig erzeugt werden. Dabei werden die in der Strukturdatei gemachten Angaben berücksichtigt.

Die Ausführung eines linearen Programmes beginnt mit der Befehlszeile bzw. mit dem `SYSGP_LinPrgNode`-Objekt, welches im Array des Programms an Position 0 steht. Das Programm wird, abgesehen von Vorwärtssprüngen, Zeile für Zeile ausgeführt, bis die letzte Befehlszeile bearbeitet

wurde. Die Ergebnisse der einzelnen Zeilen werden im jeweiligen Ergebnisregister `mResultReg` abgelegt. Das Resultat des Programmes wird aus der Speicherzelle 0 ausgelesen.

9. `SYSGP_PrgTree <T, Op>`

Die Klasse `SYSGP_PrgTree` ist, wie alle Programm-Klassen eine Unterklasse der Klasse `SYSGP_Prg`. Die Baum-Eigenschaften erbt sie von der Klasse `SYSGP_Tree`.

Ein Programm vom Typ Baum setzt sich zusammen aus Knoten der Klasse `SYSGP_Node`. Ein solcher Knoten besteht aus zwei Teilen. Zum Einen wird ein Array von Knoten gehalten, die die Kinder des betrachteten Knotens sind. Zum Anderen wird der Inhalt `mObject` des Knotens verwaltet, welcher eine Operation (Funktion oder ADF-Programm), eine Konstante oder ein Register enthalten kann. Sprünge sind bei Baum-Programmen nicht möglich. Falls der Knoten eine Operation enthält, so werden die Argumente durch die von den Kindern des Knotens gelieferten Werte gestellt.

Ein Baum-Programm kann zufällig erzeugt werden. Dabei liegen an inneren Knoten nur Funktionen oder ADF-Programme, die Blätter werden ausschließlich durch Register, Konstanten oder Terminale gebildet (ADF-Programme können Terminale verkörpern). Die in der Strukturdatei gemachten Angaben werden beim Anlegen des Baum-Programmes berücksichtigt.

Die Interpretation eines Baum-Programmes erfolgt rekursiv. Es wird zunächst die Wurzel des Baumes betrachtet. Da im Allgemeinen hier kein Blatt stehen wird, muss zur Berechnung die Auswertung der Kinder des Knotens erfolgen, da diese die Argumente für den Vater-Knoten bereitstellen. Auf diese Weise wird der Baum einem DFS-Ansatz folgend Schritt für Schritt ausgewertet, bis schließlich als Resultat des Programmes die an der Wurzel stehende Operation durchgeführt werden kann.

10. `SYSGP_Struc <Op>`

Die Klasse `SYSGP_Struc` liest die Strukturdatei und macht die darin enthaltenen Parameter für andere Objekte zugänglich. Aus diesem Grunde enthält die Klasse Attribute, welche die in der Strukturdatei gemachten Angaben aufnehmen können, und Methoden, um auf diese Attribute zuzugreifen.

Der Inhalt der Strukturdatei wird im `SYSGP_Struc`-Objekt wie folgt abgelegt: Für allgemeine Angaben wie Populationsrate oder Anzahl an Generationen stehen im Objekt selber Attribute zur Verfügung, welche die entsprechenden Werte aufnehmen. Gleiches gilt für die benutzerdefinierten Parameter, welche in einer Liste abgelegt werden. Die „Schablonen“ für die einzelnen Individuen werden in einem Array `mStructList` geführt. Jedes Element des Arrays steht für ein Individuum. Die Arrayelemente nehmen Angaben bezüglich der einzelnen Individuen auf. Für Haupt- und ADF-Programme enthalten sie wiederum Arrays, in welche die Daten der

Strukturdatei übertragen werden. Zur Veranschaulichung ein kleines Beispiel: Die Angaben für das zweite Hauptprogramm des dritten Individuums befinden sich, da es sich um das dritte Individuum handelt, im Array `mStructList` an Position 2 (Array beginnt bei 0). Im entsprechenden Element muss Position 1 des Arrays für Hauptprogramme betrachtet werden. Achtung: Die im Array `mStructList` erscheinenden Angaben sind als Schablonen zu begreifen. Ist hier nur ein Element zu finden, so bedeutet das nicht, dass nur ein Individuum erzeugt werden kann ;-).

Eine der im Array `mStructList` enthaltenen Schablonen wird als aktuelle Schablone betrachtet. Die Position dieser Schablone im Array wird im Attribut `mStructNum` festgehalten. Ebenso gibt es zu jeder Schablone ein aktuelles Programm. Die Position des aktuellen Programmes ist im Attribut `mPrgNum` abgelegt. Dieser Wert gilt sowohl für das Array der Hauptprogramme als auch für das Array der ADF-Programme.

4.2.4 Selektion

Die zur Durchführung der Selektion in `SYSGP` vorhandenen Methoden sind in der Datei `Selection.hh` zu finden. Im einzelnen sind dies:

- `void SYSGP_poolProportional(SYSGP_Pool<T,IND,Op> & pool, double maxrang=1.4)`

Die geeignetsten Individuen werden auf die weniger geeigneten kopiert. Der Parameter `maxrang` gibt an, wie viele der besseren Individuen kopiert werden. Dabei gilt, dass mindestens ein Anteil von $(2 - \text{maxrang}) * 100$ Prozent des Pools kopiert wird, maximal jedoch die Hälfte des Pools. Voraussetzung: Die besten Individuen befinden sich am Anfang des Pools.

- `void SYSGP_poolBest(SYSGP_Pool<T,IND,Op>& pool, int alife)`

Es werden die `alife` besten Individuen in die Mitte des Pool kopiert. Dafür werden alle Individuen an den Positionen des Bereiches `[alife; Pool-Größe - alife]` des Pools durch Kopien der `alife` beste Individuen ersetzt. Voraussetzung: Die besten Individuen befinden sich am Anfang des Pools. Achtung: `alife` muss kleiner als die Hälfte der Pool-Größe sein.

- `void SYSGP_poolBestAge(SYSGP_Pool<T,IND,Op> & pool, int alife, int age)`

Unter den `alife` besten Individuen werden zunächst solche bestimmt, welche mehr als `age` Generationen durchlebt haben. Gibt es unter den restlichen, schlechteren Individuen eines, dessen Alter geringer ist als `age`, so wird ein Tausch vorgenommen. Sind unter den `alife` besten Individuen die zu alten soweit wie möglich ausgetauscht, wird wie unter `SYSGP_poolBest` verfahren.

- `void SYSGP_tournementSelection(SYSGP_Pool<T,IND,Op> & pool, SYSGP_Fitness * fit, int begin, int end, int size, int`

fitFlag)

Es werden zwei Tournaments der Größe size gebildet. Dabei wird nur auf Individuen zurückgegriffen, welche im Pool zwischen den Positionen begin und end stehen. Das beste Individuum eines Tournaments ersetzt das schlechteste. Die beiden Gewinner der jeweiligen Tournaments bleiben im Array mPartners des Pools und stehen somit für die Anwendung Genetischer Operatoren bereit. Die Fitness-Funktion fit ermöglicht die Bewertung der Individuen in den Tournaments. Der Parameter fitFlag bestimmt, ob ein geeignetes Individuum einen hohen (Wert 1) oder einen niedrigen (Wert 0) Fitness-Wert aufweist.

- void SYSGP_tournemantSelection(SYSGP_Pool<T,IND,Op> & pool, int begin, int end, int size, int fitFlag)

Analog zur vorheriger SYSGP_tournemantSelection-Methode bis auf die Einschränkung, dass die Fitness der Individuen nicht neu berechnet, sondern aus den Fitness-Attributen der Individuen ausgelesen wird.

- int SYSGP_tournemantSmallest(SYSGP_Pool<T,IND,Op> & pool, int size)

Es wird ein Tournament der Größe size gebildet. Die Methode liefert die Position des Individuums mit der geringsten Fitness im Tournament zurück. Die Methode operiert auf den in den Individuen abgelegten Fitness-Werten.

- int SYSGP_tournemantGreatest(SYSGP_Pool<T,IND,Op> & pool, int size)

Analog zu SYSGP_tournemantSmallest mit der Einschränkung, dass die Position des Individuums mit der größten Fitness zurückgeliefert wird.

- SYSGP_Array<int> SYSGP_tournemant(SYSGP_Pool<T,IND,Op> & pool, int size, SYSGP_Array<int>& forbidden)

Die Methode liefert ein Tournament der Größe size zurück, welches aufsteigend nach der Fitness der Individuen sortiert ist. Das Array forbidden enthält die Positionen der Individuen, welche beim nächsten Tournament ausgeschlossen sein sollen. Die Methode operiert auf den in den Individuen abgelegten Fitness-Werten.

4.2.5 Genetische Operatoren

Die Methoden, welche die Genetischen Operatoren implementieren, befinden sich in der Datei IndXover.hh. Es sind dies:

- void SYSGP_Ind_XoverRand(SYSGP_Ind<T,Op> *prg1, SYSGP_Ind<T,Op> *prg2, SYSGP_Struc<Op>* st)
- void SYSGP_Ind_MutatRand(SYSGP_Ind<T,Op> *prg1, *SYSGP_Struc<Op>* st, T (*rnd) (double, double))

Ähnlich der Klasse `SYSGP_InterpreterInd` wird das eigentliche Crossover bzw. die eigentliche Mutation von auf die vorliegenden Programm-Typen zugeschnittenen Methoden erledigt.

Bei Anwendung der Crossover-Methode ist zu beachten, dass die beteiligten Individuen „crossover-verträglich“ sind, d. h. bei beiden Individuen müssen die Programmtypen positionsweise übereinstimmen. Ein Individuum, welches als zweites Hauptprogramm eine Baumstruktur trägt kann nur mit einem Individuum gekreuzt werden, welches als zweites Hauptprogramm ebenfalls einen Baum besitzt. Die der Mutationsmethode übergebene Zufallsfunktion muss einen zufälligen Wert des Typs `T` berechnen. Sie wird zur Bestimmung von Konstanten benötigt.

4.2.6 Die Strukturdatei

Um die Flexibilität eines mit `SYSGP` erstellten Programmes auf einfache Weise umsetzen zu können, werden die berücksichtigten Parameter in einer Strukturdatei zusammengefasst. Es ist somit möglich, das gleiche Programm mit verschiedenen Einstellungen auszuführen, ohne dass es neu kompiliert werden muss. Der Inhalt einer Strukturdatei kann in ein `SYSGP_Struc`-Objekt eingelesen werden und steht so während der Programmausführung zur Verfügung.

Eine Strukturdatei muss folgendem Aufbau genügen:

```
Structure -----
{Abschnitt allgemeine Parameter}
Individualstructure -----

{Beginn eines Individuums}
Main -----
{Programmabschnitt}
Adf -----
{Programmabschnitt}
EndIndividual

BeginUserParameters
EndUserParameters
```

Eine Strukturdatei beginnt grundsätzlich mit der Zeichenkette „Structure“. Im Anschluss daran folgt ein Abschnitt, in dem allgemeine Parameter definiert werden. Dieser Abschnitt wird durch die Zeichenkette „Individualstructure“ abgeschlossen. Es folgt die Definition eines oder mehrerer Individuen. Zu Beginn einer Individuen-Definition ist es möglich, für das Individuum allgemein geltende Parameter zu setzen. Daran anschließend erfolgt die Spezifizierung der einzelnen Programme eines Individuums. Die Zeichenkette „EndIndividual“ schließt die Definition eines Individuums ab. Der durch die Zeichenketten „BeginUserParameters“ und „EndUserParameters“ eingerahmte Abschnitt am Ende einer Strukturdatei ermöglicht dem Entwickler die Verwendung eigener Parameter. Folgende zwei Tabellen listen die von `SYSGP` vorgesehenen Parameter auf:

Tabelle 3: Allgemeine Parameter

Parameter	Bedeutung
Population	Größe der Population.
Generation	Anzahl der Generationen.
RegisterSize	Anzahl der Register im Environment. Ist der Parameter > 0 , werden Registerfunktionen automatisch in die Terminalmenge aufgenommen.
MemorySize	Größe des Speichers im Environment.
StackSize	Größe des Stacks im Environment.
MaxAge	Maximales Alter der Individuen in Generationen, wird von einigen Selektionsmethoden benutzt.
DemeSize	Anzahl der Demes pro Dimension.
DemeDim	Dimension der Demes, 1= Listen, 2= Matrizen, 3= Körper. Der Parameter ist der Exponent von DemeSize.
DemeStruct	Struktur der Demes, list= Liste, ring= Ring (DemeDim= 1), matrix= Matrix (DemeDim= 2).
Crossoverrate	Crossoverrate in Prozent.
Mutationrate	Mutationsrate in Prozent.
MainCrossoverprob	Wahrscheinlichkeit in Prozent, das während des Crossovers ein Mainprogramm und nicht ein ADF gewählt wird. Ist der Parameter gleich Null werden beim Crossover sowohl Mainprogramm als auch die ADFs gewählt.
IndSetXoverCnt	Anzahl der Individuen des Individuentyps IndSet, die für ein Crossover ausgewählt werden.
mSeed	Seed des Zufallszahlengenerators, bei 0 wird ein Seed erzeugt.

Tabelle 4: Programmbezogene Parameter

Parameter	Bedeutung
MainTyp	Programmtyp, T= Baum, L= Linear, G= Graph.
MainXoverCnt	Anzahl der Mainprogramme, die für ein Crossover gewählt werden.
MainMutCnt	Anzahl der Knoten die während einer Mutation mutiert werden.
MainInitTyp	Initialisierungsart des Programms, grow= Zufällige größe in den Initialisierungsgrenzen, full= maximale Initialisierungsgröße, halfAndHalf= 50% grow und 50% full.
MainInitNodeCnt	Maximale Knotenanzahl während der Initialisierung.
MainInitMinNodeCnt	Minimale Knotenanzahl während der Initialisierung.
MainInitDepth	Maximale Tiefe während der Initialisierung.

Tabelle 4: Programmbezogene Parameter

Parameter	Bedeutung
MainNodeCnt	Maximale Knotenanzahl der Individuen während der Evolution.
MainMinNodeCnt	Minimale Knotenanzahl der Individuen während der Evolution.
MainDepth	Maximale Tiefe der Bäume während der Evolution.
MainTermProb	Wahrscheinlichkeit in Prozent, dass ein Knoten ein Terminal ist. Bei Graphen das ein Knoten ein Register oder eine Konstante enthält.
MainConstProb	Wahrscheinlichkeit in Prozent, dass ein Terminal eine Konstante ist.
MainXoverTermProb	Wahrscheinlichkeit in Prozent, dass ein Crossoverpunkt ein Terminal ist.
MainMutTermProb	Wahrscheinlichkeit in Prozent, dass eine Mutation an einem Terminal durchgeführt wird.
MinConstant	Untere Schranke der Konstanten.
MaxConstant	Obere Schranke der Konstanten.
MinBranch	Untere Schranke der Branchingkonstanten nur für Graphen.
MaxBranch	Obere Schranke der Branchingkonstanten nur für Graphen.
MainGrad	Maximaler Outgrad für Graphknoten.
LinBranch	Maximaler Branchwert für Lineareindividuen.
EsMinValue	Minimaler Wert eines Es-Wertes.
EsMaxValue	Maximaler Wert eines Es-Wertes.
EsMutation	Mutationsschrittweite.
EsIndvLength	Länge eines Es-Individuums.
EsParLength	Länge des Parametervektors.
Operatorenmenge (+ - */)	Hinweis auf die Menge der Operatoren. Operatoren die verwendet werden dürfen, die Menge muss in Klammern ein geschlossen sein. Die Operatoren müssen durch Leerzeichen getrennt sein.
Terminalmenge ()	Hinweis auf die Menge der Terminalen, beim Graphtyp stehen hier die Branchingoperatoren. Terminaloperatoren die verwendet werden.

Die Parameter der Tabelle für Programme sind die für Hauptprogramme. Bei Anwendung in ADF-Programmen ist ggf. ein „Main“ durch „Adf“ zu ersetzen. Soll ein Parameter für alle Programme eines Individuums gelten, so kann er vor das erste Hauptprogramm des entsprechenden Individuums gesetzt werden. In diesem Fall muss sowohl ein Präfix „Main“ als auch „Adf“ gestrichen werden. In letztgenannten Abschnitt gehört auch der Parameter RunTime, welcher die maximale Laufzeit für Graph-Programme bestimmt.

4.2.7 Komponenten eines Programmes

4.2.7.1 Die Fitness-Klasse

Zur Bewertung der Fitness von Individuen ist die Implementierung einer Fitness-Klasse erforderlich. Da die Fitness eines Individuums vom betrachteten Problemfeld abhängt, kann in der SYSGP-Bibliothek keine Methode zur Fitness-Berechnung enthalten sein. Es liegt jedoch die Klasse `SYSGP_Fitness` vor, welche als Oberklasse für die zu implementierende Fitness-Klasse genutzt werden muss. Diese Klasse enthält ausschließlich eine (virtuelle) Methode `calc`, welche ein beliebiges `SYSGP_Object` als Parameter erhält und die Fitness dieses Objektes berechnet. Als Maß für die Fitness wird ein Wert vom Typ `double` genutzt. Diese Methode muss erweitert und mit Funktionalität gefüllt werden. Die Gesichtspunkte, nach denen hierbei die Bewertung erfolgt, sind selbstverständlich frei wählbar.

Darüber hinaus kann die abgeleitete Fitness-Klasse nach den Vorlieben des Entwicklers mit Methoden versehen werden. Als sinnvolles Beispiel kann das Protokollieren der von den Individuen erlangten Fitness in einer Datei betrachtet werden.

4.2.7.2 Die Main-Methode

Die vom Entwickler geschriebene Main-Methode bildet das Herzstück des auf Genetischer Programmierung basierenden Programms. Hier werden die benötigten Objekte der in SYSGP enthaltenen Klassen instanziiert und die bereitgestellten Methoden angestoßen.

Zunächst muss der Entwickler die Entscheidung treffen, mit welchem Variablentyp (`int`, `double`, ...) die in den Individuen enthaltenen Programme operieren sollen. Diese Entscheidung ist zentral, da der Großteil der in SYSGP vorhandenen Klassen diesen Typ als Template `<T>` nutzt. Im Anschluss daran werden entsprechende Objekte für die Umgebung (`SYSGP_Env`), den Interpreter (`SYSGP_InterpreterInd`), die definierten Operationen (`SYSGP_Op` und `SYSGP_Opset`), das Strukturobjekt (`SYSGP_Struc`) sowie den Individuen-Pool angelegt (`SYSGP_Pool`). Näheres zur Definition eigener Funktionen wird im Abschnitt 4.2.8 erläutert.

Sind diese Objekte instanziiert, kann das eigentliche Programm implementiert werden. Im Detail hängt dieses ab von der gewählten Evolutions-Strategie. In der einen oder anderen Form muss eine Evolutionsschleife realisiert werden. In dieser Schleife können die SYSGP-Methoden zur Selektion und zur Anwendung der Genetischen Operatoren genutzt werden. Außerdem kommt spätestens hier die vom Entwickler geschriebene Fitness-Klasse zur Bewertung der Individuen ins Spiel.

4.2.8 Selbstdefinierte Funktionen

Zwei Bereiche sind bei der Definition eigener Funktionen von Interesse, zum einen die Programmierung der Funktion in C++, zum anderen das Verfügbarmachen dieser Funktionen für SYSGP-Objekte. Im Folgenden werden diese beiden

Bereiche behandelt.

4.2.8.1 *Verfügbarmachen von Funktionen*

Die schönste Funktion nützt nichts, wenn im Unklaren bleibt, wie sie in die zu erstellenden Programme integriert werden kann. Aus diesem Grunde muss eine Funktion in ein Objekt der Klasse `SYSGP_Op` gekapselt werden. Folgendes Beispiel zeigt, wie dieses bewerkstelligt wird:

```
SYSGP_Op<double,Env> Add(add, ‘+’,2,2);
```

Es wird ein Objekt namens `Add` erzeugt. Dieses Objekt kapselt die Funktion `add`, welche bei Instanziierung des Objektes bekannt sein muss. Der erste Parameter gibt also den Namen der zu kapselnden Funktion an. Der zweite Parameter ist ein String, über den das Objekt identifiziert wird. Steht in der Strukturdatei unter den Operatoren eines Programmes ein „+“, so steht diesem Programm die Funktion des Objektes zur Verfügung, welches als identifizierenden String ein „+“ enthält. Ist kein solches Objekt vorhanden, bricht das Programm ab.

Der dritte und vierte Parameter gibt an, wie viele Argumente die Funktion höchstens verarbeiten kann (dritter Parameter) und wie viele sie mindestens benötigt (vierter Parameter).

Darüber hinaus existiert ein fünfter Parameter. Dieser zeigt den Typ der Funktion an. Defaultwert für diesen ist `-1`, der Wert, welcher normale Funktionen identifiziert. Ist der fünfte Parameter mit einem Wert größer als `-1` belegt, so beinhaltet das `SYSGP_Op`-Objekt ein ADF-Programm. Der Parameter gibt in diesem Fall an, welches ADF-Programm gemeint ist. Folgende Zeile zeigt die Instanziierung eines `SYSGP_Op`-Objektes, welches das erste ADF-Programm (Position 0 unter den ADF-Programmen) eines Individuums bezeichnet. In diesem Fall kann die Angabe einer Funktion im ersten Parameter entfallen.

```
SYSGP_Op<double,Env > Adf0(NULL, ‘adf0’,0,0,0);
```

Für lineare Programme kann der fünfte Parameter mit `-2` belegt werden, wenn der Operator einen Sprung vornehmen soll. Wird beim zufälligen Anlegen eines linearen Programmes der Wert `-2` vorgefunden, so wird das erste Register mit einer zufälligen Sprungadresse belegt, an die ggf. verzweigt werden kann. Anmerkung: Unabhängig vom im Register vorgefundenen Wert kann nur Richtung Programmende verzweigt werden.

4.2.8.2 *Programmierung einer Funktion*

Bei Funktionen muss unterschieden werden zwischen Funktionen, welche einen Wert berechnen, sowie Branchingfunktionen für Graph-Programme.

Ein einfaches Beispiel einer Funktion, welche einen Wert berechnet, ist die in `SYSGP` enthaltene Funktion `add`:

```
template <class T,class Env>
T add(SYSGP_Interpret<T,Env>& x)
{
    T first= x.eval(0);
    T second= x.eval(1);
```

```

    return (first + second);
};

```

Natürlich handelt es sich bei Funktionen um Template-Methoden, da nicht von vorneherein feststeht, welchen Variablentyp sie behandeln, und damit auch nicht, welche Umgebung zur Verfügung steht. Ebenfalls klar sollte sein, dass der von der Funktion berechnete Wert vom Typ T ist.

Als einzigen Parameter erhält eine Funktion einen Interpreter (Objekt der Klasse `SYSGP_Interpreter`), mit dem die zu verknüpfenden Argumente ermittelt werden können. Dazu wird die Methode `eval(int)` des Interpreters aufgerufen. Dieser Methode wird die Position des zu ermittelnden Arguments übergeben (0 für das erste Argument, 1 für das zweite, usw.). Die Anzahl der abfragbaren Argumente ist begrenzt (vgl. Verfügbarmachen von Funktionen oben). Die so ermittelten Werte entsprechen bei einem linearen Programm den Argumenten der einzelnen Befehlszeilen, bei einem Baum-Programm den Ergebnissen der Kinder-Knoten und bei einem Graph-Programm dem obersten Element des Stacks. (Bei Graph-Programmen ist die Angabe der Position ohne Bedeutung). Aus den so ermittelten Argumenten kann das Resultat berechnet werden.

Branchingfunktionen sehen den „normalen“ Funktionen sehr ähnlich. Auch sie erhalten einen Interpreter übergeben und liefern ein Ergebnis vom Typ T. Da Branchingfunktionen jedoch spezielle Attribute eines Interpreters für Graph-Programme nutzen, muss der übergebene Interpreter auf den Typ `SYSGP_InterpreterGraph` gecastet werden. Dieser Cast ist unproblematisch, da nur Graph-Programme Branchingfunktionen nutzen. Zu den speziell benötigten Attributen zählt die Möglichkeit, auf das zuvor im Knoten berechnete Ergebnis zuzugreifen. Würde dies über die Methode `eval` geschehen, so würde der gesuchte Wert gelesen und vom Stack gelöscht. Dieses Verhalten ist jedoch unerwünscht. Außerdem beinhaltet jeder Knoten eine Branchingkonstante, welche bei der Berechnung des Nachfolgeknotens berücksichtigt werden kann. Es folgt eine einfache Branchingfunktion:

```

template <class T, class Env>
T addB(SYSGP_Interpreter<T,Env>& x)
{
    SYSGP_InterpreterGraph<T,Env>* xG = (SYSGP_InterpreterGraph<T,Env>*) &x;

    T first= xG->mLastAction;
    return ( first + xG->mBranchConstant);
};

```

4.3 C++

Da wir unsere Arbeit eng mit SYSGP verzahnen, ist es unabdingbar unsere Software in C++ zu realisieren. Da SYSGP für den GNU Compiler g++, Version 2.7.2.3 optimiert wurde, haben wir diesen Compiler auch für das Projekt gewählt. Am

Fachbereich Informatik der Universität Dortmund ist dieser Compiler im Modul `extra/gcc/2.7.2.3` vorhanden.

4.4 Doc++

Doc++ ist ein Werkzeug, um Dokumentationen für C++ und Java Quelltexte zu erstellen. Wir benutzen Doc++ in der Version 3.12. Erhältlich ist Doc++ unter <http://www.zib.de/Visual/software/doc++/>. Es lassen sich Dokumentationen in HTML und \LaTeX 2 ϵ ausgeben.

Um Doc++ zu benutzen, wird die Dokumentation in den Quellcode integriert und in einem separaten Arbeitsschritt extrahiert. Die Anweisungen für Doc++ werden in C++ kompatible Kommentare geschrieben und für Doc++ kenntlich gemacht (im Folgenden Doc++ Kommentare genannt). Daher ist für die normale Kompilierung kein zusätzlicher Arbeitsschritt notwendig. Doc++ Kommentare sehen wie folgt aus:

```
/** ... */
```

also zwei Sterne nach dem Schrägstrich oder

```
/// ...
```

drei Schrägstriche. Diese Kommentare beachtet Doc++ bei der Erstellung der Dokumentation. In diese Kommentare können neben allgemeinen Beschreibungen auch *tags* geschrieben werden, die von Doc++ ausgewertet werden. Ein Beispielkommentar für eine Klasse sieht folgendermaßen aus:

```
/**
 * Chessinterface. Startet die crafty Instanzen und regelt die
 * Kommunikation zwischen der Klasse FDB_Spiel und crafty.
 * @author Patrick Gundlach
 * @version 1.0
 */
class FDB_Chessinterface {
...

```

Vor den Methoden-Deklarationen der einzelnen Klassen sind auch Kommentare erlaubt:

```
/** Prüft, ob setboard funktioniert hat
    @return 0 bei Fehler, 1 bei OK
 */
int checkOk();
```

Um die Dokumentation zu erstellen, wird das Kommando `doc++` aufgerufen. Folgender Beispieleintrag in einem Makefile (siehe Abschnitt 4.5 auf der nächsten Seite) erzeugt die HTML Dokumentation:

```
header = $(wildcard *.hh)
dxx    = $(patsubst %.hh,%.dxx,$(header))
```

```

%.dxx : %.hh
        docify $< $@

doc : $(dxx)
        doc++ -d docs *.dxx
        -rm *.dxx

```

Um also die endgültige Dokumentation zu erstellen, muss zuerst `docify` und anschließend `doc++` aufgerufen werden. Die so erstellte Dokumentation kann mit jedem HTML-fähigen Browser angezeigt werden. Der von `Doc++` erzeugte \LaTeX -Code ist teilweise fehlerhaft. Deshalb wird er in unserem Projekt nicht eingesetzt.

4.5 Make

4.5.1 Allgemeines

Das Werkzeug `MAKE` entscheidet, welche Teile eines größeren Programms neu kompiliert werden müssen und führt die entsprechenden Kommandos automatisch aus. `MAKE` ist aber nicht nur auf Softwareprojekte begrenzt; vielmehr eignet es sich für (fast) alle Projekte, die aus mehreren einzelnen Dateien bestehen. Beispielsweise besteht dieses Dokument aus mehreren einzelnen (\LaTeX) Dateien, dessen Kompilat (`dvi`) automatisch per `MAKE` aktualisiert wird.

Mit `MAKE` wird in diesem Abschnitt `GNU MAKE` in der Version 3.77 beschrieben. Auf den Solaris-Systemen am Fachbereich Informatik der Uni Dortmund ist `GNU MAKE` unter dem Namen `gmake` im Modul `stdenv` zu finden. Es gibt auf verschiedenen Unix-Systemen eine Vielzahl ähnlich funktionierender `MAKES`. `GNU make` ist weit verbreitet u. a. durch Linux. Einige fortgeschrittene Funktionen bietet nur `GNU make`. Daher wird im Folgenden `GNU MAKE` beschrieben.

Informationen über `MAKE` erhält man mit dem Befehl `info make`. Eine Übersicht über die Kommandozeilenparameter gibt es mit `make -h`. Verweise auf die Info Dokumentation wurden mit dem (\rightarrow) Zeichen kenntlich gemacht. Wenn möglich, wird dabei ein Hinweis auf das entsprechende Kapitel mit angegeben.

Um `MAKE` zu benutzen, muss vorher ein Makefile erstellt werden. In diesem wird beschrieben, welche Quelldateien sich wie zu den Zieldateien verhalten und mit welchen Programmen und mit welchen Regeln sich diese Zieldateien erzeugen lassen. Existiert ein solches Makefile, dann kann `MAKE` über die Kommandozeile im `xterm` aufgerufen werden. `MAKE` sucht nach einer Datei mit dem Namen `Makefile` (\rightarrow 3.2, What Name to Give Your Makefile) und führt daraufhin seine Aktionen aus.

4.5.2 Regeln

Gewöhnlich enthält ein Makefile *Regeln*. Eine Regel ist folgendermaßen aufgebaut:

```
target ... : dependencies ...
```

```
command
...
...
```

Ein *target* ist normalerweise eine ausführbare Datei, eine object-Datei (.o), eine dvi-Datei oder Ähnliches. Es kann aber auch ein symbolischer Name sein, der eine Aktion starten soll. Ein Beispiel hierfür ist *clean* (s. u.).

dependencies sind Quelldateien, auf denen die *targets* aufbauen. Beispiel hierfür sind header-Dateien (.h, .hh), source-Dateien (.c oder .cc) oder T_EX-Dateien (.tex).

command beschreibt Kommandos, die in einer Subshell (/bin/sh, → 5.2, Command Execution) ausgeführt werden. Meistens stehen hier die Kommandos, die notwendig sind, um aus den Quelldateien Zieldateien zu generieren. **Achtung:** das erste Zeichen in den Zeilen der Kommandos muss ein Tabulator sein.

Listing 1 Beispiel für ein einfaches Makefile

```
1 edit : main.o kbd.o files.o
2     cc -o edit main.o kbd.o files.o
3
4 main.o : main.c defs.h
5     cc -c main.c
6 kbd.o : kbd.c defs.h command.h
7     cc -c kbd.c
8 files.o : files.c defs.h buffer.h command.h
9     cc -c files.c
10
11 clean:
12     rm -f main.o kbd.o files.o
```

Listing 1 zeigt ein einfaches Beispiel für ein Makefile. Startet man MAKE, so wird das erste target gesucht (in diesem Fall *edit*). *edit* benötigt die Dateien *main.o*, *kbd.o* und *files.o*. Da diese (davon wird ausgegangen) noch nicht existieren, sucht MAKE nach einem target namens *main.o*. MAKE findet dieses target und führt nun das Kommando *cc -c main.c* aus. Als Kompilat erhält man eine Datei namens *main.o*. Das gleiche gilt genauso für *kbd.o* und *files.o*. Erst jetzt wird *edit* kompiliert, da die dependencies erfüllt sind.

Wird MAKE das nächste Mal aufgerufen, so überprüft es alle dependencies, ob sie noch erfüllt sind. Ändert sich im obigen Beispiel eine der Dateien *main.c*, *kbd.c*, *files.c*, *defs.h*, *buffer.h* oder *command.h*, dann werden eine oder mehrere Dateien neu kompiliert. Ist zum Beispiel die Datei *command.h* seit dem letzten Aufruf von MAKE verändert worden, so werden *kbd.o* und *files.o* erneuert, nicht aber *main.o*, da *command.h* nicht in den dependencies steht.

4.5.3 Variablen

Im obigen Beispiel werden die object-Dateien `main.o`, `kbd.o` und `files.o` einmal in den dependencies und einmal im Compileraufruf genannt. Diese doppelte Nennung ist bei vielen object-Dateien fehleranfällig. Daher benutzt man am besten Variablen. Man könnte eine Variable namens `objects` definieren:

```
objects = main.o kbd.o files.o
```

und für das target `edit` Folgendes schreiben:

```
edit : $(objects)
    cc -o edit $(objects)
```

Außerdem lautet die Regel für `clean` nun besser so:

```
clean:
    rm -f $(objects)
```

Alle Variablen aus der Umgebung (Umgebungsvariablen → 6.9, Variables from the Environment) werden dem Makefile bekannt gemacht, bzw. überschreiben die Variablen im Makefile. Dieses kann mit der `override` Anweisung (→ 9.5, The ‘override’ Directive) verhindert werden.

Listing 2 Verbessertes Makefile

```
1 objects = main.o kbd.o files.o
2 edit : $(objects)
3     cc -o edit $(objects)
4
5 main.o : main.c defs.h
6 kbd.o : kbd.c defs.h command.h
7 files.o : files.c defs.h buffer.h command.h
8
9 clean:
10     rm -f $(objects)
```

4.5.4 Implizite Regeln

Das Makefile kann aber noch wesentlich verbessert werden. `MAKE` weiß zum Beispiel, wie es aus `.c` Dateien `.o` Dateien erzeugt. Listing 2 enthält keine Regel, wie `MAKE` `main.o` erzeugen soll. Findet `MAKE` keine solche Regel, schaut es in seinen eingebauten Regeln nach. Diese Regeln werden auch manchmal implizite Regeln genannt. Die implizite Regel für `.o` Dateien lautet

```
$(CC) -c $(CPPFLAGS) $(CFLAGS) $< -o $@
```

Die Zeichen `$<` und `$@` bedeuten den Namen der ersten dependency bzw. den Namen des Targets. (→ 10.5.2, Automatic Variables). Die Variable `$(CC)` enthält normalerweise `cc` und `$(CPPFLAGS)` und `$(CFLAGS)` sind normalerweise leer. Für ein target `main.o` sähe die erzeugte Programmzeile folgendermaßen aus:

```
cc -c main.c -o main.o
```

Eine vollständige Liste der eingebauten Befehle steht in → 10.2, Catalogue of Implicit Rules. Findet MAKE keine Datei mit Namen **Makefile**, so versucht es anhand der eingebauten Regeln das gewünschte target zu erstellen. Eine Datei **helloworld.c** könnte (sofern ein simpler Compileraufruf genügt) mit **make helloworld** erzeugt werden, ohne dass MAKE ein Makefile braucht. Auch ein leeres Makefile würde funktionieren.

4.5.5 Bedingungen in Makefiles

MAKE⁴¹ kennt Bedingungen, mit denen sich das Verhalten von Makefiles beeinflussen lässt. Die Syntax für Bedingungen sieht folgendermaßen aus:

```
1 Bedingung
2   (Anweisungen wenn Bedingung wahr ist)
3 else
4   (Anweisungen wenn Bedingung falsch ist)
5 endif
```

wobei die Zeilen 3 und 4 auch ausgelassen werden können. Listing 3 ist ein

Listing 3 Beispiel für bedingte Makefiles

```
1 libs_for_gcc = -lgnu
2 normal_libs =
3
4 ifeq ($(CC),gcc)
5     libs=$(libs_for_gcc)
6 else
7     libs=$(normal_libs)
8 endif
9
10 foo: $(objects)
11     $(CC) -o foo $(objects) $(libs)
```

Beispiel für Bedingungen im Makefile (→ 7.2, Syntax of Conditionals).

4.5.6 Unterverzeichnisse

Viele größere Projekte werden in Unterverzeichnisse aufgeteilt. MAKE stellt verschiedene Mechanismen zur Verfügung, um mit Unterverzeichnissen umzugehen.

MAKE kann andere Makefiles aus Unterverzeichnissen aufrufen. Dazu kann man die Syntax **\$(MAKE) -C Unterverzeichnis** benutzen. Das veranlasst MAKE dazu, das Verzeichnis zu wechseln und MAKE erneut aufzurufen. Dieses neue MAKE wird mit den gleichen Parametern aufgerufen wie das aufrufende MAKE (→ 5.6.3, How the „MAKE“ Variable Works).

⁴¹ Dieses gilt nur für GNU MAKE.

Es gibt auch die Möglichkeit, dass MAKE automatisch Unterverzeichnisse nach dependencies absucht. Dazu schreibt man in die Variable `VPATH` die Verzeichnisse hinein, die MAKE durchsuchen soll, durch Doppelpunkte getrennt. Dann sucht MAKE auch in diesen Verzeichnissen nach Quelldateien, stellt das Kompilat aber in das aktuelle Verzeichnis (\rightarrow 4.3 ‘`VPATH`’: Search Path for All Dependencies).

4.5.7 Weitere Funktionen

In Listing 1 auf Seite 99 wurde als target `clean`⁴² angegeben. Existiert eine Datei dieses Namens, so wird anhand der Datei festgestellt, ob das target aktuell ist oder nicht. Da das bei `clean` nicht gewünscht ist (`clean` soll jedes Mal ausgeführt werden, wenn es gefordert ist), muss man das target `clean` als *phony* deklarieren. Dazu schreibt man ein neues target:

```
.PHONY : clean
```

Jetzt wird eine Datei namens `clean` ignoriert.

Jedes Kommando, das MAKE ausführt, wird im Terminalfenster ausgegeben. Möchte man diese Ausgabe unterdrücken, so schreibt man ein `@` vor das Kommando (\rightarrow 5.1, Command Echoing).

Tritt bei einem Kommando ein Fehler auf, so unterbricht MAKE sein Schaffen. In manchen Fällen ist das aber nicht gewünscht. Möchte man sicherstellen, dass ein Verzeichnis existiert und legt es mit `mkdir` an, würde MAKE abbrechen, wenn es schon existiert. Um das zu verhindern schreibt man vor das Kommando (hier: `mkdir`) ein `-` (Bindestrich):

```
-mkdir Verzeichnis
```

4.5.8 Abschließendes Beispiel

Listing 4 auf der nächsten Seite zeigt ein konkretes Beispiel, das der Autor dieses Textes benutzt, um seinen \LaTeX Text zu übersetzen. Die Zeilennummern dienen nur zur Orientierung. Zeile 1 zeigt einen Kommentar. Kommentare werden mit einem `#` Doppelkreuz eingeleitet und können an einer beliebigen Stelle in einer Zeile stehen. Dieses Zeichen und alles weitere in der Zeile wird ignoriert. (\rightarrow 5.2, Writing the Commands in Rules). In Zeilen 4–13 werden Variablen gesetzt. Manche dieser Variablen werden innerhalb von anderen Variablen eingesetzt. MAKE ersetzt diese Variablen erst, wenn sie tatsächlich benutzt werden. So wäre es kein Problem, die Zeilen 7 und 8 zu vertauschen, obwohl `MAIN` erst später bekannt ist. Soll dieses Verhalten unterdrückt werden, so muss `:=` benutzt werden (\rightarrow 6.2, Setting Variables). In Zeilen 16–23 wird die Regel für `paper.dvi` definiert. `paper.dvi` hängt ab von den Dateien, die in der Variable `SCRS` stehen. Als Kommando wird erst die Zeile 17 an eine Subshell geleitet und dann Zeilen 18–23, wobei die Schrägstriche am Ende der Zeilen entfernt werden. Das

⁴² `clean` löscht in der Regel alle überflüssigen Dateien, die mit Hilfe des Makefiles erzeugt werden können.

Listing 4 Etwas komplexeres Makefile

```
1 # goals: dvi ps print clean
2
3
4 DVIPS = dvips
5 TEX = latex
6
7 MAIN = paper
8 DVI = $(MAIN).dvi
9 PS = $(MAIN).ps
10 SCRS = $(MAIN).tex $(MAIN).sty make.tex
11
12 REFWARN = 'Rerun to get cross-references'
13 LATEXMAX = 5
14
15
16 $(DVI) : $(SCRS)
17     $(TEX) $(MAIN)
18     @RUNS=$(LATEXMAX); \
19     while [ $$RUNS -gt 0 ] ; do \
20         if grep $(REFWARN) $(MAIN).log > /dev/null ; \
21         then $(TEX) $(MAIN).tex ; else break; fi; \
22         RUNS='expr $$RUNS - 1 ';\
23     done
24
25 ps : $(PS)
26
27 $(PS) : $(DVI)
28     $(DVIPS) $(DVI)
29
30 .PHONY : clean print
31
32 clean :
33     -rm -f $(PS) $(DVI) *.aux *.log
34
35 print : $(PS)
36     lpr $(PS)
```

@-Zeichen unterdrückt die Ausgabe dieser Zeile im Terminalfenster. In Zeile 19 wird der Shell eine Variable durchgereicht. Damit diese nicht von MAKE ersetzt wird, bevor die Shell die Variable sieht, werden zwei \$ Zeichen anstatt einem benutzt. Zeile 25 definiert ein target ps, welches von `paper.ps` abhängt. Dieses wird in Zeilen 27 und 28 erzeugt. Zeile 30 deklariert die targets clean und print als phony; das heißt, dass MAKE Dateien mit diesem Namen ignoriert bei der Betrachtung, ob das target aktuell ist oder nicht. Die letzte Besonderheit ist in Zeile 33 zu finden: hier steht ein - (Bindestrich) vor dem rm Kommando. Dadurch werden Fehler beim Ausführen des Kommandos ignoriert (→ 5.4, Errors in Commands).

4.6 Versionsmanagement

4.6.1 Allgemeines

Neben dem eingebauten Versionsmanagement, welches kommerzielle Entwicklungsumgebungen anbieten, gibt es verschiedene Werkzeuge, die nicht nur auf eine Entwicklungsumgebung oder auf eine Programmiersprache beschränkt sind. Hier sind besonders das Revision Control System (RCS) und Concurrent Versions System (CVS) zu erwähnen. Beide Programme dienen dazu, Revisionen einer Datei zu speichern, wiederzuholen, zu identifizieren und zusammenzufügen. Es lassen sich Unterschiede zwischen mehreren Versionen herausfinden und Informationen über die einzelnen Revisionen speichern. CVS ist ein Frontend für RCS. Daher sind beide Programme in ihrer Funktionsweise sehr ähnlich. Während RCS immer auf einer einzelnen Datei arbeitet, behandelt CVS auch ganze Dateibäume. Weiterhin muss bei RCS jede Datei, die bearbeitet werden soll, mit einem *lock* versehen sein. Das heißt, es kann nur ein Benutzer an einer Datei arbeiten (schreibend) während alle anderen Benutzer nur Leserecht an dieser Datei haben. In manchen Fällen ist dieses zwar erwünscht; in anderen Fällen führt das aber auch zu Komplikationen. CVS bietet hier verschiedene Möglichkeiten an, diese Probleme zu behandeln. Im Weiteren gehe ich nur noch auf CVS ein, da es bei einem Projekt von mehreren Leuten vermutlich die bessere Wahl ist.

4.6.2 Verfügbarkeit

Im Fachbereich Informatik der Uni Dortmund liegt CVS im Modul `extra/cvs`. Informationen zu CVS erhält man mit `info cvs`. Im weiteren Verlauf dieses Dokuments verweise ich auf diese info-Anleitung (→).

4.6.3 Einrichten von CVS

CVS besteht aus einem zentralen Verzeichnisbaum und mehreren dezentralen. Der zentrale Verzeichnisbaum wird auch als *repository* bezeichnet. Die dezentralen Verzeichnisbäume liegen in der Regel bei den Entwicklern und werden *Arbeitsverzeichnis* oder auch *working directory* genannt. Die Versionsverwaltung kommuniziert immer zwischen dem Arbeitsverzeichnis und dem repository. In

diesem repository werden die Dateien gespeichert, die die Informationen über die verschiedenen Versionen der Projekte beinhalten. Auf das repository wird nur über das cvs Kommando zugegriffen. Um cvs einzurichten, muss zuerst ein Verzeichnis, cvsroot, erstellt werden. Das wird hinterher als *cvsroot* bezeichnet und sollte, der besseren Erkennung wegen, auch den Namen cvsroot erhalten. Da auf das Verzeichnis später nicht mehr direkt zugegriffen wird, spielt dieser Name an sich keine Rolle.

Als Erstes wird `cvs -d Verzeichnis init` aufgerufen. Das Verzeichnis gibt die Stelle von cvsroot an. Anstelle von `-d Verzeichnis` kann auch die Umgebungsvariable `CVSROOT` gesetzt werden (das gilt für alle cvs Kommandos). Wichtig: alle cvs Kommandos brauchen die Verzeichnisangabe. Daher ist es sinnvoll, `CVSROOT` zu setzen. Liegt das Verzeichnis cvsroot unter `/home/pg348/share/dev/cvsroot`, so setzt man `CVSROOT` auf `/home/pg348/share/dev/cvsroot` (→ 2.6, Specifying a repository). Nun kann mit `cvs import` (→ 3.1, Creating a directory tree from a number of files) ein Verzeichnis in die Obhut von cvs gegeben werden. Beispiel: es soll ein einfaches Programm in der Programmiersprache C von mehreren Benutzern erstellt werden. Dazu ruft ein Benutzer

```
cvs import -m "Einfaches Programm" display prog start
```

auf. Hier bedeutet `-m "Einfaches Programm"` eine Bezeichnung für das Projekt, `display` der Name des Verzeichnisses, das später bei `cvs checkout` erzeugt werden soll, `prog` der sogenannte *vendor tag* und `start` der sogenannte *start tag*. Die beiden letzten Angaben haben nur geringe Bedeutung (→ 13, Tracking third-party sources), müssen aber dennoch angegeben werden. Die beiden Tagnamen können beliebig gewählt werden. Dieser Befehl erzeugt im repository ein Verzeichnis `display`, in das alle Dateien des aktuellen Verzeichnisses sowie alle darunter liegenden Verzeichnisse und Dateien kopiert werden. Als nächstes könnte noch ein Modul eingetragen werden (→ 3.2, Defining the module). Wird das unterlassen, so wird als Modulname der Name des Verzeichnisses (hier: `display`) angenommen. Einzelne Projekte identifiziert man immer über den Modulnamen.

Es besteht auch die Möglichkeit, bestehende Verzeichnisstrukturen mit Dateien in ein cvs repository einzufügen. Man könnte beispielsweise das vorhandene Literaturverzeichnis `/home/pg348/share/literatur` unter cvs Verwaltung stellen, indem man in das Verzeichnis wechselt und

```
cvs import -m "Literatur" literatur lit start
```

ausführt. Jetzt sollte das Verzeichnis sicherheitshalber gelöscht werden und mittels `cvs checkout literatur` (im share Verzeichnis) unter die cvs Verwaltung gestellt werden. (Das Verzeichnis wird dann wieder erstellt, hat aber noch zusätzliche Dateien für cvs.) Das neu erzeugte Verzeichnis kann natürlich auch an jeder anderen Stelle wieder angelegt werden. So könnte jeder in der Projektgruppe sein eigenes Literaturverzeichnis erstellen, `~/work/literatur` zum Beispiel, aber auf den gemeinsamen Datenbestand zurückgreifen, der in dem repository liegt.

4.6.4 Bedienung

Wird cvs zum ersten Mal benutzt, so ruft man `cvs checkout Modulname` auf. Im Beispiel wäre das `cvs checkout display`. Nun generiert cvs (im aktuellen Verzeichnis) ein Verzeichnis mit dem Namen, der bei dem `cvs import` Befehl angegeben wurde und kopiert alle Dateien aus dem repository hinein. Den `checkout` Befehl benutzt man ab diesem Zeitpunkt nur noch, um ganz spezielle Revisionen zu erhalten.

Dieses erzeugte Verzeichnis ist das Arbeitsverzeichnis. Neue Textdateien können mit einem Editor erzeugt und mit `cvs add Dateiname` dem repository hinzugefügt werden. Existiert schon ein Arbeitsverzeichnis, benutzt man den Befehl `cvs update`, um den neusten Stand der Quellen zu erhalten. Jetzt können die Quellen verändert werden. Mehrere Benutzer können gleichzeitig dieselbe Datei per `cvs update` holen und verändern. Um dem repository die Änderungen zu übermitteln, führt man zuerst `cvs update` und dann `cvs commit` aus. Bei dem `update` versucht cvs die Änderungen, die lokal gemacht wurden, und die Version im repository zusammenzufügen. Gelingt dieses, kann das `commit` ausgeführt werden. Wurde die Datei zwischenzeitlich von einem anderen Benutzer verändert, *und* die Zeilen, die der andere Benutzer verändert hat, überschneiden sich mit den Zeilen, die lokal geändert wurden, dann muss der Quelltext per Hand korrigiert werden. Dazu markiert cvs die sich überschneidenden Zeilen mit den Zeichen <<<<<< zu Beginn der Überlappung und >>>>>> am Ende. Erst wenn diese Stellen korrigiert wurden, kann die neue Version mittels `cvs commit` in das repository übertragen werden.

In dem C-Programm-Beispiel existieren folgende drei Zeilen:

```
zahl[zahlPos++] = brettString[++stringPos];
if (brettString[stringPos + 1] >= '0'
    && brettString[stringPos + 1] <= '9')
    zahl[zahlPos++] = brettString[++stringPos];
```

Benutzer Andreas fügt nun die Zeile

```
zahl[zahlPos] = '\0';
```

an das Ende des Programmes an. Währenddessen meint Benutzer Bodo, dass es aber besser

```
anzahlLeer = str2int(zahl);
```

heißen soll. Jetzt ruft Benutzer Andreas `cvs update` und danach `cvs commit` auf, um seine Version dem repository bekannt zu machen. Danach möchte Benutzer Bodo das Gleiche machen und ruft `cvs update` auf. Jetzt beschwert sich cvs bei Bodo, dass ein Konflikt aufgetreten sei:

```
cvs update: Updating .
RCS file: /home/pg348/work/cvsroot/display/main.c,v
retrieving revision 1.6
retrieving revision 1.7
```

```

Merging differences between 1.6 and 1.7 into main.c
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in main.c
C main.c

```

Jetzt ist Bodo aufgefordert, den Konflikt in der Datei `main.c` manuell zu ändern. Die Datei sieht folgendermaßen aus:

```

zahl[zahlPos++] = brettString[++stringPos];
if (brettString[stringPos + 1] >= '0'
    && brettString[stringPos + 1] <= '9')
<<<<<<< main.c
zahl[zahlPos++] = brettString[++stringPos];
=====
anzahlLeer = str2int(zahl);
>>>>>>> 1.7

```

Der Konflikt ist leicht erkennbar und es sollte keine Schwierigkeiten machen, ihn zu lösen. Jetzt kann `cvs commit` aufgerufen werden.

4.6.5 Information im Quelltext

Oft ist es erwünscht, im Quelltext Informationen über die Revisionen zu haben. Hierfür gibt es den Mechanismus *keyword substitution*. Fügt man in den Quelltext die Zeichen `Id` ein, so ändert cvs diese Zeichen in `$Id:` Quelldatei, Versionsnummer, Datum/Uhrzeit und Benutzername `$`, wobei diese Wörter durch entsprechende Information ersetzt werden. Ein Beispiel für das C Programm könnte so aussehen:

```
$Id: main.c,v 1.8 1999/08/29 13:54:53 pg Exp $
```

Weitere Schlüsselwörter, die cvs ersetzt, sind zum Beispiel `Log`, `$Author$` oder `$Revision$` (→ 12.1, RCS Keywords, → 12.4, Problems with the `Log` keyword). `Log` wird ersetzt durch den Text, der bei `cvs commit` als Revisionsinformation angegeben wird. Dadurch, dass sich dieser Text oft über mehrere Zeilen erstreckt und durch mehrere Revisionen öfters vorkommt, stellt man `Log` in der Regel an das Ende der Quelldatei.

4.6.6 Tags

Jedes Mal, wenn eine Datei verändert und `cvs commit` aufgerufen wird, erhöht cvs die Revisionsnummer um eine Ziffer (hinter dem Dezimalpunkt). Es besteht die Möglichkeit, die Revisionsnummer nach oben hin zu verändern. Die Anleitung von cvs schlägt vor (→ 4, Versions, revisions and releases), für Versionsnummern des gesamten Projekts („erste Betaversion“, „fertige Fassung“, „Version 1.0“) *tags* zu benutzen. Durch das rekursive Vorgehen von cvs ist es leicht, allen aktuellen Versionen denselben symbolischen Namen (=tag) zuzuordnen. Hierfür wird das Kommando `cvs tag -r tagname` verwendet. Möchte man sich von allen Dateien die Revisionen holen, die dazu benutzt worden sind eine

bestimmte Version zu erhalten, macht man das mit dem Kommando `cvs checkout -r tagname`. Die aktuellen Revisionen erhält man mit `cvs update -A`.

4.7 Oracle

Unsere Überlegungen, welche Datenbank wir am besten für unser Projekt verwenden, führten dazu, Oracle als DBMS (Database Management System) zu benutzen. Zu Oracle haben wir eine Hochschullizenz, so dass wir eine Version davon von der IRB zur Verfügung gestellt bekommen haben.

Die Oracle-Datenbank zeichnet sich vor allem dadurch aus, dass sie einfach zu bedienen bzw. zu erlernen ist. Die mitgelieferten Dokumentationen sind sehr gut und einfach geschrieben, so dass sie leicht verständlich sind. Man kann außerdem mit Oracle sehr effizient arbeiten, da sie kurze Zugriffszeiten auf die gespeicherten Daten ermöglicht. Außerdem kann man an Speicherplatz sparen, indem Tabellen in sog. Tablespace gepackt werden.

4.7.1 Bedienung von Oracle

Oracle ist ein relationales DBMS. D. h. es basiert auf der Verwaltung von Tabellen, die Informationen bzw. Datensätze enthalten. Die Datensätze einer Tabelle gehören zu einer Entität, also zu etwas, das existiert, und haben Beziehungen zueinander, stehen also in Relationen. Genauso verhalten sich auch die Daten aller von einer Datenbank (abgekürzt DB) verwalteten Tabellen, sie stehen also auch in Relationen. Oracle kann via SQL (Structured Query Language) angesprochen werden. Dies ist eine relationale DB-Anfragesprache, mit Hilfe derer man Daten in die DB einfügt, vorhandene manipuliert oder löscht. Im Folgenden werden die wichtigsten SQL-Befehle kurz erläutert.

4.7.1.1 Starten des Oracle-Abfragetools

Um auf Oracle zugreifen und SQL-Anweisungen eingeben zu können, muss man erst eine Verbindung zu Oracle herstellen. Dazu steht dem Benutzer ein Abfragetool zur Verfügung, nämlich `SQLPLUS`. Dieses Tool wird mittels des Aufrufs `sqlplus` gestartet. Nach Eingabe dieses Befehls wird man aufgefordert, einen Benutzernamen und ein Passwort einzugeben. Sind diese richtig, so hat man die Verbindung zu Oracle erfolgreich hergestellt. Dann kann man SQL-Anweisungen interaktiv, also direkt am Bildschirm, eingeben. Der interaktive Zugriff auf Oracle erfolgt meist, wenn man nur Daten definieren will bzw. wenn einfache SQL-Anweisungen durchgeführt werden sollen. Für weitere Details wird auf die Oracle-Dokumentation [ORA99] verwiesen.

4.7.2 SQL

Die hier behandelten SQL-Anweisungen können jedem in SQL oder Datenbanken einführenden Buch entnommen werden. Hier wird auf [PET90] verwiesen.

- **create database**

Mit diesem Befehl erzeugt man eine DB unter Oracle, die dann Datentabellen enthält. Der DB muss beim Erzeugen ein Name gegeben werden.

Syntax:

```
create database db_name;
```

- **create table**

Mit diesem Befehl erzeugt man eine Tabelle. Eine Tabelle besteht aus Spalten und Reihen. Die Spalten enthalten die Attribute, also die Eigenschaften, die zu einer Entität (Datum) gehören. Die Reihen enthalten dann die Daten.

Syntax:

```
create table table_name [Spalte x Datentyp] [Spalte y  
Datentyp];
```

- **alter table**

Mit diesem Befehl lässt sich eine Spalte in eine vorhandene Tabelle einfügen oder eine bereits vorhandene modifizieren.

Syntax:

```
alter table tabel_name  
add Spalten_name Datentyp; modify Spalten_name Datentyp;
```

- **drop table**

Löscht eine Tabelle vollständig aus der DB.

Syntax:

```
drop table_name;
```

- **drop database**

Mit diesem Befehl löscht man eine Datenbank.

Nun wollen wir Daten Speichern und gespeicherte ändern. Das macht man anhand folgender SQL-Anweisungen:

- **insert into table**

Dieser Befehl fügt eine neue Reihe (also ein neues Datum) in eine Tabelle ein. Entweder ist die Reihe ganz neu definiert oder sie stammt aus einer anderen Tabelle. Bei diesem Befehl muss man auch die Spaltennamen angeben, in die die neuen Werte gespeichert werden sollen, falls nicht alle Spalten mit Daten belegt werden. Bei Nichtangabe werden alle Spalten angesprochen. Der Typ des zu speichernden Wertes muss mit dem der zugehörigen Spalte übereinstimmen.

Syntax:

1. `insert into table_name Values;` für das Hinzufügen neuer Daten
2. `insert into table1_name Values form tabel2_name;` für das Hinzufügen der Daten einer Tabelle in eine andere

- **update**

Dieser Befehl ermöglicht das Ändern von Tabelleninhalten.

Syntax:

```
update table_name set Spalten_name1 = wert1... where
Bedingung;
```

- **delete**

Mit diesem Befehl kann man Daten aus Tabellen löschen:

Syntax:

```
delete from table_name where Bedingung;
```

4.7.3 SQL-Anfragen

Mit Hilfe von SQL-Anfragen kann man auf die Daten einer Datenbank zugreifen. Dazu muss man authorisiert sein. Die einfachste Form einer SQL-Anfrage lautet:

- `select * from table_name;`

Damit kann man alle Daten einer Tabelle abrufen. Bei der Suche nach bestimmten Daten muss man der Anfrage die Where-Klausel anfügen.

- `select data_name from table_name where Bedingung;`

Dabei werden nur Daten herausgefiltert, für die die Bedingung erfüllt ist. Die Bedingung kann einfach oder zusammengesetzt sein. Sie ist einfach, wenn nur eine einzige Spalte der Tabelle angesprochen wird. Zusammengesetzt bzw. verknüpft muss die Bedingung sein, wenn man mehrere Spalten abfragt.

4.7.3.1 Beispiel

Als Beispiel wird unsere DB genommen. Die DB war für uns schon angelegt, sodass wir nur noch unsere Tabellen definieren mussten. U.a. haben wir definiert:

```
Create table brett_kodierung (brett_id number(7) primary key,
  brett_str varchar(64) null unique, endstellung integer null
) Tablespace chessgp ;
```

Diese Tabelle enthält alle von der PG entworfenen Schachbretter. Ein Schachbrett besteht aus einer ID und einem String. Die BrettID soll eindeutig sein und der String beschreibt die Steinstellung auf dem Brett. Insbesondere werden Schachbretter, die eine Endstellung darstellen, gekennzeichnet.


```
Create table bewertung (bew_id number(2) primary key,
    bew_name varchar(20) null) Tablespace chessgp;}
```

Diese Tabelle enthält die Bewertungen aller betrachteten Schachstellungen (statische Bewertung) und aller vorgenommenen Züge (Dynamische Bewertung).

- Insert-Anweisung

```
Insert into bewertung values(1, 'Material');}
Insert into bewertung values(10, 'Tauschzuege');}
    (dynamische Bewertung)
```

- Select-Anweisung:

```
Select brett_id from brett_kodierung
where endstellung = 0 and brett_id > 20000;
```

Alle Bretter die *keine* Endstellung darstellen und deren ID größer als 20 000 ist, werden hier herausgefiltert.

```
drop table brett_kodierung;
```

Dieser Befehl löscht die Tabelle „brett_kodierung“ aus der DB.

4.7.4 Embedded SQL

Bisher wurde erklärt, wie SQL-Anweisungen interaktiv benutzt werden. Eine andere Möglichkeit besteht in der Benutzung der Anweisungen innerhalb eines Anwendungsprogramms. SQL-Anweisungen werden dabei im Quellcode eingebettet. Anhand eines Precompilers wird dieser Code dann übersetzt und ein C- bzw. C++-Code daraus generiert. Die Benutzung der eingebetteten Anweisungen empfiehlt sich zur Formulierung zum einen komplexer Abfragen oder Änderungen einer DB und zum anderen wiederholter Anweisungen, z. B. wenn die „Insert“-Anweisung mehrmals durchgeführt wird. Eingebettete SQL-Anweisungen ähneln den interaktiven jedoch mit dem Unterschied, dass sie zusätzliche Schlüsselwörter enthalten, die in interaktiven Anweisungen nicht vorkommen. Demnach kann jede interaktive SQL-Anweisung auch eine eingebettete sein.

4.7.4.1 Kennzeichnen der eingebetteten SQL-Anweisungen

Damit der Precompiler eine eingebettete SQL-Anweisung erkennt, muss diese gekennzeichnet sein. Dazu wird ihr als Präfix das Schlüsselwort **EXEC SQL** vorangestellt. Das Ende einer SQL-Anweisung wird mit **END-EXEC** oder mit einem Semicolon „;“ angedeutet.

4.7.4.2 Host- und Indikator-Variablen

Der Datenaustausch zwischen dem Anwendungsprogramm (Quellcode) und Oracle wird mit Hilfe von Variablen ermöglicht. Die Variablen befinden sich innerhalb der eingebetteten SQL-Anweisungen und sind nichts anderes als Variablen der Hostsprache, die auch als solche definiert werden müssen. Damit sie erkennbar sind, wird den Variablen jeweils ein „:“ vorangestellt. Es gibt zwei Arten von Variablen, die Host- und die Indikator-Variablen. Host-Variablen dienen als Zielvariablen einer Abfrage, bzw. können anstelle einer Konstanten in den Datenmanipulationsanweisungen verwendet werden. Sie können beispielsweise in folgenden Klauseln erscheinen:

- into-Klausel einer Select-Anweisung,
- Where-Klausel
- und values-Klausel der insert-Anweisung.

Host-Variablen müssen dieselben Datentypen haben wie die Datenwerte in der DB, die sie repräsentieren. Indikator-Variablen sind spezielle Variablen, die nur im Zusammenhang mit Host-Variablen verwendet werden können. Jede Host-Variable kann somit eine zugehörige Indikator-Variable haben, die zusätzliche Information über den gespeicherten Wert in der Host-Variable liefert, z. B. ob der Host-Variable ein NULL-Wert zugewiesen wurde. Die Host- und die dazu gehörige Indikator-Variable werden im Anwendungsprogramm folgendermaßen geschrieben:

```
:hostvar:indvar
```

Host- und Indikator-Variablen müssen im Anwendungsprogramm explizit definiert werden. Dies geschieht in der sog. **Declare Section** des Anwendungsprogramms:

```
EXEC SQL BEGIN DECLARE SECTION;  
    Var Deklarationen  
EXEC SQL END DECLARE SECTION;
```

In C sowie C++ kann die letzte Zeile des Declare Sections durch ein Semicolon ersetzt werden.

4.7.4.3 Fehlerbehandlung

Nach der Abarbeitung jeder ausführbaren SQL-Anweisung wird die Information, ob diese Anweisung erfolgreich ausgeführt wurde, an eine spezielle Struktur geliefert. Diese Struktur heißt **SQLCA** (SQL Communication Area) und muss in jedem Host-Programm explizit eingefügt werden, damit die in ihr gespeicherte Information benutzt werden kann. Das Einfügen der SQLCA-Struktur erfolgt mittels folgender SQL-Anweisung:

```
EXEC SQL  
INCLUDE SQLCA
```

Als Alternative zur expliziten Prüfung der SQL-Anweisungen mit Hilfe der SQLCA-Struktur bietet sich die Benutzung der WHENEVER-Anweisung an. Die allgemeine Form von WHENEVER ist: `WHENEVER bedingung massnahme`

4.7.4.4 Verwendung eines Cursors

Ein Datenbankcursor lässt sich mit dem Cursor einer Textverarbeitung vergleichen, der die aktuelle Position auf dem Bildschirm anzeigt. Wenn man die Pfeiltasten nach oben oder unten betätigt, bewegt sich der Cursor um eine Zeile nach oben bzw. unten. Andere Tasten wie (Page Up) oder (Page down) bewirken einen Sprung von mehreren Zeilen in die entsprechende Richtung. In der gleichen Weise funktionieren Datenbankcursor. Mit einem Datenbankcursor wählt man eine Gruppe von Daten aus, scrollt durch die Gruppe der Datensätze (auch Recordset genannt) und untersucht die einzelne Datenzeile, auf die der Cursor zeigt. Ein anderer häufiger Einsatzfall der Cursor ist die Speicherung von Abfrageergebnissen für spätere Verwendung. Die Ergebnismenge eines Cursors wird aus der Ergebnismenge einer Select-Abfrage gebildet. Wenn eine Anwendung oder Prozedur wiederholt auf eine Gruppe von Datensätzen zugreifen muss, geht es schneller, einen Cursor einmalig zu erzeugen und ihn mehrmals wiederzuverwenden als wiederholt die Abfrage der Datenbank auszuführen. Darüberhinaus hat man den Vorteil, dass man mit einem Cursor durch die Ergebnismenge der Abfrage schneller scrollen kann. Zunächst muss ein Cursor deklariert werden. Dies geschieht folgendermassen:

```
EXEC SQL DECLARE cursorname CURSOR FOR SELECT-Anw.
```

Für die Tabelle „brettkodierung“ in unserer DB sieht somit ein Cursor so aus:

```
EXEC SQL
  DECLARE crs_brett CURSOR FOR
  SELECT brett_id, brett_str, endstellung
  FROM brettkodierung
  WHERE endstellung = 0;
```

- **OPEN cursorname**

Mit dieser Anweisung wird die Select-Anweisung ausgeführt, die sich innerhalb der DECLARE CURSOR-Anweisung befindet. Das Ergebnis solcher Select-Anweisung wird Treffermenge genannt. Nach Ausführung der Select-Anweisung befindet sich der Cursor vor der ersten Reihe der Treffermenge.

- **FETCH cursorname INTO :host_var**

Mit dieser Anweisung wird der Cursor auf die nächste Reihe der Treffermenge positioniert. Nach der ersten FETCH-Anweisung wird der Cursor auf die erste Reihe gesetzt, nach der zweiten auf die zweite usw. Alle Reihen der Treffermenge können nur sequentiell abgearbeitet werden. Die FETCH-Anweisung, die ausgeführt wird, nachdem der Cursor schon auf die letzte Reihe positioniert war, weist der Host-Variablen (bei Oracle) den Wert 1403 zu. Für weitere Details siehe auch [PET90]

- CLOSE cursorname

Mit dieser Anweisung wird ein Cursor geschlossen.

4.7.4.5 Beispiel aus der PG-DB

Anhand dieses Beispiels werden die eingebetteten SQL-Anweisungen demonstriert. Das Beispiel enthält Ausschnitte aus dem Programm zur statischen Bewertung. (In diesem Beispiel wird kein Cursor verwendet)

```

1  #include-Direktiven
2
3  #define USERNAME    /* Username sowie Passwd werden definiert, */
4  #define PASSWORD    /* um auf Oracle zugreifen zu k"onnen */
5  #define host "birke" /* Name des Rechners,
6                        auf dem Oracle installiert ist */
7
8  /* Deklarationsabschnitt
9   hier werden auch die Host-Variablen definiert */
10
11
12  EXEC SQL BEGIN DECLARE SECTION;
13
14  char *username = USERNAME;
15  char *password = PASSWORD;
16  char brettstr_db[65];
17  int brettidmax=0;
18  int brett_zaehler=0;
19  int bewertung = 0;
20  char *db_string = host;
21
22  EXEC SQL END DECLARE SECTION;
23  EXEC SQL INCLUDE sqlca;      /* SQLCA zur Fehlerbehandlung*/
24  void sql_error(char *msg);   /* Funktion zur expliziten Fehlerbeh. */
25
26  char brettstr[65];
27
28  /* Objekt der Klasse BewStatisch anlegen */
29
30  FDB_BewStatisch zurBewertung;
31
32  int main() {
33
34  EXEC SQL WHENEVER SQLERROR DO sql_error("'Oracle error'");
35
36  /*Verbindung zu Oracle herstellen*/
37

```

```

38 EXEC SQL CONNECT :username IDENTIFIED BY :password USING :db_string;
39
40 /* Eine eingebettete Select-Anweisung */
41
42 EXEC SQL SELECT MAX(brett_id)
43 INTO :brettidmax
44 FROM brett_kodierung;
45
46 /*Eine eingebettete Insert-Anweisung*/
47 /* Material */
48
49 bewertung = zurBewertung.material(brettstr);
50 EXEC SQL INSERT INTO Bewertung_statisch
51 VALUES (:brett_zaehler, 1, :bewertung);
52
53 /*Verbindung zu Oracle unterbrechen*/
54
55 EXEC SQL COMMIT WORK RELEASE;
56
57 exit(0);
58 }

```

4.8 PVM

PVM (parallel virtual machine) ist ein Softwarepaket mit dessen Hilfe miteinander vernetzte Unix Rechner wie ein Parallelrechner eingesetzt werden können. Damit kann sich entsprechend vorbereitete Software die gesamte Rechenleistung eines Rechnerpools für parallel abzuarbeitende Berechnungen zu Nutze machen, ohne dass ein echter Parallelrechner vorhanden sein muss. Das Auslagern der Berechnungen geschieht zum grössten Teil, ohne dass der Benutzer eingreift. Im folgenden wird mit PVM das Softwarepaket bezeichnet und mit VM (virtuelle Maschine) die logische Struktur des Rechnernetzes.

4.8.1 Verfügbarkeit und Dokumentation

Im Fachbereich Informatik der Universität Dortmund liegt PVM im Modul `extra/pvm` vor. Informationen zu PVM sind in den manpages enthalten. Zusätzlich existiert eine Online-Version des offiziellen PVM Buchs unter <http://www.netlib.org/pvm3/book/pvm-book.html>; weiterführende Information finden sich unter <http://www.epm.ornl.gov/pvm/>.

4.8.2 Kommunikation

Die Kommunikation mit PVM geschieht auf zwei verschiedenen Ebenen. Auf der oberen Ebene tauschen die Benutzerprogramme untereinander Nachrichten aus, während die untere Ebene die Kommunikation zwischen den pvm-Dämonen regelt.

Die Kommunikation auf der Benutzerebene geschieht in einer abstrakten Weise, so dass der Benutzer sich nicht darum kümmern muss, ob der Kommunikationspartner auf dem lokalen Rechner läuft oder auf einem entfernten. Auf dieser Ebene stellt PVM verschiedene Hilfsmittel zur Verfügung um einfache Datenstrukturen zwischen zwei Prozessen auszutauschen. Nachrichten zwischen Prozessen werden in einfache Datenstrukturen kodiert und können mit unterschiedlichen Tags versehen werden, so dass der Empfänger bestimmte Nachrichten herausfiltern kann. Nachrichten werden in einer Warteschlange gespeichert. PVM garantiert, dass die Reihenfolge der gesendeten Nachrichten beim Empfänger erhalten bleibt. PVM unterstützt unicast-, multicast- und in speziellen Fällen (Prozessgruppen) auch broadcast-Nachrichten.

Die Kommunikation auf der unteren Ebene funktioniert mittels der, durch das Betriebssystem zur Verfügung gestellten, IP (Internet Protocol) Varianten TCP und UDP. Kommuniziert wird immer zwischen pvm-Dämonen. Auf jedem Rechner der PVM wird ein pvm-Dämon (`pvm`) gestartet. Einer dieser pvm-Dämon Prozesse ist der Master-pvm-Dämon. Sobald dieser nicht mehr existiert, wird die gesamte VM angehalten. Bei einem Ausfall eines der anderen pvm-Dämonen wird nur dieser Teil der VM berührt.

4.8.3 Benutzung von PVM

Um die VM zu konfigurieren und zu administrieren existiert ein (Consolen-) Programm `pvm`.

Mit dem Consolenprogramm `pvm` können Rechner in die VM mit einbezogen und wieder herausgenommen werden. Als Argument für `pvm` sollte die Hostdatei (siehe Abschnitt 4.8.4 auf der nächsten Seite über die Hostdatei) mit angegeben werden. Läuft noch kein `pvm` auf dem Rechner, auf dem das Kommando `pvm` aufgerufen wurde, so wird lokal der Master-pvm-Dämon gestartet. Auf den anderen Rechnern, die in der Hostdatei aufgelistet sind, wird ebenfalls ein `pvm` gestartet. Dieses geschieht mittels `remote shell`.⁴³ Sobald `pvm` gestartet wurde steht dem Benutzer eine Eingabezeile zur Verfügung. Die wichtigsten der hier möglichen Kommandos sollen im folgenden erklärt werden.

conf Es wird eine Übersicht über die vorhandenen Rechner in der VM gegeben. Alle Rechner, die hier aufgelistet sind, können Prozesse in der VM ausführen.

add Fügt einen Rechner, der als Argument übergeben wird, zur VM hinzu. Ist dieser Rechner in der Hostdatei eingetragen, so werden die dort gültigen Optionen beachtet. Ansonsten werden die zuletzt gültigen globalen Optionen benutzt. Es wird angezeigt, ob der Rechner erfolgreich in die VM hinzugefügt wurde.

⁴³ Man kann zum starten von Kommandos auch `secure shell` oder andere Authentifizierungsmethoden benutzen. Diese müssen aber bei der Kompilierung des `pvm` angegeben werden.

```

1 # dx: wo liegt der pvmd? (der macht nichts anderes, als der
2 # orig. pvmd nur noch zusaetzlich nice 10)
3 # ep: wo liegen die exe Dateien?
4 * dx=/home/pg348/share/bin/pvmd ep=$HOME/bin:/home/pg348/share/bin
5
6 sven
7 eiche

```

Abbildung 10: Einfache Hostdatei

delete Entfernt einen Rechner aus der VM. Es wird angezeigt, ob der Rechner erfolgreich aus der VM entfernt wurde.

4.8.4 Die Hostdatei

Jeder Benutzer kann sich durch die Hostdatei seine eigene Virtuelle Maschine konfigurieren. In der einfachsten Form besteht die Hostdatei aus einer Liste von Hostnamen, die in die Virtuelle Maschine mit aufgenommen werden sollen. Jeder Hostname steht in einer eigenen Zeile, Leerzeilen und Zeilen, die mit **#** beginnen, werden ignoriert.

In dieser Datei (Beispiel in Abbildung 10) stehen die Namen der Rechner, die in die virtuelle Maschine mit einbezogen werden sollen. Im obigen Beispiel sind das die Rechner *sven* und *eiche*. Neben diesen Rechnernamen können auch Kommentare (Zeilen 1–3), Leerzeilen und Optionenzeilen (Zeile 4) enthalten sein. Optionenzeilen fangen mit einem ***** an und gelten bis zur nächsten Optionenzeile.

Der Eintrag **dx** gibt den Ort des **pvmd** an, der auf den Rechnern gestartet werden soll. Ist dieser nicht angegeben, wird ein Standard **pvmd** gestartet. Einen eigenen **pvmd** kann man beispielsweise dazu benutzen, um die aufgerufenen client-Prozesse mit einem bestimmten *nice*-Wert zu starten.

Der Eintrag **ep** gibt Verzeichnisse an, unter denen die ausführbaren Programme abgelegt sind, die durch einen **pvmd** gestartet werden sollen. Ist diese Option nicht angegeben, wird unter **\$HOME/pvm3/bin/\$PVM_ARCH/** gesucht; in unserem Fall, auf den Solaris Rechnern, ist **\$PVM_ARCH** das Verzeichnis **SUN4SOL2**. Der Rechner, von dem aus die VM gestartet wird, sollte auch in der Hostdatei stehen, da ansonsten die **ep**-Option ignoriert wird.

4.8.5 Beispiel

In den Listings 5 auf Seite 119 und 6 auf Seite 120 ist eine einfache verteilte Anwendung dargestellt, die mit Hilfe von PVM auf verschiedene Knoten der VM ausgelagert wird und dort eine einfache Berechnung ausführt. Dieses Beispiel enthält bis auf Fehlerabfragen alle notwendigen Mechanismen, die auch für eine komplexe Anwendung gebraucht werden. Eine PVM-Anwendung besteht in der Regel aus einem „Hauptprogramm“ sowie einem „Unterprogramm“, welches die gewünschten Funktionen berechnet. Das Unterprogramm wird auf der VM ver-

teilt. Listing 5 stellt das Hauptprogramm dar (ohne Berechnung), Listing 6 das Unterprogramm.

Die beiden Programme sind im Sinne des Betriebssystems eigenständige, ausführbare Dateien. Im Hauptprogramm muss angegeben werden, wie das Unterprogramm heißt. PVM sucht in den vorgegebenen Pfaden nach diesem Programm (s. Kapitel 4.8.4 auf der vorherigen Seite). Das Unterprogramm wird mit der Methode `pvm_spawn` aufgerufen. PVM startet das angegebene Programm entweder auf einem zufällig ausgewählten Knoten (Rechner) der VM, auf einer bestimmten Gruppe von Knoten (dieselbe Architektur) oder auf einem bestimmten Knoten. Ab diesem Zeitpunkt kommunizieren die einzelnen Prozesse über Datenstrukturen, die mit `pvm_init`send vorbereitet werden, mit `pvm_pk...` in den internen Puffer geschrieben und mit `pvm_send` an einen entfernten Prozess geschickt werden (unicast).

Im Listing 5, dem Hauptprogramm, werden in den Zeilen 11–22 die Berechnungen auf der VM gestartet. Dazu wird in Zeile 12 ein neuer Prozess erzeugt. Jeder Prozess in der VM erhält eine eindeutige Nummer (`tid`). Diese wird bei einem `pvm_spawn` erzeugt und ist für die Kommunikation notwendig, da sie die Empfänger der Nachrichten bestimmt. In Zeile 16 wird die Anfrage in einen Puffer geschrieben und in Zeile 18 dem entfernten Prozess gesendet.

In den Zeilen 25–30 empfängt das Programm Nachrichten von den ausgelagerten Prozessen und gibt diese auf den Bildschirm aus. Dazu wird in Zeile 26 ein blockierendes `pvm_recv` aufgerufen, das auf Nachricht von einem beliebigen Prozess wartet. In Zeile 27 werden Informationen über die erhaltene Nachricht eingeholt; in Zeile 28 wird diese Nachricht in eine Datenstruktur (`char *`) gewandelt. PVM erlaubt blockierendes, nicht-blockierendes und zeitgesteuertes Warten auf eingehende Nachrichten (`pvm_recv`, `pvm_probe` und `pvm_trecv`).

Das Unterprogramm (Listing 6 auf Seite 120) wird von einem `pvmd` Prozess aufgerufen. Als erstes (Zeile 10) wird die `tid` des aufrufenden `pvm_spawn` ermittelt. Dann wird blockierend auf eine Nachricht von diesem Prozess gewartet und in ein `int` umgewandelt (Zeile 11 und 12). In den Zeilen 19 und 20 wird ein PVM Puffer vorbereitet und das in den Zeilen 14–17 erzeugte `char`-Array in ein PVM Puffer gepackt. Der `pvm_send`-Aufruf schickt dem aufrufenden Prozess diesen Puffer.

4.9 xboard

Als graphisches Frontend für unsere Schachprogramme dient `xboard`. `xboard` existiert für verschiedene Unix-Derivate und wird unter der GPL (Gnu General Public License) vertrieben und ist im Quellcode unter <ftp://ftp.gnu.org/pub/gnu/xboard/xboard-4.0.7.tar.gz> erhältlich. `xboard` kommuniziert wie viele Schachprogramme über das Chess Engine Communication Protocol (siehe Kapitel 4.10 auf Seite 121). `xboard` stellt eine Schachsituation dar und wandelt Züge, die durch eine Maus eingegeben werden in textuelle Form um. Diesen Zugstring erhält das dahinterliegende Schachprogramm, welches seinerseits einen Zugstring oder den Status des Spiels (Matt, Patt etc.) zurückliefert.

Listing 5 Einfaches pvm-gestütztes Programm (Anfrage)

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include "pvm3.h"
4
5 int main (void) {
6     int cc,tid,ret;
7     char buf[100],tmp;
8     int num = 5;          /* Anzahl der zu berechnenden Quadrate */
9
10    /* num Anfragen losschicken */
11    for (int i =1 ; i <= num ; ++i) {
12        if ( (cc = pvm_spawn("calcxsquare", (char **) 0, 0, "", 1, &tid)) ==1 ) {
13            printf ("habe tid: %x gestartet\n", tid);
14
15            pvm_initsend(PvmDataDefault);
16            if ( (ret = pvm_pkint(&i, 1, 1 )) < 0)
17                perror("pkint");
18            pvm_send(tid,1);
19        } else {
20            perror("starting calcxsquare");
21        }
22    }
23
24    /* num Antworten erhalten */
25    for (int i =1 ; i <= num ; ++i) {
26        cc = pvm_recv(-1, -1);
27        pvm_bufinfo(cc, (int*)0, (int*)0, &tid); //von welcher tid ist die Antwort
28        pvm_upkstr(buf);          // in char* wandeln
29        printf("Antwort von %x: %s\n", tid,buf);
30    }
31    pvm_exit();
32    exit(0);
33 }
34
```

Listing 6 Einfaches pvm-gestütztes Programm (Berechnung)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include "pvm3.h"
5
6 int main(void) {
7     int ptid,cc,i;
8     char buf[100];
9
10    ptid = pvm_parent(); /* "Eltern" tid */
11    pvm_recv(ptid, 1);
12    pvm_upkint(&i,1,1);
13
14    strcpy(buf, "Berechnung findet auf ");
15    gethostname(buf + strlen(buf), 64);
16    strcat(buf, " statt. Antwort ist: ");
17    sprintf(buf + strlen(buf), "%d" , i*i);
18
19    pvm_initsend(PvmDataDefault);
20    pvm_pkstr(buf);
21    pvm_send(ptid, 1);
22
23    pvm_exit();
24    exit(0);
25 }
```

4.10 Crafty

Crafty ist ein für unsere Zwecke kostenloses Computerschachprogramm. Es ist im Quellcode unter <ftp://ftp.cis.uab.edu/pub/hyatt/> erhältlich. Wir verwenden Crafty in der Version 17.9.

Das Schachprogramm wird benutzt um die in der Datenbank enthaltenen Schachbretter zu vervollständigen. Die Spiele in der Datenbank enthalten keine Schachstellungen aus „Endphasen“ von Spielen, da Schachpartien von menschlichen Spielern, wie sie in unserer Datenbank gespeichert sind, oftmals vorher aufgegeben werden.

Crafty unterstützt das Chess Engine Communication Protocol.⁴⁴ Dieses Protokoll wird von vielen Schachprogrammen und Frontends, unter anderem xboard unterstützt. Damit ist es möglich, mit Crafty mit standardisierten Befehlen zu kommunizieren. Es wurde ein Programm geschrieben, das mit Crafty über diese Schnittstelle kommuniziert. Dieses Programm holt die Spielstellungen aus der Datenbank, die aus abgebrochenen Spielen resultieren, und übergibt diese an Crafty. Jetzt wird Crafty veranlasst, diese Partien zu Ende zu spielen. Die Ausgaben von Crafty werden wieder in Schachbretter umgewandelt und in die Datenbank geschrieben.

Weiterhin wird Crafty dazu benutzt, Bewertungen für die Stufe b) (siehe Kapitel 6.2.4.2) zu erstellen. Dazu wird an Crafty eine Spielposition übergeben. Anschließend führt Crafty eine Bewertung der Position durch und übergibt die gewonnenen Werte an die Datenbank.

⁴⁴ <http://www.research.digital.com/SRC/personal/mann/xboard/engine-intf.html>.

5 Vorgehen

In diesem Kapitel wird ein weitgehend chronologischer Überblick darüber gegeben, welche einzelnen Aufgaben sich der PG stellten und wie diese angegangen wurden.

5.1 Allgemeiner Überblick

Am Anfang stand die Festlegung der grundsätzlichen Konzeption des Schach-GP-Systems, wobei das im Projektgruppenantrag erwähnte Konzept (Evolution von Steuerungsprogrammen für jeden Schachstein-Typ, die abhängig von der Brettstellung jeweils einen regelgerechten Zug vorschlagen, von denen auf einer darüberliegenden Ebene genau ein Zug ausgeführt wird) aufgrund von Überlegungen zu der Funktion von Schachsteinen im Schachspiel⁴⁵ letztlich durch das in Kapitel 6.1 beschriebene Stufenkonzept ersetzt wurde. Das erste Konzept wurde dabei jedoch nicht gänzlich verworfen, sondern es könnte später vielleicht als alternativer Entwurf implementiert werden, um es dann mit dem von uns favorisierten Konzept zu vergleichen⁴⁶.

Nachdem das Konzept stand, musste man sich einen groben Überblick über die anfallenden Aufgaben verschaffen, um sie in verschiedene Aufgabenbereiche aufzuteilen, an denen man in Kleingruppen parallel arbeiten konnte. Dabei ergab sich die folgende Einteilung:

- Erstellung der Datenbank und der Fitnessfunktionen
- Erstellung der Datenstrukturen und Operationen
- Anpassung von SYSGP

In den ersten Aufgabenbereich fielen im einzelnen der Entwurf und die Erstellung von zwei Datenbanken, um eine effiziente Fitnessbewertung zu ermöglichen: In einer wurden Brettstellungen und dazugehörige statische Bewertungen gespeichert und in der anderen Züge und deren Bewertungen. Bevor die Datenbanken gefüllt werden konnten, war, aufgrund der Beschaffenheit der Trainingsdaten, deren Auswahl in Abschnitt 5.2 ausführlich beschrieben ist, noch eine Vorverarbeitung der Daten vonnöten: Da die Partiedaten im PGN-Format⁴⁷ vorlagen und auch nur Zugfolgen anstatt der benötigten Stellungen beschrieben, mussten hierfür noch Konvertierungs- und Importfunktionen erstellt werden. Weiterhin mussten vor der Füllung der Datenbank alle Schachstellungen, in denen Schwarz am Zug ist, in analoge Weißpositionen umgewandelt werden, da die GP-Programme alle Schachstellungen unter der Annahme, dass Weiß am Zug ist, betrachten. Außerdem musste eine Anbindung an das Schachprogramm Crafty⁴⁸ erstellt werden, damit Partien, die sehr frühzeitig aufgegeben wurden,

⁴⁵ Wie in Abschnitt 3.1.4 erwähnt, stellt ein Schachstein nach gängiger Meinung keine hinreichende funktionelle Einheit dar, um den obigen Ansatz zu begründen.

⁴⁶ Siehe dazu auch die Verbesserungsmöglichkeiten im Fazit (Abschnitt 8).

⁴⁷ PGN: Portable Game Notation.

⁴⁸ Siehe Abschnitt 4.10.

von diesem „zuende gespielt“ werden konnten, um auch unausgeglichene Stellungen für die Datenbank zur Verfügung zu haben. Für die zweite Datenbank, in der zu jeder vorhandenen Stellung alle möglichen Züge abgelegt wurden, war die Ermittlung aller gültigen Züge aus einer gegebenen Stellung heraus notwendig. Diese Aufgabe wurde von dem Programmmodul `CHESSGP_Board`⁴⁹, das sehr mächtige Schachfunktionen zur Verfügung stellt, übernommen. In den ersten Aufgabenbereich fiel – wie bereits erwähnt – auch der Entwurf und die Erstellung der Fitnessfunktionen, die im Einzelnen in Abschnitt 6.2.4 beschrieben werden. Diese stehen in einem engen Zusammenhang mit den Datenbanken: Die Fitnessfunktionen der Stufe a greifen nämlich auf die erste Datenbank und die der Stufe b auf die zweite Datenbank zurück.

Der zweite Aufgabenbereich enthielt die Implementierung des schon erwähnten Moduls `CHESSGP_Board`, in der sozusagen die gesamte „Schach-Funktionalität“ enthalten ist. (Eine genauere Beschreibung hierzu findet sich in Abschnitt 6.2.2.1.) Außerdem gehörte der Entwurf und die Implementierung effizienter und vor allem zweckmäßiger Datenstrukturen z. B. für die Darstellung von Schachstellungen, Zügen und Zugfolgen dazu. Ferner fiel die Entwicklung der Operationen, aus denen die „Schachprogramme“/GP-Individuen bestehen sollen, in diesen Bereich. Bei diesen Operationen ist es besonders wichtig, dass sie nicht nur effizient sind, sondern auch für Genetisches Programmieren im allgemeinen geeignet sind sowie im besonderen auf den jeweils gewählten Programmtyp (Tree, Linear, Graph, LinTree⁵⁰) abgestimmt sind. Selbstverständlich ist an die Operationen auch die Forderung zu stellen, dass sie einerseits sinnvoll in einem Schachprogramm eingesetzt werden können, aber andererseits auch nicht so mächtig sind, dass ein gutes Schachprogramm unweigerlich entstehen müsste.⁵¹ Welche Operationen konkret gewählt wurden, ist in Abschnitt 6.1 nachzulesen.

Der dritte große Aufgabenbereich umfasste alles, was an Änderungen und Anpassungen an der C++-Bibliothek `SYSGP`, die ausführlich in Abschnitt 4.2 beschrieben wird, nötig war. Hierzu gehörte insbesondere die Erweiterung von `SYSGP` um den schon erwähnten Programmtyp „LinTree“⁵², der für die Evolution von Stufe b (dem „Zuglistengenerator“) benötigt wurde.⁵³ Desweiteren war es notwendig, etliche Stellen in `SYSGP` anzupassen, da trotz durchgehender Verwendung von Templates⁵⁴ für die Register an einigen Stellen explizit mit dem Datentyp „double“ gearbeitet wurde, wir aber die von uns geschriebenen und auf Schachoperationen spezialisierten Typen benutzen. Außerdem mussten noch Anpassungen an den Individuen und den jeweiligen Interpretern vorgenommen wer-

49 Siehe Abschnitt 6.2.2.1.

50 Baumstruktur mit linearen Programmen in den Knoten

51 Ein (Extrem-) Beispiel für eine zu mächtige Operation wäre „Berechne den besten Zug mit einem Brute-Force-Algorithmus und führe ihn aus“.

52 Diese Erweiterung von `SYSGP` wurde weitestgehend von Wolfgang Kantschik ausgeführt, aber auch innerhalb der PG getestet.

53 Aufgrund massiver Speicherprobleme, die vor allem durch das Crossover von (linearen) Bäumen verursacht waren, wurde der Individuentyp in Stufe b später auch auf „Linear“ umgestellt, so daß nun alle drei Stufen mit linearen Programmen arbeiten.

54 Templates sind Platzhalter für beliebige Datentypen, die erst später durch den jeweils benötigten Datentyp ersetzt werden.

den. (Ein Individuum der Stufe b besteht nicht nur aus dem LinTree-Individuum, das in dieser Stufe angelernt wird, sondern zusätzlich noch aus acht Individuen der Stufe a, die acht verschiedene statische Stellungsbewertungen zur Verfügung stellen.) Zuletzt musste auch noch für jede Stufe ein Hauptprogramm, sowie eine Konfigurationsdatei („Strukturfile“), die die Einstellungen für die anpassbaren Parameter der Evolutionsläufe enthält, geschrieben werden.

Im zweiten Teil der Projektgruppe ging es vor allem um die Evolution von Individuen in den drei Stufen. Für die Evolutionsläufe wurde die Anzahl der erlaubten Operationen und Terminale eingeschränkt, da sich in vorhergehenden Testläufen gezeigt hat, dass die Vielzahl der Operationen den Suchraum dermaßen stark vergrößert, dass eine „vernünftige“ Evolution dadurch sehr erschwert wird.⁵⁵ Außerdem wurde ein sogenannter *Restart*-Mechanismus⁵⁶ eingeführt, der die Fortschrittsrate beschleunigen soll, indem er – falls die Fortschrittsrate zu stagnieren droht – das beste Individuum aus einer vorhergehenden Generation (mit höherer Fortschrittsrate) in die aktuelle Generation übernimmt. Um die Schwierigkeit der Fitnesslandschaft besser abschätzen zu können, wurde zusätzlich ein *Random walk*-Modul implementiert.

Weiterhin mußte die Fitnessbewertung der Stufe c, die auf der im Schachspiel gebräuchlichen Elo-Bewertung⁵⁷ basiert, implementiert werden. Da diese Fitnessbewertung mit viel Rechenaufwand verbunden ist, weil die einzelnen Individuen ganze Partien gegeneinander spielen müssen, wurde zusätzlich eine Parallelisierung notwendig. Mit dem Programm *pvm* wurde daher die Ausführung der zur Fitnessbewertung notwendigen Partien auf mehrere Rechner verteilt.

Zusätzlich zu den oben schon bereits beschriebenen Änderungen wurde *SYS-GP* grundlegend überarbeitet, so dass jetzt durchgängig auf den Gebrauch von Templates verzichtet wird und von uns nicht benötigte Klassen herausgenommen sind. Allein der Verzicht auf Templates reduzierte den Speicherverbrauch um rund ein Drittel.

5.2 Auswahl der Trainingsdaten

Die Ergebnisse, die ein maschinelles Lernverfahren erzielt, hängen nicht unwesentlich von der Beschaffenheit der Trainingsdaten ab. Diese Trainingsdaten sind in unserem Fall – wie bereits erwähnt – Schachstellungen einschließlich der acht Bewertungen pro Stellung (für Stufe a). Bei der Auswahl der Schachstellungen für die Trainingsdaten stellen sich verschiedene Anforderungen:

- Die Anzahl der Beispiele sollte weder zu groß noch zu klein sein.
- Die Beispiele sollten realistische Schachstellungen sein.
- Die Trainingsmenge sollte eine möglichst hohe Diversität aufweisen.

Die von uns verwandte Trainingsmenge basiert auf 640 real gespielten Partien. Ihre Grundlage ist die 1996 erschienene Jubiläumsausgabe der jugoslawischen

⁵⁵ Siehe auch Abschnitt 7.3.3

⁵⁶ Eine genaue Beschreibung dieses Verfahrens findet sich in Abschnitt 6.1.3.

⁵⁷ Eine ausführliche Beschreibung hierzu befindet sich in Kapitel 3.6.2.

Schachzeitschrift „Sahovski informator“, in Deutschland bekannt als „Schachinformator“. Jede Ausgabe dieser 1966 gegründeten Zeitschrift enthält etwa 600 schachlich hochwertige Partien mitsamt Kommentaren namhafter Großmeister, oft der Spieler selbst. In jeder Ausgabe (zwei, ab 1991 drei pro Jahr) werden von einer Jury aus berühmten Schachspielern die jeweils zehn besten Partien der vergangenen Ausgabe prämiert.

Die Sonderausgabe „640 best games, 64 golden games“ [MAT96] fasst nun die besten Partien aus den ersten 64 Ausgaben zusammen. Dieser Band ist „auf Papier“ erschienen, und die Partien wurden „privat“ in elektronische Form gebracht. Die Partiensammlung wurde in eine Menge von etwa 40 000 Schachstellungen transformiert. Diese Kardinalität erscheint sowohl ausreichend für den Lernprozess, als auch vom Rechenaufwand beherrschbar.

Diverse Firmen bieten sehr große Datendanken mit teilweise deutlich mehr als einer Million Partien an, was einer zweistelligen Millionenzahl von Stellungen entspricht. Derart große Datenbanken enthalten zwangsläufig viele unterklassige, für „richtiges“ Schach wenig repräsentative Partien, die erst aufwendig herausgefiltert werden müssten. Daher begnügen wir uns mit der praktikabel scheinenden Verwendung der 640 Informator-Partien.

Die Realitätsnähe der in der Trainingsmenge enthaltenen Beispiele soll durch die hohe Spielkultur der „Urheber“ der Partien gesichert werden; unter letzteren finden sich alle Weltmeister der Nachkriegszeit.⁵⁸

Dieses hohe Spielniveau hat aber leider auch eine Verringerung der Diversität zur Folge, unter anderem dadurch, dass die Partien sehr zeitig aufgegeben wurden, beispielsweise schon nach dem Verlust eines einzigen Bauern. Diese im strengen Regelsinn (erst Matt beendet die Partie) verfrüht aufgegebenen Partien sollen von einem Brute-force-Schachprogramm „gegen sich selbst“ zuende gespielt werden, um auch Beispiele von nicht ausgeglichenen Stellungen in der Trainingsmenge zu haben.

Weiterhin ist die Datenbank durch die vielen beteiligten Spieler mit ihren unterschiedlichen Stilmerkmalen und Eröffnungsrepertoires divers genug, um einen allgemeinen Schachfundus darzustellen, den man für den Versuch eines universellen Schachlernens zu benötigen scheint.

5.2.1 Auswahl der Fitnesscases

Um die Programme der Stufe b, die Listen von – nach verschiedenen Kriterien ausgewählten – Zügen⁵⁹ liefern, bewerten zu können, müssen diese Listen mit vorgegebenen, „möglichst guten“ Listen verglichen werden. Zu jeder in der Datenbank vorhandenen Stellung sind daher zusätzlich zu den Stellungsbewertungen auch noch Zuglisten gespeichert. Da aber nicht in jeder Stellung auch jede Art von Zügen vorkommt (z. B. sind *Opferzüge* relativ selten, wohingegen in jeder Stellung *beste Züge* vorkommen), muss bei der Auswahl der Fitnesscases darauf geachtet werden, dass die Stellungen bevorzugt werden, in denen die be-

⁵⁸ Michail Botwinnik, Wassili Smyslow, Michail Tal, Tigran Petrosjan, Boris Spasski, R. J. „Bobby“ Fischer, Anatoli Karpow und Garri Kasparow.

⁵⁹ Zu den Auswahlkriterien siehe Abschnitt 6.2.4.2.

treffenden Züge auch tatsächlich existieren. Ansonsten würde nämlich z. B. ein Opferzug-Individuum, das *immer* eine leere Zugliste liefert, eine gute Bewertung bekommen, da in den meisten Stellungen keine Opferzüge existieren. Daher werden z. B. für die Fitnesscases der Opferzug-Bewertung nur die Datenbankstellungen, die Opferzüge beinhalten, zugelassen. Bei allen anderen Bewertungen sind sämtliche Datenbankstellungen für die Fitnesscases zugelassen. Die endgültige Auswahl der Fitnesscases erfolgt dann in der Klasse `CHESSGP_FitnessArray`, die für jede Position des Arrays eine zugelassene Datenbankstellung zufällig auswählt. Eine Stellung kann daher auch durchaus mehrmals als Fitnesscase gewählt werden.

6 Architektur

6.1 Konzept

6.1.1 Einleitung

Hauptziel der Projektarbeit ist die Evolution von GP-Agenten mit Schachwissen und deren Integration in ein Computerschachsystem. Jedes Schachsystem, welches zum Spielen geeignet ist, muss zu einer Brettstellung einen gültigen Zug ermitteln können. Ein „gutes“ Schachsystem wird darüberhinaus einen Zug liefern, der eher zum Spielsieg führt, als die anderen.

In diesem Kapitel steht die Auswahl eines (gültigen) Zugs durch das System im Mittelpunkt. Insbesondere soll erklärt werden, inwiefern dieser Schritt der Evolution unterliegt. Das restliche System bleibt von der Evolution unberührt.

Im folgenden Abschnitt werden die Grundlagen der vorhandenen Architektur motiviert. Dabei findet ein Vergleich mit anderen „Architekturen“, wie etwa menschlichen Schachspielern, statt. In den späteren Abschnitten wird die Architektur im einzelnen erläutert⁶⁰.

6.1.2 Problemspezifische Herangehensweise

Ein System zu schreiben, welches einen „guten“ Zug liefert, ist leicht. Hinweise finden sich in Kapitel 3.2. Soll das System zusätzlich ressourcenschonend sein, wird die Aufgabe schwieriger.

Dem System der Projektarbeit wird in erster Linie nicht abverlangt, dass es hinsichtlich einer problemspezifischen⁶¹ Zeitkomplexität effizient ist. Bedeutender ist der Aspekt, dass auf unkonventionelle Weise, und zwar mit Hilfe der Genetischen Programmierung, das Ziel erreicht werden soll.

An welcher Stelle setzen evolutionäre Methoden jedoch an? Und inwiefern ist der eventuelle Erfolg auf diese zurückzuführen, oder ist er nur Konsequenz des möglicherweise eingebauten „Schachwissens“⁶²?

Wenn ein System außer der Regelkenntnis nicht mit Schachwissen gespeist wird, trotzdem unter Einsatz evolutionärer Verfahren Schach „ordentlich“ spielen kann, ist der Erfolg auf diese zurückzuführen. Dies schließt aber Schachwissen nicht prinzipiell aus.

Vor der Herleitung der Konzeption wird eine plane Struktur diskutiert. Im Unterschied zu einer Baumsuche in dem Spielbaum wird hier nur die Wurzel des Spielbaumes, also die gegebene Brettstellung, „betrachtet“. Es werden also keine „Züge ausgeführt“. Man stelle sich nun ein Programm vor, welches, ohne Züge „auszuführen“, mit „einfachen“ Operationen, einen „guten“ oder sogar einen besten⁶³ Zug ermitteln soll. Dies ist zweifelsohne theoretisch möglich, man denke etwa an Boolesche Funktionen. Diese können in minimale Normalform gebracht werden, welche mit zwei Elementaroperationen auskommen. Die Simula-

60 Auf implementierungstechnische Details wird möglichst verzichtet.

61 Etwa Anzahl der Knoten im Suchbaum, die besucht werden.

62 Schachwissen umfasse neben bloßer Regelkenntnis auch Strategien und Heuristiken.

63 Ein bester Zug führt zu einem Knoten mit gleichem Minimaxwert.

tion solcher Formelbäume ist in einer planen Struktur möglich und benötigt nur einen unwesentlichen Teil an Speicher. Allerdings wird sich die Formel vermutlich nicht in hinreichend kleiner Größe repräsentieren und gleichzeitig effizient auswerten lassen.⁶⁴ Interessant könnte aber eine Approximation dieser Funktion sein, welche eine mit heutigen Ressourcen realisierbare Darstellungsgröße hat. Könnte es etwa solche geben, die einen deutlich besseren Zug „ermitteln“ als ein Zuggenerator, der einen zufälligen⁶⁵ Zug liefert?

Die Möglichkeit einer formel-ähnlichen Baum-GP-basierten Realisierung des planen Ansatzes wurde nicht näher in Betracht gezogen. Statt dessen stelle man sich eine Liste von Instruktionen, wie etwa bei Linear-GP, vor, welche Zeile für Zeile abgearbeitet wird.

In die plane Struktur an sich, ist kein Schachwissen integriert. Natürlich können viele Operationen angeboten werden. An Ausdruckskraft, im Sinne von „was ist möglich?“, fehlt es nicht.⁶⁶ Doch wird es mit GP kaum möglich sein, ohne in die Operationen Schach-Know-How zu stecken, ein „sinnvolles“ System zu erlangen. Wenn sich das System mit seiner planen Struktur um alle Einzelheiten kümmern muß, ist es zu groß, um mit Genetischer Programmierung gefunden zu werden. Die Aufgabe, einen der besten Züge zurückzugeben, ist zudem zu schwierig, um von einem evolvierten Codeblock ausgeführt zu werden.⁶⁷ Selbst wenn man mächtige Operationen zuläßt, dabei die eigentliche Berechnung aber noch im planen Programm erfolgt, ist es eher unwahrscheinlich, ein gutes Schachprogramm erhalten zu können.

Die plane Struktur wird nicht umgesetzt. Statt dessen wird der menschliche Schachspieler betrachtet, und aus den gewonnenen Erkenntnissen die Konzeption abgeleitet. Im folgenden wird aufgezählt, was einen menschlichen Schachspieler auszeichnet:

1. Gute Kenntnisse der modernen Eröffnungstheorie.
2. Verständnis der Grundideen, die in den typischen Mittelspielstellungen immer wieder auftauchen. Je größer das Wissen und Gedächtnis eines Spielers sind, desto einfacher überschaut er das Brettgeschehen und kann gegebenenfalls eine Musterstellung zum Vergleich heranziehen. Freilich ist hiermit nicht die mechanische Wiedergabe gemeint, sondern die Kenntnis von Methoden und nicht selten von eigenen Plänen, Zügen und Kombinationsideen. Wir wollen die ersten beiden Punkte aus gutem Grund als „theoretisches Rüstzeug“ bezeichnen.
3. Sorgfältige und korrekte Beurteilung einer Stellung.

64 Codiert man das Brett mit 194 Bit (die Position jeder Figur (maximal 32) mit 6 Bit und Rochadeinformationen mit zwei weiteren) und die Ausgabe mit 10 Bit (Start- und Zielfeld) erreicht man eine obere Schranke (Wertetabelle als Repräsentant) von 28544953854119197621165719388989902727654932480 Terabyte.

65 Bei uniformer Verteilung.

66 Erlaubt man etwa zwei Stacks und endliche viele Konstanten, kann ohne weiteres eine Turingmaschine simuliert werden.

67 Es handelt sich hierbei um Vermutungen.

4. Auffinden des richtigen Plans, der den Erfordernissen der gegebenen Position entspricht.
5. Rasche und präzise Berechnung jener Varianten, die sich im weiteren Spielverlauf ergeben könnten.

([JKA83], S. 11)

Für unsere PG sollen die wissensbasierten ersten beiden Punkte außer Acht gelassen werden, da menschengeschaffene Datenbanken bereits in herkömmlichen Schachsystemen integriert sind. Ontogenetisch gelernte Wissensbasen vergrößerten den Suchraum und erschwerten damit die Lernaufgabe beträchtlich.

Die anderen drei Aspekte sollen jedoch in unsere Konzeption Eingang finden. Das fertige Individuum soll demnach *Stellungen beurteilen*, *Pläne finden* und *Varianten berechnen* können. Dafür liegt eine Teilung des Individuums in drei nebeneinander bestehende Teilproblemlöser auf der Hand. Es ist leicht einsehbar, dass diese Module stufenartig aufeinanderbauen.

- Um einen Plan aufzustellen, ist es hilfreich, verschiedene Stellungen, die im Zusammenhang mit bestimmten Plänen stehen, anhand ihrer Bewertung miteinander zu vergleichen.
- Um Varianten zu berechnen, sollte „Rohmaterial“ in Form von möglichen Schachzügen, die verschiedene Pläne realisieren, vorhanden sein.

Die *Stellungsbewertung* eines Schachexperten ist sehr komplex, und über sie ist wenig bekannt. Jedenfalls kann gesagt werden, dass die menschliche Stellungsbewertung eine totale Ordnung über sinnvolle Schachstellungen legt. Wir werden versuchen, diese Stellungsbewertung durch eine Komposition aus mehreren plausibel scheinenden Abbildungen vom Typ $f : \text{Schachstellung} \rightarrow \mathcal{R}$ zu imitieren.

Ein *Plan* ist eine irgendwie miteinander verknüpfte Menge von Schachzügen. Ein geübter Schachspieler findet spontan mehrere solcher Pläne. Wäre es legitim, mehrere Züge gleichzeitig auszuführen, so bildeten die dann gewählten Züge einen Plan. Ein Plan kann aber auch Züge enthalten, die in der aktuellen Brettstellung nicht möglich sind und es erst durch andere Züge aus der Plan-Menge werden. Da es jedoch aufwendig und schwierig ist, festzustellen, welche Züge über welche Spielbaumpfade wann möglich sein könnten, werden wir uns darauf beschränken, nur zum Betrachtungszeitpunkt bereits mögliche Züge zu berücksichtigen.

Können Menschen auch hervorragend Stellungen bewerten und Pläne fassen, so verhält es sich bei der *Variantenberechnung* anders. Beispiele wie Deep Blue zeigen die Mächtigkeit einer erschöpfenden Variantenberechnung, bei der in einer Sekunde eine hohe Zehnerpotenz von Knoten abgearbeitet wird. Auch der menschliche Schachspieler traversiert einen Suchbaum, aber oft kommt er vom Pfad ab und muss zu der gegebenen Wurzelposition zurückkehren, oder er „markiert“ nicht die bereits betrachteten Varianten und durchläuft Pfade mehrfach. Ist die Zugsuche noch nicht beendet, hat die bereits verbrauchte Bedenkzeit aber

einen Schwellwert der Vertretbarkeit überschritten, so führt der Spieler meist „probabilistisch“ einen hinlänglich scheinenden Zug aus. Menschliche Spieler betrachten folglich nur eine sehr kleine Zahl von Stellungen, sogar internationale Turnierspieler untersuchen oft weniger als 20 Knoten pro Zugwahl.

Es stellt sich die Frage, wie eine solche „schlechte“ Baumsuche vom zu erstellenden System bewältigt werden kann, will man dem Menschen nacheifern. Bäume lassen sich leicht darstellen, aber sich einen „chaos-order-Durchlauf“ vorzustellen, bedarf der Phantasie. Aber warum sollte man nicht ambitionieren, gleich besser zu sein als der Mensch? Wir werden also unserem System Komponenten einer *sinnvollen Baumsuche* zur Verfügung zu stellen versuchen. Unter einer sinnvollen Baumsuche verstehen wir insbesondere, dass der Spielbaum nicht erschöpfend und damit zeitineffizient durchsucht wird. Der evolutionäre Lernprozess möge die umsetzenden Heuristiken hervorbringen.

Unser Ansatz besteht also aus drei Stufen. Auf der untersten werden Stellungsbewertungen realisiert, auf der mittleren werden Zuglisten generiert und auf der obersten wird eine Baumsuche vollzogen, deren Ergebnis ein Zug ist. Höhere Stufen können dabei auf tiefere zugreifen. Dabei beinhaltet ein Individuum alle drei Stufen. In dem Teil der Stufe a des Individuums enthält es mehrere Stellungsbewertungen, in dem Teil der Stufe b gibt es verschiedene Zuglistenlieferanten und innerhalb von Stufe c realisiert das Individuum die Baumsuche.⁶⁸

Die Baumsuche sei hier rekursiv beschrieben: Das Individuum startet an dem Knoten im Spielbaum, der der aktuellen Brettstellung entspricht. Allgemein befinde es sich an einem Knoten des Spielbaumes. Es entscheidet im Rahmen der Stufe c, ob weiter gesucht wird. Falls ja, spannt es den Baum mit einer von der Stufe b generierten Zugliste auf, führt jeden Zug aus und vollzieht eine Baumsuche an dem somit erreichten Knoten (hier erfolgt der rekursive Aufruf). Soll der Baum nicht weiter expandiert werden, berechnet das Individuum mit Hilfe der mit Stufe a gegebenen Stellungsbewertungen einen Wert des erreichten Blattes.

Auf diese Weise wird ein Spielbaum aufgespannt, dessen Struktur vollständig durch das in allen Stufen evolvierte Individuum determiniert ist. An den Blättern ist eine Bewertung zu finden, welche durch evolvierte Teile des Individuum bestimmt wird. Für diese wird über eine Rückbewertung durch einfache Maximum- und Minimumbildungen der Minimaxwert an der Wurzel berechnet, um den besten Zug zu bestimmen.

Der letzte Abschnitt wird hier von zwei Seiten beleuchtet:

1. Wird hier nicht bereits zuviel „Schachwissen“ implementiert?

Die Frage wird nicht abschließend beantwortet, allerdings wird eine provokative Behauptung aufgestellt. Da bei Schach zwei Spieler abwechselnd ziehen dürfen, entsteht auf „natürliche“ Art und Weise der Spielbaum (etwa bildlich oder mathematisch). Es gehört quasi zu den Regeln eines Zweipersonen-Nullsummenspiels mit vollständiger Information, dass jeder einen gewissen Betrag gewinnen möchte und dass dieses Ziel genau dann

68 Näheres finde der Leser in den einzelnen Unterkapiteln.

erreicht wird, wenn der andere diesen Betrag verliert. Entscheidungsalternativen der sich am Zug befindenden Person an einem Knoten sind die unterschiedlichen Züge. Es wird derjenige gewählt, der den eigenen Gewinn maximiert, bzw. den des Gegners minimiert. Dies ist die Minimaxstrategie. Um es noch klarer zu sagen: Baumsuche und Berechnung des Minimaxwertes „implizieren“ sich. Dies ist nicht wörtlich zu nehmen, etwa gibt man sich meist mit Approximationen des Minimaxwertes zufrieden, oder mit abschätzenden Schranken, für gewisse Teilbäume. Die unbewiesene Vermutung soll aber verdeutlichen, dass, falls die Entscheidung für eine Baumsuche gefallen ist, zwar noch algorithmisch keine Festlegungen getroffen wurden, aber, wie auch immer realisiert, die Berechnung des Minimaxwertes gefragt ist.

Auffallend ist, dass ein Minimaxalgorithmus gepaart mit einer einfachen Materialbewertung nur wenigen Zeilen Codes bedarf. Wir wollen den Minimaxalgorithmus nicht per Evolution finden, sondern eine eigene Suche evolvieren, welche den besten Zug berechnet. Diese benutzt lediglich zur Rückbewertung die Minimaxdefinition (Minimaxdefinition siehe Seite 13). Das die Unterschiede zu dem Minimaxalgorithmus gravierend sind, wird gegen Ende von Abschnitt 6.1.6.5 erläutert.

2. Warum wird kein schnellerer Baumsuchalgorithmus für die Rückbewertung benutzt?

Dies liegt daran, dass es auf Anhieb nicht möglich zu sein scheint, bei der Suche Teilbäume abzuschneiden, welche das Ergebnis nicht beeinflussen. Der Grund hierfür ist, dass das Individuum seinen Zustand während der Expansion an jedem Knoten ändern kann. Insbesondere ist dies in nicht relevanten Teilbäumen möglich. Die Art und Weise in der beim α - β -Verfahren die Informationen an den Blättern, also die Stellungsbewertungen, verarbeitet werden, ist stets die gleiche. Das Individuum kann hingegen in jedem Knoten Entscheidungen treffen, welche in einer unterschiedlichen Behandlung der später besuchten Knoten resultieren. Es ist insbesondere möglich, dass sich das Suchverfahren während der Suche ändert. Die dadurch denkbaren Seiteneffekte sind hier gewünscht. Dennoch ist eine Variante, welche automatisch schneiden kann, sicherlich interessant. Zunächst möchten wir aber den Individuen, welche die Aufgabe bekommen, einen Zug zu ermitteln, die freie Gestaltung ihrer Baumsuche gewähren.

Der Evolutionsablauf gliedert sich zunächst in die getrennte Evolution der einzelnen Stufen nacheinander. Dabei kann bei der Evolution einer Stufe auf die bereits evolvierten, tiefer in der Hierarchie liegenden Stufen zurückgegriffen werden. Am Schluss findet eine simultane Evolution der einzelnen vortrainierten Stufen statt. Dabei werden ähnliche Ziele festgelegt wie für die Stufe c. Es ist somit letztendlich der Evolution überlassen, was die Stufen a und b semantisch bewirken.

Im folgenden Abschnitt wird ein für die Stufe a und b grundlegendes Vorgehen während der Evolutionsläufe vorgestellt. Was die einzelnen Stufen bein-

halten und wie sie ineinander greifen, wird in den darauf folgenden Abschnitten erläutert.

6.1.3 Restart

Restart bezeichnet ein Verfahren, dass den Verlauf der Evolution anhand von Kennziffern überwacht, um gegebenenfalls mit einer Reinitialisierung des Pools unter Verwendung von „guten“ Individuen zu reagieren. Die Kennziffern werden aus der Fitnesskurve und deren Verlauf über der Zeit berechnet.

Entstanden ist diese Idee aus der Notwendigkeit des Neustarts von Läufen durch äußere Umstände (z. B. Rechnerabsturz o.ä.). Ein weiterer Grund für einen Neustart der Evolution lag in der Stagnation der Fitnesswerte, die nach längerer Zeit auf dem gleichen Niveau keine Verbesserung mehr zu versprechen schienen. Dabei erschien es sinnvoll, den Lernvorgang nicht ganz von vorne zu beginnen, sondern das bis zu diesem Zeitpunkt im Evolutionslauf entstandene Wissen zu nutzen. Dazu wurde das beste Individuum der letzten Generation des Vorlaufes eingelesen und an verschiedene zufällig gewählte Stellen in den Pool gesetzt. Die Anzahl der Kopien wird durch einen Parameter festgelegt.

Der nächste Schritt bestand darin, aus dem von einem Benutzer vorgenommenen Neustart eines Laufes einen automatischen und softwaregesteuerten *Restart* zu machen. Dazu wurde die eben skizzierte Vorgehensweise der Poolinitialisierung mit Bewahrung einiger gewünschter Individuen entwickelt.

Ein wenig genauer betrachtet, gestaltet sich der Vorgang des „internen“ Restarts folgendermaßen:

Während des Laufes wird aus jeder Generation das beste Individuum in einem zweiten Pool gesichert. Auf diesen Pool wird bei einem Restart zurückgegriffen. Die Überprüfung, ob ein Restart durchgeführt werden soll, findet nur zu bestimmten Zeitpunkten (*Restartpunkte*) statt, die vom Benutzer vorgegeben werden. Die Idee hierbei ist es, nach einer gewissen Zeitspanne in die Vergangenheit und die Steigung der Fitnesskurve zu betrachten. Ist diese über einen längeren Zeitraum betrachtet zu „flach“, so kann vermutet werden, dass der Evolutionsprozess zum Stillstand gekommen ist. Das Kriterium hierfür ist die Steigung der Fitnesskurve zwischen dem letzten Restartpunkt⁶⁹ und der aktuellen Generation. Ist diese Steigung geringer als es der Benutzer im Strukturfile vorgegeben hat, wird der Restart durchgeführt.⁷⁰

Nun muss noch spezifiziert werden, *welche* der gesicherten Individuen in den neuen Pool übernommen werden sollen. Dazu werden drei Annahmen gemacht:

1. Interessant sind diejenigen Individuen aus den Generationen, die sich mitten in einem Abwärtstrend der Fitnesswerte (zum Optimum hin) befanden, also aus einer „turbulenten“ Zeit vor der Stagnation.

⁶⁹ Beziehungsweise der ersten Generation, falls es sich um den ersten Restartpunkt handelt.

⁷⁰ Da in unserem Fall der zu erreichende optimale Fitnesswert 0 ist, und es sich somit und eine negative Steigung handelt, wird überprüft, ob die Steigung einen bestimmten Wert überschreitet (in den meisten Fällen ist dies -1).

2. Diese Zeitspanne kann angenähert mit einem Kriterium ermittelt werden, das die *Rate* der Fitnesswerte auf einen angenommenen Endzeitpunkt hin bestimmt (*Ratenkriterium*).
3. Um keine Rückschritte in der Fitnessentwicklung zu realisieren, ist es nötig auf jeden Fall das beste Individuum der letzten Generation ebenfalls in den neuen Pool zu übernehmen.

Somit sollen sowohl aktuelle Individuen gesichert werden, als auch „interessante“ Individuen, die unter Anwendung des Ratenkriteriums gesucht werden.

Das Ratenkriterium entstand aus der Fragestellung, welcher durchschnittliche Fitnessverbesserung pro Generation vorhanden sein muss, um das Optimum in einer vorgegebenen Zeit zu erreichen. So ergibt sich z.B. bei einem Startfitnesswert von 5 000 und einer vorgegebenen Laufzeit von 1 000 Generationen, dass ungefähr ein Fortschritt von 5 Fitnesszählern im Schnitt pro Generation nötig ist, um zu diesem Ziel zu gelangen. Die *minimale Rate* ist dieser Wert, der mindestens im Durchschnitt erzielt werden muss.⁷¹

Dieses Kriterium wurde für den Restart dazu benutzt, um die Zeiträume in der Fitnessentwicklung zu finden, in denen „noch viel passierte“. Grob abgeschätzt waren das die Zeiträume, in denen die aktuell gemessene Rate mindestens doppelt so hoch war wie die minimale Rate. Nach diesen Zeiträumen wird im Falle eines Restarts gesucht, um von dort Individuen in den neuen Pool zu übernehmen. Anschliessend an die Suche nach geeigneten Individuen – die auch erfolglos verlaufen kann – wird ein neuer Pool erzeugt, der den alten Pool überschreibt und mit den gewünschten Individuen „angereichert“ wird.

Zusammenfassend dargestellt gestaltet sich der Ablauf des Restarts innerhalb des Evolutionslaufes wie folgt:

- In jeder Generation werden die jeweils besten Individuen gesichert und die aktuelle Rate ermittelt.
- Wenn an einem Restartpunkt das Steigungskriterium zutrifft wird ein Restart durchgeführt:
 - Ermittlung der Generation mit stark abfallenden Tendenzen (Richtung Optimum) in der Fitnesskurve (*GenAlt*).
 - Erzeugung einer neuen zufälligen Population, die die alte ersetzt.

⁷¹ Dazu sei noch angemerkt, dass der Ablauf der Hauptschleife vor der Einführung des Restarts noch über eine festgelegte Anzahl Generationen determiniert wurde. Für eine Abschätzung des Evolutionsverlaufes war es hilfreich zu wissen, welcher Evolutionsfortschritt approximativ nötig ist, damit das Evolutionsziel in der vorgegebenen Zeit erreicht werden kann. blieb der aktuelle Fortschritt zu lange unter dem Mindestniveau, war ein erfolgreicher Abschluss unwahrscheinlich. Später wurde die Hauptschleife dahingehend geändert, dass sie erst bei Erreichen des Optimums abbricht. Durch die automatische Durchführung des Neustarts als Restart war eine vorgegebene Evolutionszeit nicht mehr sinnvoll. Der Parameter *Generation* aus dem Strukturfile hatte nun keine Auswirkung auf die Länge des Evolutionslaufes, wurde aber weiterhin für die Ermittlung der Rate genutzt.

- Einfügen einer spezifizierten Anzahl Individuen aus GenAlt und eines Individuums der letzten Generation an zufällige Stellen im Pool.
- Durchführung der genetischen Operatoren und fortfahren mit der Evolutionsschleife.

6.1.4 Stufe a – die Bewertung einer Brettstellung

6.1.4.1 Grundgedanken

Das Individuum der Stufe a berechnet eine Zahl. Es kann dabei auf Informationen der aktuellen Brettstellung über Operationen zugreifen. Die Zahl wird als statische Bewertung, der Stellung auf dem Brett interpretiert. Jedes Individuum realisiert somit eine statische Bewertungsfunktion $f_{\text{ind}} : \mathcal{A} \rightarrow \mathbb{R}$, welche eine Schachstellung aus der Menge aller zulässigen⁷² Schachstellungen \mathcal{A} auf eine reelle Zahl abbildet. Die Bewertungsfunktion f_{ind} soll nun auf einem bestimmten Definitionsbereich $\mathcal{B} \subset \mathcal{A}$ eine von acht vorgegebenen Funktionen möglichst gut approximieren. Der einschränkende Definitionsbereich soll insbesondere nicht erreichbare Stellungen ausgrenzen. Es wird versucht, nur „relevante“⁷³ Stellungen zu betrachten. Es wird gehofft, dass sich das Individuum nur im Verhalten, also seiner Funktion $f|_{\mathcal{B}}$, an die von außen vorgegebene Bewertungsfunktion anpasst.

Die Bewertungsfunktionen unterscheiden sich hinsichtlich des Blickwinkels des Bewertenden. Vier der Funktionen sind neutrale Bewertungen der Schachstellung (unabhängig von der Farbe am Zug). Die anderen bewerten die Stellung aus der Sicht des Spielers, der gerade am Zug ist.

Im folgenden werden die einzelnen Bewertungsfunktionen genannt, die von außen vorgegeben werden. Jedes a-Individuum wird genau einer davon zugeordnet. Die ersten vier Bewertungen sind vom neutralen Betrachter, die letzten vier vom Spieler am Zug aus zu sehen.

1. Einfache Materialbewertung: Die Figuren jeder Farbe werden, je nach Typ unterschiedlich gewichtet und getrennt addiert; anschließend wird die Differenz gebildet.
2. Clevere Stellungsbewertung: Diese setzt sich aus einem Materialteil und einem Teil zusammen, der die Stellung bezüglich positioneller Kriterien bewertet.
3. Komplexitätsbewertung: Es wird angegeben, wie *komplex* die Stellung ist.
4. Endspielbewertung: Es wird angegeben, in welchem Maße ein Endspielstatus vorhanden ist.
5. Angriffsbewertung: Es wird angegeben, inwiefern der Spieler am Zug einen Angriff führt.

⁷² In diesem Abschnitt werden Stellungen als zulässig bezeichnet, in denen genau zwei, verschieden farbige Könige vorkommen, welche nicht benachbart sind und von denen maximal einer im Schach steht.

⁷³ Relevant sind solche Stellungen, die im „Spiel“ tatsächlich „vorkommen“.

6. Verteidigungsbewertung: Es wird angegeben, inwiefern der Spieler am Zug verteidigend steht.
7. Defensiv-Bewertung: Es wird angegeben, wie defensiv die Stellung des Spielers am Zug ist.
8. Aggressivitätsbewertung: Es wird angegeben, wie aggressiv die Stellung des Spielers am Zug ist.

6.1.4.2 Datentyp der Stufe *a*

Objekte dieses Datentyps sollen Felder⁷⁴, Linien und Reihen des Schachbretts adressieren, mit Schachfiguren umgehen und eine Zahl abspeichern können. Dazu speichern sie eine Linie (line) und Reihe (row) des Brettes, eine Figur (piece) und eine Zahl (value). Es handelt es sich bei den Linien, Reihen und Figuren auch um Zahlen, welche bijektiv abgebildet werden, um stets eine wohldefinierte Bedeutung zu haben. Es ist dabei nicht erforderlich, dass diese Daten in einem Kontext stehen. Vielmehr wird dieser durch einige Operationen hergestellt.

Es gibt mehrere Register dieses Datentyps, darunter ein ausgezeichnetes – der Akkumulator, kurz Akku. Dieser ist an den meisten Operationen per Definition beteiligt.

6.1.4.3 Konstanten der Stufe *a*

Die Konstanten sind Objekte des Datentyps, mit zufälliger aber fester Belegung der Feld-, Figur- und Zahlvariable. Dabei liegt eine uniforme Verteilung zu Grunde.

6.1.4.4 Operationen der Stufe *a*

Die semantische Trennung in Feld-, Figur- und Zahlvariablen soll von den Operationen *sinnvoll* berücksichtigt werden. Es soll etwa keine Operationen geben, welche die Linie mit der Reihe multipliziert und das Ergebnis in der Figur speichert.⁷⁵

Es gibt prinzipiell zwei Typen von Operationen: Basisoperationen und Terminale. In jedem Programmschritt wird eine Basisoperation ausgeführt.⁷⁶ Viele von diesen arbeiten auf dem Akkumulator. Einige fordern einen zusätzlichen Operanden. Dies kann eine Konstante, ein Register oder ein Rückgabewert eines Terminals sein. Genau im letzten Fall, wird ein Terminal ausgeführt. Terminale können also nicht, wie Basisoperationen, direkt aufgerufen werden, sondern nur über eine Basisoperation.

⁷⁴ Felder werden über die Linie und Reihe adressiert.

⁷⁵ Da die Menge der zulässigen Figurenzahlen, bezüglich der Multiplikation ein Gruppe bilden, wäre hier zumindest kein zahlentheoretisch begründbarer Vorteil für einzelne Figuren erkennbar.

⁷⁶ Die Schleifenoperationen stellen hier eine Ausnahme dar, da sie den Programmablauf beeinflussen.

Während Basisoperationen oft den Akku überschreiben, lassen Terminale alle Register unberührt. Das Berechnungsergebnis wird ausschließlich durch die Rückgabe vermittelt. Die aufrufende Basisoperation verarbeitet das berechnete Ergebnis weiter.⁷⁷

Viele Operationen können das Schachbrett einsehen, sie liefern bestimmte Informationen. Keine Operation kann das Schachbrett selbst verändern. Im folgenden sind die einzelnen Operationen in Gruppen aufgeführt. Handelt es sich um Terminale, so wird extra darauf hingewiesen.

Operationen für die Steuerung des Programmablaufs

1. **boardLoop**: Wiederholt die folgenden n Operationen für alle Felder des Schachbrettes. Die Zahl n wird dabei dem **value**-Attribut des Akkus entnommen. Vor jeder Wiederholung wird das aktuelle Feld in den Akku geschrieben. Die Operationen in der Schleife können auf dieses Feld zugreifen, sie können es auch verändern. Letzteres kann nicht zu Endlosschleifen führen, da das Feld am Schleifenkopf stets unabhängig vom Inhalt mit dem richtigen Wert überschrieben wird. Ferner ist eine maximale Grenze für n von 100 vorgesehen. Größere Werte werden modulo 100 interpretiert. Alle Schleifenbefehle innerhalb eines Schleifenbefehls werden ignoriert, um Schachtelungen und damit verbundene Laufzeitprobleme zu vermeiden.
2. **rowLoop**: Wiederholt n Operationen für alle Felder einer Reihe des Schachbrettes. Ansonsten analog zu **boardLoop**.
3. **lineLoop**: Wiederholt n Operationen für alle Felder einer Linie des Schachbrettes. Ansonsten analog zu **boardLoop**.
4. **pieceLoop**: Wiederholt n Operationen für alle Felder des Schachbrettes, die in einem Zug von dem Stein auf dem aktuellen Feld (des Akkus) *erreicht* werden können. Dies sind alle Felder, die auf legale Weise erreicht werden können (ein sich möglicherweise indirekt ergebendes Schach wird allerdings ignoriert), wenn alle anderen Figuren die gegnerische Farbe hätten, also auch Felder mit eigenen Figuren. Auf diese Weise kann das Individuum theoretisch einsehen, inwiefern dieses Feld von den eigenen Figuren Deckung genießt oder diesen bietet.
5. **ifThen**: Wenn der Wert des **Zahl**-Attributs des Akkus größer als 0 ist, dann werden n (wird übergeben) Instruktionen übersprungen, falls nicht genügend da sind, wird die Programmausführung beendet. Sprünge sind immer vorwärtsgerichtet. Ein Sprungbefehl ist die einzige Möglichkeit, die oben genannten Schleifenoperationen vorzeitig abzuberechnen. Falls diese Anweisung aus dem Bereich der Schleife herausspringt, terminiert diese automatisch.

Operationen zur Ermittlung von Werten

⁷⁷ Zumindest wird das übergebene Register verwendet.

1. **whichPiece**: Liefert den Typ der Figur auf dem im Akku gespeicherten Feld.
2. **pieceValue**: Liefert eine einfache statische Bewertung des Steines, der im **piece**-Attribut des Akkus steht. Weiße Figuren bekommen positive Werte, schwarze Figuren negative. Die Absolutbeträge sind gleich: Bauern bekommen den Wert 10 zugeordnet, Springer und Läufer 30, Türme 50, Damen 90 und Könige 1 000. Leere Felder bekommen den Wert 0. Diese und die folgenden drei Operationen sind die einzigen Terminale. Sie können nur von Basisoperationen aufgerufen werden.
3. **attackedByValue**: Liefert die Bewertung desjenigen Steines, der den Stein auf dem im Akku gespeicherten Feld angreift. Bei mehreren Steinen wird der kleinste Wert geliefert.
4. **attackedByNumber**: Liefert die Anzahl der Figuren, die den Stein auf dem im Akku gespeicherten Feld bedrohen.
5. **legalMovesNumber**: Liefert die Anzahl aller regelgerechten Züge, die der Stein auf dem im Akku gespeicherten Feld ausführen darf.

Arithmetische Operationen

1. **add**: Addiert zum **value**-Attribut des Akkus den übergebenen Wert. Dieser kann eine Konstante, der Wert eines Registers, aber auch das Ergebnis einer Operation sein.
2. **sub**: Subtrahiert den übergebenen Wert vom **value**-Attribut des Akkus. Der Wert kann eine Konstante, der Wert eines Registers, aber auch das Ergebnis einer Operation sein.
3. **mul**: Multipliziert das **value**-Attribut des Akkus mit einem übergebenen Wert. Der Wert kann eine Konstante, der Wert eines Registers, aber auch das Ergebnis einer Operation sein.
4. **div**: Dividiert das **value**-Attribut des Akkus durch einen übergebenen Wert. Der Wert kann eine Konstante, der Wert eines Registers, aber auch das Ergebnis einer Operation sein.
5. **absolute**: Ersetzt den Wert des **value**-Attributs des Akkus durch dessen Betrag.

Operationen zum „Verschieben“ des aktuellen Feldes

1. **rowShift**: Addiert die Zahl aus dem **value**-Attribut des Akkus zu dem Wert des **row**-Attributs des Akkus. Das aktuelle Feld wird also nach oben (bei negativen Zahlen: nach unten) verschoben. Falls das Ende der Reihe überschritten wird, wird nicht von vorn angefangen (=modulo 8 gerechnet), sondern es wird der höchste mögliche Wert (=8) geliefert. Für

das Überschreiten des Anfangs der Reihe (durch Addition einer negativen Zahl) wird entsprechend der niedrigste mögliche Wert (=1) geliefert. Diese Technik wird mit *saturating arithmetic* bezeichnet.

2. **lineShift**: Addiert die Zahl aus dem **value**-Attribut des Akkus zu dem Wert des **line**-Attributs des Akkus. Das aktuelle Feld wird also nach rechts (bei negativen Zahlen: nach links) verschoben. Falls das Ende der Linie überschritten wird, wird nicht von vorn angefangen (=modulo 8 gerechnet), sondern es wird der höchste mögliche Wert (=8) geliefert. Für das Überschreiten des Anfangs der Linie (durch Addition einer negativen Zahl) wird entsprechend der niedrigste mögliche Wert (=1) geliefert.
3. **northWestShift**: Addiert die Zahl aus dem **value**-Attribut des Akkus zu dem Wert des **line**-Attributs und subtrahiert dieselbe Zahl vom **row**-Attribut des Akkus. Das aktuelle Feld wird also nach rechts unten (bei negativen Zahlen: nach links oben) verschoben. Falls das Ende der Linie überschritten wird, wird nicht von vorn angefangen (=modulo 8 gerechnet), sondern es wird der höchste mögliche Wert (=8) geliefert. Für das Überschreiten des Anfangs der Linie (durch Addition einer negativen Zahl) wird entsprechend der niedrigste mögliche Wert (=1) geliefert.
4. **northEastShift**: Addiert die Zahl aus dem **value**-Attribut des Akkus zu dem Wert des **line**-Attributs und des **row**-Attributs des Akkus. Das aktuelle Feld wird also nach rechts oben (bei negativen Zahlen: nach links unten) verschoben. Falls das Ende der Linie überschritten wird, wird nicht von vorn angefangen (=modulo 8 gerechnet), sondern es wird der höchste mögliche Wert (=8) geliefert. Für das Überschreiten des Anfangs der Linie (durch Addition einer negativen Zahl) wird entsprechend der niedrigste mögliche Wert (=1) geliefert.

Lade- und Speicheroperationen

1. **loadAll**: Lädt den gesamten Inhalt eines Registers, dessen Adresse übergeben wird, in den Akku.
2. **loadPiece**: Lädt den Inhalt des **piece**-Attributs eines Registers, dessen Adresse übergeben wird, in den Akku.
3. **loadValue**: Lädt den Inhalt des **value**-Attributs eines Registers, dessen Adresse übergeben wird, in den Akku.
4. **loadLine**: Lädt den Inhalt des **line**-Attributs eines Registers, dessen Adresse übergeben wird, in den Akku.
5. **loadRow**: Lädt den Inhalt des **row**-Attributs eines Registers, dessen Adresse übergeben wird, in den Akku.
6. **storeAll**: Speichert den gesamten Inhalt des Akkus in einem Register, dessen Adresse übergeben wird.

7. **storePiece**: Speichert den Inhalt des **piece**-Attributs des Akkus in einem Register, dessen Adresse übergeben wird.
8. **storeValue**: Speichert den Inhalt des **value**-Attributs des Akkus in einem Register, dessen Adresse übergeben wird.
9. **storeLine**: Speichert den Inhalt des **line**-Attributs des Akkus in einem Register, dessen Adresse übergeben wird.
10. **storeRow**: Speichert den Inhalt des **row**-Attributs des Akkus in einem Register, dessen Adresse übergeben wird.

Operationen zum Setzen von Konstanten

1. **setAll**: Setzt den Inhalt des Akkus auf eine Konstante, ein Register oder den Rückgabewert eines Terminals. Der zu setzende Wert wird bei allen set-Operationen übergeben.
2. **setPiece**: Setzt den Inhalt des **piece**-Attributs des Akkus auf eine Konstante, ein Register oder den Rückgabewert eines Terminals.
3. **setValue**: Setzt den Inhalt des **value**-Attributs des Akkus auf eine Konstante, ein Register oder den Rückgabewert eines Terminals.
4. **setLine**: Setzt den Inhalt des **line**-Attributs des Akkus auf eine Konstante, ein Register oder den Rückgabewert eines Terminals.
5. **setRow**: Setzt den Inhalt des **row**-Attributs des Akkus auf eine Konstante, ein Register oder den Rückgabewert eines Terminals.
6. **swap**: Tauscht die Inhalte von Akku und eines Register komplett gegeneinander aus.

6.1.4.5 Programmausführung der Stufe a

Abgesehen von den vier Schleifenoperationen und dem Sprungbefehl, wird das Programm Knoten für Knoten abgearbeitet. Wie bereits erwähnt, ist die Schleifenlänge beschränkt. In der Schleife werden allerdings weitere Schleifenbefehle ignoriert. Der Sprungbefehl, der stets vorwärtsgerichtet ist, kann die Schleife auch durch Verlassen abbrechen.

6.1.5 Stufe b – die Zuglistenlieferanten

6.1.5.1 Grundgedanken

Ein *b-Programm* hat die Aufgabe, zu einer gegebenen Schachstellung eine Zugliste zu ermitteln. Mit diesen Zuglisten wird vom Individuum im Rahmen der Stufe c der Suchbaum aufgespannt. Wie die *a-Programme* werden die *b-Programme* komplett evolviert. Den *b-Programmen* stehen, neben der Sicht auf das Schachbrett, *a-Programme* zur Verfügung, welche nach Belieben gestartet

werden dürfen. Allerdings ist es nicht gestattet, das Schachbrett geeignet zu verändern, um etwa einen Zug auszuführen. Es handelt sich also, wie bei den a-Programmen, um eine statische Sicht auf das Schachbrett. Insbesondere wird die Vorwegnahme einer Baumsuche somit nicht gesondert⁷⁸ unterstützt.

Auch für die b-Programme sind wieder unterschiedliche Typen vorgesehen. B-Programme des gleichen Typs sollen Zuglisten berechnen, deren Züge sich in eine bestimmte Kategorie einordnen lassen. Die Züge sollen dabei gemäß der Ausprägung, des durch die Kategorie geforderten Merkmals, absteigend sortiert sein.⁷⁹ Vorgesehen sind folgende Typen:⁸⁰

1. Beste Züge, welche also einen guten Minimaxwert liefern und somit im allgemeinen zu empfehlen sind.
2. Gefährliche Züge
3. Desperado-Züge
4. Angriffszüge
5. Verteidigungszüge
6. Tauschzüge
7. Opferzüge

Die Bezeichnung „Beste Züge“ wirft die Frage auf, ob ein entsprechendes b-Programm nicht bereits ausreicht, das in der Einleitung formulierte Gesamtziel, einen „guten“ Zug zu finden, zu erreichen. Dies würde die Architektur, insbesondere die Stufe c, in Frage stellen. Es wird aber nicht davon ausgegangen⁸¹, dass es möglich ist, b-Programme zu evolvieren, welche nur beste Züge liefern, oder alle Züge annähernd nach Wertigkeit sortieren. Vielmehr wird bereits mit erheblichen Schwierigkeiten gerechnet, Programme zu finden, welche verstärkt bessere Züge liefern. Dies liegt insbesondere daran, dass die Stufen a und b plan⁸² sind.

Nichtsdestotrotz wird vermutet, dass die Programme Heuristiken realisieren könnten, welche von der Evolution gefunden wurden. Nicht bloß die Qualität⁸³ der Züge, sondern auch die Reihenfolge und Listenlänge spielen eine gehobene Bedeutung. Je kleiner die Liste und je besser die Reihenfolge⁸⁴ ist, desto schneller kann die Suche erfolgen.

Um den Suchraum zu verkleinern und somit schnell sinnvolle Lösungen zu erlangen, wird vereinbart, dass nur mit zulässigen⁸⁵ Zügen operiert wird. Anfangs

78 Es bleibt dem Programm, unter geschickter Einsetzung der Operationen, allerdings die Möglichkeit, eine partielle Baumsuche approximativ zu simulieren.

79 Die Reihenfolge wird bei der Fitnessbewertung berücksichtigt.

80 Eine genaue Beschreibung finde der Leser in Kapitel 6.2.4.2.

81 Die Möglichkeit wird aber prinzipiell nicht ausgeschlossen.

82 Der Begriff plan, wird in Abschnitt 6.1.2 erläutert.

83 Definiert über die Bewertungsfunktionen, die in Kapitel 6.2.4.2 besprochen werden.

84 Der immensen Bedeutung der Reihenfolge widmet sich Abschnitt 3.2.3.2.

85 Zulässig sind genau die Züge, die gemäß den Schachregeln möglich sind.

steht dem b-Programm die Menge der legalen Züge zur Verfügung. Die Operationen garantieren, dass die berechnete Liste nur zulässige Züge beinhalten wird. Es ist jedoch möglich, dass auch die leere Liste berechnet wird. Dieses Verhalten wird insbesondere dann befürwortet, wenn auf der aktuellen Brettstellung keine Züge möglich sind, bei denen die durch den Typ festgelegte Eigenschaft stark ausgeprägt ist.

Im folgenden wird die Programmausführung beschrieben. Anschließend werden Datentyp, Konstanten und Operationen vorgestellt.

6.1.5.2 Programmausführung

In Stufe b gibt es ein Programm, dessen Struktur und Interpretation eine neue Kombination zwischen linearem GP und Baum-GP bildet. Die von Wolfgang Kantschik an der Universität Dortmund entwickelte Struktur werde mit „Tree-Linear-GP“ bezeichnet. Im folgenden werden die Merkmale des Programmtyps skizziert:

- **Struktur** Das Programm hat die Struktur eines Baumes, dessen Knoten lineare Sequenzen von Instruktionen enthalten. Jeder innerer Knoten hat genau zwei Nachfolger.
- **Interpretation** Die Interpretation startet an der Wurzel. Am aktuellen Knoten, wird die Sequenz der Instruktionen iterativ abgearbeitet. Dabei greifen die Operationen auf globale Register zu. Handelt es sich bei dem Knoten um ein Blatt, so ist die Interpretation abgeschlossen. Der Rückgabewert findet sich in einem vordefinierten Register. Ansonsten ist durch die Sequenz festgelegt worden, welches Kind des Knotens als nächstes interpretiert wird.
- **Initialisierung, Mutation und Rekombination** erfordern spezielle Anpassungen, die hier nicht aufgeführt werden.

Im Unterschied zu Baum-GP wird bei der Interpretation eines „Tree-Linear-GP“-Individuums, den Knoten kein Wert zugeordnet. Insbesondere werden berechnete Werte nicht zur Wurzel zurückgereicht.

Im folgenden werden Ideen diskutiert, Tree-Linear-GP zu verbessern, daneben bieten sich Verbesserungen der Initialisierungs-, Mutations- und Rekombinationsoperatoren.

- Bisher kann ein Kind nur explizit gewählt werden. Ein Negieren der bisher getroffenen Festlegung (etwa der Standardeinstellung) könnte von Vorteil sein.
- Bedingte Sprungbefehle, welche an das Ende einer Sequenz springen, könnten von Vorteil sein.
- Eine einseitige Kommunikation zwischen aufeinanderfolgenden Knoten ist denkbar. Ein interpretierter Knoten gibt etwa einen Wert an sein Kind

weiter.⁸⁶ Auch wenn diese prinzipiell über die globalen Register möglich ist, würde die Benutzung bereits berechneter Information gefördert.

- Der Einsatz von zusätzlichem lokalen Speicher, den nur die jeweilige Sequenz eines Knotens nutzen darf, wäre vermutlich von Vorteil und könnte deswegen im Laufe der PG noch näher untersucht werden. Auf diese Weise würde lokal auf großem Platz eine Berechnung möglich, das platzsparende Ergebnis könnte aber allein global platziert werden.
- Außerdem könnte man das Konzept zu azyklischen Graphen erweitern. Diese könnten Bausteine auf mehreren Wegen benutzen. Dadurch würde die Ausdruckskraft von kleineren⁸⁷ Individuen erhöht, gleichzeitig würde die Ausführungszeit im worst case ebenso linear, bezogen auf die Größe des Individuums, bleiben. Probleme ergeben sich bei der Formulierung der Rekombinationsoperatoren, dort könnten eventuell solche Einsatz finden, die in Graph-GP angewendet werden.
- Interessant wäre die Einführung von randomisierten Knoten. Diese enthalten eine Wahrscheinlichkeit, unter deren Verwendung die Wahl des Kindes zufällig erfolgt. Zum einen könnte diese durch Initialisierung/Mutation verändert werden, es ist aber auch denkbar, dass das Individuum sich diese selbst berechnet.⁸⁸
- Eine parallele, nichtdeterministische Variante, wie etwa bei *Nondeterministic Decision Diagrams*, beschrieben etwa in [WEG00], wird wegen des enorm hohen Rechenzeitbedarfs bei konkreter Auswertung, nicht in Betracht gezogen.

6.1.5.3 Datentyp

Es gibt konstant viele Register, darunter auch einen Akkumulator. Ein Register besteht aus einer reellen Zahl, einer Suchmaske und einer Zugliste, wie in Abbildung 11 auf der nächsten Seite dargestellt. Das Zahl-Feld dient zur Adressierung und als Quelle und Ziel diverser Operationen, etwa den arithmetischen. Die Züge der Zugliste sowie die Suchmaske haben einen gemeinsamen Datentyp, der in Abbildung 12 dargestellt ist.

Er enthält redundante Informationen. Das Zahlattribut der Suchmaske und der Züge kann vom b-Programm als Maß dafür verwendet werden, wie gut (angriffslustig, komplex, ...) ein Zug vom Individuum eingeordnet wird. Mit der Suchmaske werden alle Züge dargestellt, die ihren einzelnen Komponenten genügen.⁸⁹ Für jede Komponente des speziellen Datentyps gibt es ein Wildcard, also ein Ersetzungssymbol (*), welches für einen beliebigen Inhalt steht. Man betrachte dazu folgende Beispiele:

⁸⁶ Dieser könnte eventuell im Falle eines Blattes als Gesamtergebnis dienen.

⁸⁷ Etwa polynomiell in der Eingabegröße oder Problemdimension.

⁸⁸ Im Moment ist dies der Fall, es sind aber nur die Wahrscheinlichkeiten 0 und 1 erlaubt.

⁸⁹ Die Zahlenkomponenten müssen nicht exakt gleich sein. Die Zahl der Suchmaske dient als untere Schranke.

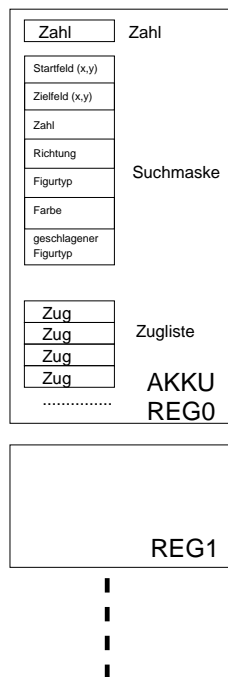


Abbildung 11: Registerstruktur



Abbildung 12: Datentyp der Züge und der Suchmaske

$\{(e, *); (*, *); *, *, \text{Springer}; \text{schwarz}; *\}$ repräsentiert alle Züge, die einen schwarzen Springer von der e-Linie wegbewegen. $\{(*, *); (*, *); *, *, *, \text{schwarz}; \text{Bauer}\}$ steht für alle Züge, die einen schwarzen Bauern schlagen.

Die durch die Suchmaske beschriebene Menge von legalen Zügen kann leer sein. Dies ist insbesondere dann der Fall, wenn sie widersprüchliche Angaben enthält.

Die in den Zuglisten enthaltenen Züge sind vom gleichen Typ wie die Suchmaske.

6.1.5.4 Konstanten

Konstanten haben in Stufe b eine nicht zu vernachlässigende Bedeutung. Es sind neun Konstanten für die Reihe (eine für das Wildcard) und ebenso viele für die Linie vorgesehen. Man beachte, dass diese keine wirklichen Zahlen sind, sondern abstrakte Objekte, die Ordnungsrelationen unterworfen sind (etwa „links von“). Dies ist wichtig, um die Symmetrie des Schachbretts zu bewahren.

Durch eine Konstante in Verbindung mit einem Wildcard als Tupelpartner, kann auch eine Linie oder Reihe beschrieben werden. Es gibt ferner neun Richtungskonstanten, sowie 6 Figurkonstanten in je zwei Farbausführungen (einschließlich Wildcard, insgesamt also weitere 13). Jede Konstantenbelegung soll mit gleicher Wahrscheinlichkeit realisiert werden.

6.1.5.5 Operationen

Operationen zur Steuerung des Programmablaufs

1. **b_ifThen** Die Operation realisiert eine bedingte Verzweigung: Ist der Akku größer als 0, dann wird in den rechten von zwei Pfaden verzweigt, ansonsten in den linken.
2. **b_callAdfA** Dem Aufruf von a-Programmen mit der aktuellen Brettstellung dient diese Operation. Die vom a-Programm zurückgelieferte Zahl wird im Zahlattribut des Akkus abgelegt.

Arithmetische Operationen

1. **b_add** Addition einer Konstanten zum Zahlattribut des Akkus. Das Ergebnis wird in ein von der Operation unabhängiges, beliebiges Register geschrieben, dessen Adresse übergeben wird.
2. **b_sub** Siehe **b_add**, die Konstante dient als Subtrahend.
3. **b_mul** Siehe **b_add**, die Konstante dient als Multiplikator.
4. **b_div** Siehe **b_add**. Die Konstante ist der Dividend. Sollte dieser 0 sein, wird die Operation ignoriert.

Mengenbasierte Operationen auf der Liste

1. **b_getAllMoves** Schreibt alle legalen Züge in die Zugliste eines beliebigen, aber wohldefinierten Registers.
2. **b_unionize** Vereinigung der Zugliste des Akkus mit der Zugliste eines übergebenen Registers. Das Ergebnis ersetzt die Liste des Akkumulators.
3. **b_intersection** Schnittmenge der Zugliste des Akkus mit der Zugliste eines übergebenen Registers. Das Ergebnis ersetzt die Liste des Akkumulators.
4. **b_difference** Differenz der Zugliste des Akkus und der Zugliste eines übergebenen Registers. Das Ergebnis ersetzt die Liste des Akkumulators.
5. **b_sortByValue** Die Zugliste des Akkus wird gemäß dem Zahlattribut absteigend sortiert.
6. **b_cardinality** Mächtigkeit der Zugliste des Akkus in das Zahlfeld des Akkus schreiben.
7. **b_subset** Filtert die Teilmenge der Züge im Akku heraus, welche dem, durch die Suchmaske angegebenen Attribut, entsprechen. Die Ergebnisliste ersetzt die ursprüngliche Liste des Akkus. Die Zahl der Zugliste wird ignoriert und grenzt somit keine Lösungen aus.

8. **b_subsetByBorder** Analog zu der Operation **b_subset**, bis auf die Tatsache, dass die Zahl als Merkmal des Attributs, in Form einer unteren Schranke Berücksichtigung findet.
9. **b_number** Anzahl der Züge in der Zugliste des Akkus mit dem durch die Suchmaske angegebenen Attribut in den Wert des Akkus schreiben.
10. **b_numberByBorder** Anzahl der Züge mit angegebenem Attribut, wobei zusätzlich die Zahl (als untere Schranke) berücksichtigt wird, in den Akku schreiben.
11. **b_set2Mask** Ermittlung einer passenden minimalen Suchmaske zu einer Zugliste (in etwa invers zu der Operation **subset**). Minimal bedeutet, dass so wenig Wildcards wie möglich benutzt werden, trotzdem aber die Maske die Zugliste vollständig überdeckt.
12. **b_incrementIfEqual** Falls die Zugliste des Akkus mit der Zugliste des übergebenen Registers übereinstimmt, wird die Zahl im Akku um 1 erhöht. Dabei spielt die Reihenfolge der Züge für die Gleichheit der Zuglisten keine Rolle. Diese Operationen könnte im Zusammenhang mit indirekten Lade- und Speicheroperationen Sinn machen.
13. **b_setAllValues** Setzt das Zahlattribut jedes Zuges der Zugliste eines bestimmten Registers auf den Wert, der im Zahlattribut desselben steht.
14. **b_increaseAllValues** Erhöht das Zahlattribut jedes Zuges der Zugliste eines bestimmten Registers um den Wert, der im Zahlattribut dieses Registers steht.
15. **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.
16. **b_loadMedianMoveValue** Setzt die Zahl im Akku-Zahlattribut auf den Median der in den Zügen der Zugliste vorkommenden Zahlen.
17. **b_loadMinMoveValue** Setzt die Zahl im Akku auf die niedrigste Zahl, die in der Zugliste vorkommt.
18. **b_loadMaxMoveValue** Setzt die Zahl im Akku auf die höchste Zahl, die in der Zugliste vorkommt.
19. **b_loadMoveValue** Setzt die Zahl im Akku auf die Zahl, die in dem i -ten Zug der Zugliste steht, wobei i aus dem Zahlattribut des Akkus genommen wird.

Lade- und Speicheroperationen

1. **b_loadValue** Dieser Befehl lädt das Zahlattribut eines Registers in das des Akkus.

2. **b_storeValue** Diese Operation schreibt das Zahlattribut des Akkus in das eines Registers.
3. **b_loadValueIndirect** Das Zahlattribut des Registers, dessen Adresse übergeben wird, wird in das Zahlattribut des Akkus geladen. Hier liegt eine indirekte Adressierung vor.
4. **b_storeValueIndirect** Diese Operation speichert das Akku-Zahlattribut indirekt.
5. **b_loadSearchmask** Diese Operation lädt die komplette Suchmaske aus einem Register in die Suchmaske des Akkus.
6. **b_storeSearchMask** Hier wird die Suchmaske des Akkus in einem Register gespeichert.
7. **b_loadSearchmaskIndirect** Hier erfolgt das Laden der Suchmaske eines Registers in der Suchmaske des Akkus indirekt. Die Adressierung erfolgt über das Zahlattribut des Akkus.
8. **b_storeSearchmaskIndirect** Hier erfolgt die Speicherung der Suchmaske des Akkus in der Suchmaske eines anderen Registers indirekt. Die Adressierung erfolgt über das Zahlattribut des Akkus.
9. **b_loadMoveList** Analog zu **b_loadValue**.
10. **b_storeMoveList** Analog zu **b_storeValue**.
11. **b_loadMoveListIndirect** Analog zu **b_loadSearchmaskIndirect**.
12. **b_storeMoveListIndirect** Analog zu **b_storeSearchmaskIndirect**.
13. **b_loadAll** Lädt ein ganzes Register in den Akku.
14. **b_storeAll** Speichert den ganzen Akku in einem anderen Register.
15. **b_loadAllIndirect** Indirekte Adressierung, analog zu **b_loadValueIndirect**.
16. **b_storeAllIndirect** Indirekte Adressierung, analog zu **b_storeValueIndirect**.
17. **b_swap** Tauscht die Inhalte von Akku und eines Register komplett gegeneinander aus.

Operationen zum Setzen von Konstanten und Sonstiges

1. **setAll2WildCard** Bis auf das Suchmasken-Zahlattribut werden durch diese Operation alle Attribute der Suchmaske eines Registers auf Wildcards gesetzt. Wildcards auf einzelne Attribute können mit den folgenden Operationen jedoch auch gesetzt werden.

2. `setSourceRow2WildCard`
3. `setSourceLine2WildCard`
4. `setDestinationRow2WildCard`
5. `setDestinationLine2WildCard`
6. `setDirection2WildCard`
7. `setPiece2WildCard`
8. `setHitPiece2WildCard`
9. `setColour2WildCard`
10. `setSourceRow` Dies setzt die Reihe des Startfeldes der Suchmaske eines Registers auf den übergebenen Wert. Die folgenden Operationen verlaufen analog.
11. `setSourceLine`
12. `setDestinationRow`
13. `setDestinationLine`
14. `setValue`
15. `setDirection`
16. `setPiece`
17. `setHitPiece`
18. `setColour`
19. `setSearchMaskValue`
20. `copyPiece2HitPiece` Dieser Befehl kopiert innerhalb der Suchmaske des Akkus den Figurtyp in das Typ-Feld für der geschlagenen Figur.
21. `copyHitPiece2Piece` Analog zu `copyPiece2HitPiece`.

6.1.6 Stufe c – die Auswahl eines Zuges

6.1.6.1 Grundgedanken

Evolvierte Programme der Stufe a sollen statische Bewertungen einer Brettstellung bezüglich zahlreicher Kriterien ermöglichen. Die der Stufe b liefern zu einer Stellung eine Zugliste, welche Züge enthält, die sich hinsichtlich eines Merkmals besonders eignen. Es sind unterschiedliche Merkmale vorgesehen, in die bereits dynamische Aspekte einfließen.

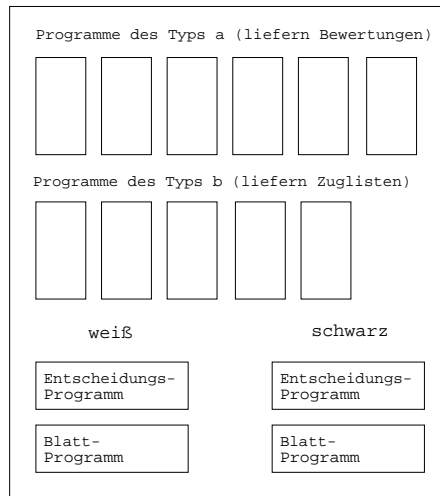


Abbildung 13: Aufbau eines Individuums der Stufe c

Insgesamt handelt es sich bei den Programmen nur um Bausteine, die, kombiniert mit einem Programm der Stufe c, kurz *c-Programm*, das Gesamt-Individuum ausmachen. Das Individuum besteht also aus einem c-Programm, sowie einigen a- und b-Programmen. In diesem Abschnitt werden die c-Programme beschrieben, sowie die Ausführung des gesamten Individuums erklärt.

Die Aufgabe dieser zusammengesetzten Individuen ist es, zu einer Stellung einen „guten“ Zug zu liefern. In Stufe c findet hierfür eine Baumsuche statt. Ausgangspunkt ist die gegebene Brettstellung, im folgenden auch mit *Wurzel* bezeichnet. Die von Programmen des Typs b gelieferten Listen legaler Züge werden zur Expansion des Baumes an einem Knoten benutzt. Dabei müssen nicht alle Züge ausgeführt werden, da ein vorzeitiger Abbruch erlaubt ist. Wird ein Knoten nicht expandiert, handelt es sich also um ein Blatt, so erfolgt eine Bewertung der erreichten Brettstellung. Dabei können mehrere Programme der Stufe a aufgerufen werden, um an der Berechnung teilzunehmen. Von den Blättern bis zur Wurzel erfolgt eine automatische Rückbewertung mit dem Minimaxverfahren.

Um eine clevere Baumsuche bereits mit kleinen Individuen zu ermöglichen, wird das c-Programm des Individuums rekursiv ausgeführt. Rekursion ist im Falle einer Baumsuche ein sehr natürliches Konzept, welches den GP-Individuen eine hohe Ausdruckskraft ermöglicht. Der rekursive Quantor wird hier auch mit *RFOR* bezeichnet. Insgesamt entscheidet das Individuum über die komplette Struktur der Baumsuche und die Art der Bewertung.

Die Ebenen eines Suchbaumes entsprechen abwechselnd dem weißen und schwarzen Spieler. Das Programm der Stufe c ermöglicht auf einfache Weise die Simulation zweier Spieler mit unterschiedlicher Strategie. Das Individuum enthält hierfür, neben Programmen vom Typ a und b, das eigentliche c-Programm, bestehend aus vier Modulen (Abbildung 13). Davon simulieren jeweils zwei einen Spieler. Jedem Spieler ist ein *Entscheidungs-Modul* und ein

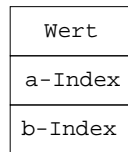


Abbildung 14: Datentyp der Operationen der Entscheidungs- und Blatt-Module

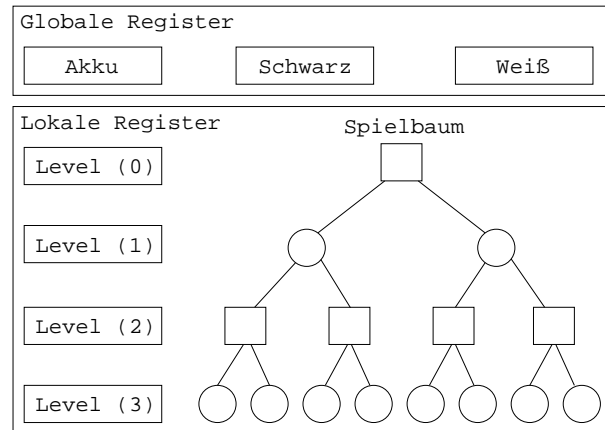


Abbildung 15: Die Register der Stufe c

Blatt-Modul zugeordnet.

Die Inhalte aller vier Module unterliegen der Evolution. Wie bereits erwähnt entstehen auch Programme vom Typ a und b auf diese Weise.

Bevor erläutert wird, wie mit den Modulen eine Baumsuche realisiert wird, erfolgt eine Beschreibung des zugrundeliegenden Datentyps, der Konstanten, sowie der Operationen, welche die Module benutzen dürfen.

6.1.6.2 Datentyp

Die Operationen der vier Module arbeiten mit einem speziellen Datentyp, der in Abbildung 14 dargestellt ist. Dieser enthält die drei Zahlen *Wert*, *a-Index* und *b-Index*. Letztere werden als Verweise auf ein Programm vom Typ a oder b des Individuums interpretiert. Auch wenn es sich stets um Zahlen handelt, wird in seltenen Fällen die Bezeichnung *Verweis* vorgezogen werden. Im folgenden werden die Objekte dieses Datentyps Register genannt. Die Zahl *Wert* eines Registers wird vereinfachend auch als Wert des Registers bezeichnet.

In Stufe c gibt es mehrere Register unterschiedlicher Bedeutung. Ein Überblick bietet Abbildung 15. Es gibt ein Register *Akku*, das von allen vier Modulen, also insbesondere von beiden bei der Baumsuche simulierten Spielern, benutzt werden darf. Zusätzlich gibt es für beide simulierten Spieler ein Register mit geschütztem Zugriff (*Schwarz*, *Weiß*). Darüberhinaus kann ein Speichern und Laden im Register-Array *Level* erfolgen, welches pro besuchte Ebene des Baumes

ein Register sichern kann.

6.1.6.3 Konstanten

Einige Operationen benutzen Konstanten. Sie werden bei der Initialisierung der Individuen zufällig gesetzt. Darüberhinaus kann die Mutation eine zufällige Änderung vornehmen. Ansonsten bleiben diese unverändert. Insbesondere belässt die Ausführung der Individuen diese Konstanten beim ursprünglichen Wert.

6.1.6.4 Operationen

Im folgenden werden die Operationen beschrieben, welche, wenn nicht anders erwähnt, von allen vier Modulen eines c-Programmes benutzt werden können. Das Akku-Register dient, wie der Name schon sagt, bei vielen Operationen als Akkumulator.

Operationen zur Steuerung des Programmablaufs

1. **callAdfA**: Ruft das Programm vom Typ a auf, auf welches in Akku durch a-index verwiesen⁹⁰ wird. Die berechnete Zahl wird im Wert von Akku abgelegt.
2. **ifThen**: Vorwärtsgerichteter bedingter Sprungbefehl. Ist der Wert in Akku positiv, springt die Ausführung des Modules konstant viele Zeilen weiter. Sind nicht genügend Zeilen vorhanden, so wird die Ausführung des Modules beendet.
3. **exit**: Stoppt die Ausführung des Entscheidungs-Modules und startet das Blatt-Modul. Diese Operation wird im Falle der Wurzel ignoriert. Sie darf nur in den Entscheidungs-Modulen benutzt werden.
4. **cut**: Stoppt die Ausführung des Entscheidungs-Modules, startet das Blatt-Modul und schneidet weitere Züge des Elterknotens ab. Diese Operation wird im Falle der Wurzel ignoriert. Sie darf nur in den Entscheidungs-Modulen benutzt werden.

Arithmetische Operationen

1. **addConst**: Addiert Konstante zum Wert von Akku.
2. **subConst**: Subtrahiert Konstante vom Wert von Akku.
3. **mulConst**: Multipliziert Wert von Akku mit Konstante.
4. **divConst**: Dividiert Wert von Akku durch eine Konstante. Ist diese 0 wird die Operation nicht durchgeführt.

⁹⁰ Die Zahl a-Index wird als Veweis auf das Programm vom Typ a der Nummer a-Index mod (Anzahl der Programme vom Typ a im Individuum) interpretiert.

5. `addColourConst`, `subColourConst`, `mulColourConst`, `divColourConst`, `addLevelConst`, `subLevelConst`, `mulLevelConst`, `divLevelConst`: Analog.
6. `addColourValue2Global`: Addiert den Wert des entsprechenden Farbregisters zu dem Wert von Akku. Simuliert das sich in Ausführung befindliche Blatt- oder Entscheidungsmodul den schwarzen Spieler, so handelt es sich um das Register Schwarz, ansonsten um Weiß.
7. `subColourValue2Global`: Subtrahiert den Wert des entsprechenden Farbregisters von dem Wert von Akku.
8. `mulColourValue2Global`: Multipliziert den Wert von Akku mit dem Wert des entsprechenden Farbregisters.
9. `divColourValue2Global`: Dividiert den Wert von Akku durch den Wert des entsprechenden Farbregisters, falls dieser ungleich 0 ist.
10. `addLevelValue2Global`, `subLevelValue2Global`, `mulLevelValue2Global`, `divLevelValue2Global`: Analog.

Lade- und Speicheroperationen

1. `storeValueInAdfA`: Kopiert innerhalb Akku den Wert nach a-Index.
2. `storeValueInAdfB`: Kopiert innerhalb Akku den Wert nach b-Index.
3. `loadValueFromAdfA`: Kopiert innerhalb Akku den a-Index in den Wert.
4. `loadValueFromAdfB`: Kopiert innerhalb Akku den b-Index in den Wert.
5. `storeColourRegister`: Kopiert Akku in das zur Farbe gehörende Register. Simuliert das sich in Ausführung befindliche Blatt- oder Entscheidungsmodul den schwarzen Spieler, so handelt es sich um das Register Schwarz, ansonsten um Weiß.
6. `storeAdfAInColourRegister`: Kopiert a-Index von Akku in a-Index des entsprechenden Farbregisters.
7. `storeAdfBInColourRegister`: Kopiert b-Index von Akku in b-Index des entsprechenden Farbregisters.
8. `storeValueInColourRegister`: Kopiert den Wert von Akku in den Wert des entsprechenden Farbregisters.
9. `storeLevelRegister`: Kopiert Akku nach Level.
10. `storeAdfAInLevelRegister`: Kopiert a-Index von Akku nach a-Index von Level.
11. `storeAdfBInLevelRegister`: Kopiert b-Index von Akku nach b-Index von Level.

12. `storeValueInLevelRegister`: Kopiert den Wert von Akku in den Wert von Level.
13. `loadColourRegister, loadAdfAFromColourRegister, loadAdfBFromColourRegister, loadValueFromColourRegister, loadLevelRegister, loadAdfAFromLevelRegister, loadAdfBFromLevelRegister, loadValueFromLevelRegister`: Die zu 5.–12. analogen Ladeoperationen.

Operationen zum Setzen von Konstanten und Sonstiges

1. `getDepth`: Ermittelt die aktuelle Tiefe im Suchbaum und speichert sie im Wert des Akku-Registers ab.
2. `setAdfA`: Setzt den a-Index in Akku auf eine konstante Zahl.
3. `setAdfB`: Setzt den b-Index in Akku auf eine konstante Zahl.
4. `setValue`: Setzt in Akku den Wert auf eine konstante Zahl.

6.1.6.5 Programmausführung (RFOR)

Programme vom Typ a und b, sowie die Blatt- und Entscheidungsmodule für jede Spielfarbe unterliegen der Evolution. Das Ausführungsgerüst RFOR ist allerdings vorgegeben.

Gegeben ist eine Brettstellung. Die Suche startet in dem entsprechenden Wurzel-Knoten. Im allgemeinen Fall erreicht die Suche einen neuen Knoten des Suchbaumes. Ohne Einschränkung sei weiß bezüglich des neu aufgefundenen Knotens am Zug, der andere Fall verläuft analog. Es beginnt die zeilenweise Ausführung des Entscheidungs-Moduls des weißen Spielers (siehe Abbildung 16). Es sei noch einmal darauf hingewiesen, dass das Entscheidungs-Modul durch Evolution entsteht. Das Entscheidungs-Modul entscheidet über den weiteren Verlauf der Suche. Entweder der Baum wird an diesem Knoten weiter expandiert, oder die Expansion wird abgebrochen. Im ersten Fall berechnet das Entscheidungs-Modul auch den Verweis⁹¹ b-Index auf ein Programm vom Typ b, welches nun gestartet wird, um die zur Expansion erforderliche Zugliste zu liefern. Der zweite Fall tritt genau dann ein, wenn bei der Ausführung des Entscheidungs-Moduls die Operation `cut` oder `exit` aufgetreten ist. In diesem Fall wird der Knoten als Blatt interpretiert und das Blatt-Modul zur Berechnung einer Bewertung der erreichten Brettstellung aufgerufen (siehe Abbildung 17). Diese Bewertung wird dann zurückgereicht, die Bearbeitung des Knotens ist beendet.

Hat sich das Entscheidungs-Modul vom weißen Spieler für eine Expansion des Knotens entschieden, so wird die Zugliste beginnend mit dem ersten Zug abgearbeitet. Der Zug wird ausgeführt und es erfolgt ein rekursiver Aufruf von RFOR für den erreichten Knoten (Brettstellung nach Ausführung des Zuges). Nun

⁹¹ Die Interpretation der Zahl b-Index als Verweis erfolgt analog zu der von a-Index.

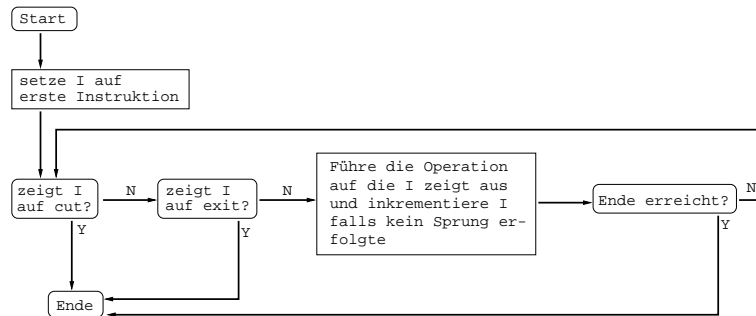


Abbildung 16: Ausführung eines Entscheidungs-Modules mit Instruktionszeiger I.

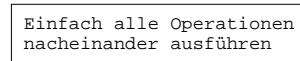


Abbildung 17: Ausführung des Blatt-Modules

wird selbstverständlich das schwarze Entscheidungs- und Blatt-Modul benutzt. Für den Fall, dass das Programm vom Typ b die leere Liste liefert, erfolgt ein Aufruf des Blatt-Modules des weißen Spielers um eine Bewertung zu berechnen, welche dann zurückgegeben wird; auch in diesem Fall ist die Bearbeitung des Knotens abgeschlossen.

Verursacht ein **cut** die vorzeitige Beendigung der Abarbeitung des Aufrufes, so erfolgt ein *abschneiden* der restlichen Zugliste. In diesem Fall gibt der Knoten den Minimaxwert bezüglich der ausgeführten Züge der Zugliste zurück und ist somit abgearbeitet.

Falls der Aufruf nicht von einem **cut** abgebrochen wird, erfolgt ein Aufruf für den nächsten Zug der Zugliste. Ist diese leer, wird der Minimaxwert bezüglich der Züge der Zugliste zurückgegeben.

Insgesamt findet eine Rückbewertung von den Blättern ausgehend, bis zur Wurzel statt, so dass an der Wurzel am Schluß der Minimaxwert bezüglich der Blätter steht. Der Gesamttablauf an einem Knoten ist nochmal in [Abbildung 18](#) dargestellt.

Folgende Ausnahmen sind vorgesehen:

1. Die Operationen **cut** oder **exit** werden für den Wurzelknoten ignoriert.
2. Liefert das Individuum keinen Zug zurück⁹², so wird der erste zulässige Zug genommen, den ein deterministischer, fest vorgegebener, Zuggenerator liefert, welcher kein Schachwissen⁹³ implementiert.

⁹² Programme vom Typ b können theoretisch immer die leere Liste zurückliefern.

⁹³ Der Generator betrachtet nacheinander alle Figuren, für jede dann jedes Feld, stets in der gleichen Reihenfolge. Er gibt genau die zulässigen Züge aus.

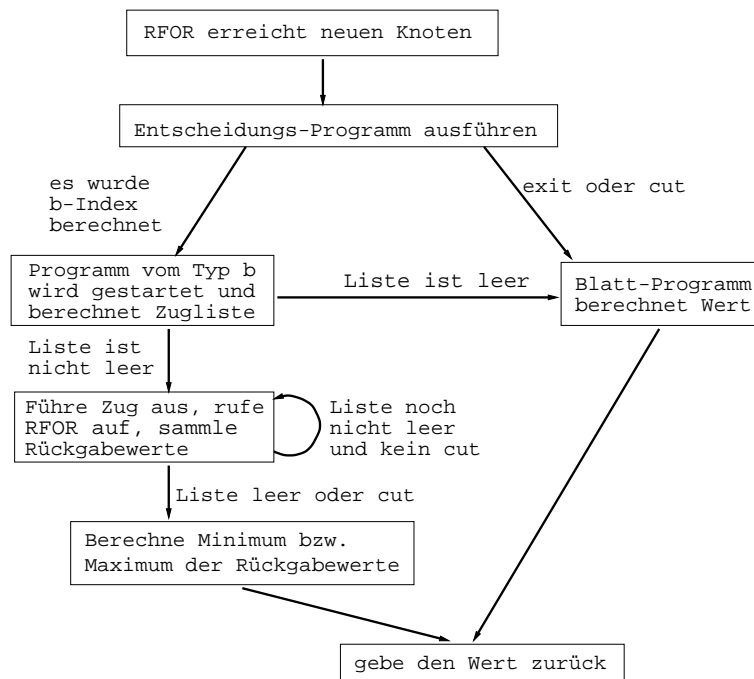


Abbildung 18: Das RFOR, dargestellt für einen Knoten

3. Üblicherweise gönnt man Individuen mit rekursiven Anteilen oder sonstigen möglichen Endlosschleifen (etwa bei Graph-GP) nur konstante oder zumindest endliche Ausführungszeit. Darüberhinaus erfolgt hier eine Festsetzung einer maximalen konstanten Suchtiefe durch das Individuum selbst. Entscheidend ist die Interpretation beim Überschreiten dieser. Eine Möglichkeit ist die folgende: Ein Überschreiten kommt einem `exit` gleich. Zusätzlich könnten nur konstant viele Überschreitungen zugelassen werden (danach könnte eine Rückbewertung erfolgen). Hier besteht noch Diskussionsbedarf.

Auch wenn zur Rückbewertung das Minimaxverfahren eingesetzt wird, unterscheidet sich die Suche von diesem deutlich. Die Unterschiede sollen hier noch einmal hervorgehoben werden:

1. Es werden nicht alle legalen Züge betrachtet. Vielmehr berechnet⁹⁴ das Entscheidungs-Modul mit dem `b-Index` einen in der jeweiligen Situation günstigen Zuggenerator. Zusätzlich können durch die Operationen `cut` oder `exit` Züge abgeschnitten oder nicht weiter verfolgt werden. Die Entscheidungs-Module und somit die Entscheidungs-Strategien werden evolviert. Programme vom Typ `b`, welche die geeigneten Zuglisten liefern sollen, werden auch evolviert. Sie liefern je nach Situation passende Zuglisten,

⁹⁴ Es kann zur Berechnung auch Programme vom Typ `a` aufrufen, welche eine Stellungsbewertung liefern.

insbesondere auch passender Zugsortierung und Länge.

2. Es findet nicht eine konventionelle Stellungsbewertung statt. Es wird auch nicht bloß ein evolviertes Programm befragt. Vielmehr entscheidet das Blatt-Modul, welche, von den nach unterschiedlichen Kriterien evolvierten Programmen vom Typ a, in welchem Maße an der Berechnung des Wertes teilhaben. Das Blattmodul wird evolviert, um die Entscheidungsfindung ebenfalls der Evolution zu überlassen. Es ist etwa möglich, dass eine gewichtete Summe berechnet wird, wobei die Gewichte sich bei der Abarbeitung des Baumes ändern können.
3. Die maximal erlaubte Suchtiefe ist nicht fest vorgegeben, sondern ein Attribut des Individuums, welches durch Mutation geändert werden kann. Diese Tiefe soll nur selten erreicht werden, da eine stark selektive Suche angestrebt wird.

6.1.7 Konzept Parallelisierung in der Stufe c

6.1.7.1 Grundlagen

Da der Bedarf an Rechenzeit für die Evolution Stufe c augenscheinlich immens ist und ein Pool mit 28 Sun-Workstations zur Verfügung steht, liegt es nahe, Berechnungen parallel durchführen zu lassen.

Der Großteil der benötigten Rechenzeit entfällt auf das Spielen von Schachpartien, also die Evaluation der Fitnesswerte. Die anderen Stationen der Evolutions-Hauptschleife können daher, was die Ausführungszeit betrifft, vernachlässigt werden und bedürfen der Parallelisierung durchaus nicht.

Als eine gute Strategie scheint es, den GP-Lauf auf einem einzigen festen Client-Rechner durchzuführen, dem die anderen 27 als Server bei der Berechnung der Fitnesswerte unter die Arme greifen. Man lasse sich hier nicht von der ungewöhnlichen 1:n-Client-Server-Relation verwirren.

Die ausgelagerten Entitäten sollen Schachpartien sein. Die als Server arbeitenden Rechner sollen zwei Individuen erhalten, die auf ihnen eine Partie austragen und dann das Ergebnis zurückmelden.

Alternativ dazu käme es ebenfalls in Betracht, die Parallelisierung auf der Ebene von einzelnen Zügen durchzuführen. Dann würde jeweils ein Individuen und eine Schachposition an den Slave-Rechner verschickt, und dieser lieferte dann einen auszuführenden Zug zurück. Der Zentral-Rechner wäre in diesem Fall auch mit der Durchführung der eigentlichen Partien betraut worden. Der Vorteil dieses Ansatzes wäre es, die Fitnessauswertung in feinerer Granularisierung durchzuführen, aber es träte ein erhöhter Kommunikationsbedarf auf.

Für die Idee der partienweisen Paketschnürung spricht neben dem geringeren Informationsaustausch auch die Umsetzung konsequenter Objektorientiertheit.

Partien sollen also dezentral auf vielen Servern gespielt werden, aber eine zentral auf genau einem Client verwaltete Elo-Zahl (3.6.2) haben, die den Fitnesswert bezeichnet.

6.1.7.2 Die Server

Das Elo-System (3.6.2) fußt auf der Ausgeglichenheit der Position, ab der gespielt wird. Niemand kann (derzeit) beweisen, daß die Grundposition ausgeglichen ist, aber jeder Schachspieler wird mit dieser Annahme leben können. Andernfalls wäre es erforderlich, zwei verschiedene Wertungszahlen für jeden Spieler zu berechnen. Die Projektgruppe wird mit der Schach-Grundstellung arbeiten.

Für eine Partie wird, das Schachspiel besitzt die Nullsummeneigenschaft, in der Summe genau ein Punkt an die beiden Kontrahenten ausgeschüttet. Der ermittelte und schließlich zurückgegebene Wert *whiteScore* bezeichnet das Spielergebnis aus der Sicht des weißen Spielers und entstammt dem Intervall $[0, 1]$. Ein *whiteScore* von 1 entspräche einem Sieg von Weiß, einer von 0 dementsprechend einem Verlust von Weiß. Der Erfolg der schwarzen Partei ergibt sich direkt aus $1 - \text{whiteScore}$.

Es wird die Design-Entscheidung getroffen, im Unterschied zur realen Schachpraxis auch andere, reellwertige Spielergebnisse zuzulassen. *whiteScore* kann im folgenden also einen beliebigen Wert aus dem Intervall $[0, 1]$ annehmen. Die dahinter stehende Idee ist es, durch die Hinzunahme von Partie-Sekundärmerkmalen (siehe 3.6.1) den Bewertungsprozess spürbar zu beschleunigen. Diese Beschleunigung erfolgte dadurch, aussagekräftige als nur ternäre Partieergebnisse zu verwenden und so mehr Information pro Partie zu gewinnen. Fernerhin wird die Fitnessbewertung auch durch den durch die möglich gewordene reellwertige Partieabschätzung praktikierbaren vorzeitigen Partieabbruch beschleunigt.

Aus der Schachpraxis sollen aber gewisse „Axiome“ in all ihrer Informalität übernommen werden:

- Remispartien sollen immer mit je einem halben Punkt für beide Parteien bewertet werden.
- Falls es einen Sieger gibt, soll dieser einen höheren Punktanteil erhalten als der Verlierer.
- Je überzeugender ein Sieg ausfällt, desto höher soll dieser bewertet werden.
- Matt soll immer mit einem ganzen Punkt belohnt werden.

Der erste und der letzte Punkt sind eindeutig zu implementieren.

Aus Gründen der Rechenzeit sollen Partien möglichst früh beendet sein oder abgebrochen werden. In der Arbeit der Projektgruppe wurde eine Partielänge von 70 Zügen (also 140 Halbzügen) als obere Schranke gewählt. Hatte die Partei bei Erreichen dieser Schranke noch keinen natürlichen Endzustand erreicht, soll sie abgebrochen und als Ergebnis den Parteien ihr jeweiliger Anteil am Gesamtmaterialwert der Schlußposition gutgeschrieben werden. Hätte beispielsweise Weiß nur noch eine Dame (Wert: neun Bauern) und Schwarz nur noch einen Läufer (drei), so erhielte Weiß $\frac{9}{12}$ und Schwarz $\frac{3}{12}$ Punkte.

Partien werden auch dann beendet, wenn über die zeitliche Distanz von mindestens vier Halbzügen der Spieler *A* eine „riesige“ Materialüberlegenheit besitzt, was als bei Erfüllung folgender Bedingung als gegeben angesehen wird:

$(Mat(A) > 2 * Mat(B) \wedge Mat(A) > 9) \vee Mat(A) - 9 > Mat(B)$, wobei $Mat(\text{Farbe})$ den Gesamtwert der Spielsteine einer Farbe in Bauerneinheiten bezeichnet. Schachgroßmeister kapitulieren teilweise im Minderbesitz von nur einem einzigen Bauern, und hier scheinen die geforderten neun Bauerneinheiten oder die doppelte Materialsomme bei recht vollem Brett angemessen.

In diesem Fall gab es abhängig von der Partielänge Bonuspunkte für schnelle Partie-Beendigung: $Score(\text{Sieger}) = 1 - \frac{moves-20}{200}$. Dies belohnte beispielsweise einen Beraubungssieg in 70 Zügen mit 0,75 Punkten und einen 22-zügigen mit 0,99.

6.1.7.3 Der Client

Dem Client obliegt es, Selektion, Rekombination und Mutation vorzunehmen, und überdies die Fitnessauswertung zu koordinieren. Letzterer Punkt sollte keinesfalls unterschätzt werden.

Es müssen die Gegner für ein Spiel „gewissenhaft“ ausgewählt und diese an einen anderen Rechner zwecks Austragung der angesetzten Partie verschickt werden. Diese Auswahl soll beispielsweise berücksichtigen, dass keine aktuell spielenden Individuen gelöscht und ersetzt werden. Fernerhin müssen vom Pool-Rechner laufend hereinkommende Partieergebnisse empfangen und bearbeitet werden.

Die entworfenen Algorithmen für alle benötigten Auswahlen seien im folgenden vorgestellt.

Verwendete Selektionsmechanismen

- Selektionsmechanismen für die Evolution

- Reproduktion

Eine Rekombination oder Mutation wird genau dann durchgeführt, wenn einerseits seit der Erzeugung des letzten Individuums zehn neue Partien beendet wurden und weiterhin sich mindestens eine vorgegebene Anzahl von ersetzbaren Individuen (gemeint sind solche mehr als sieben gespielten Partien) im Individuen-Pool befindet. Diese Anzahl wird als Parameter *replacementThreshold* im Strukturfile gesetzt und benutzt einen Defaultwert von fünf.

Für die Auswahl der Individuen zur Mutation und zum Crossover wird ein einfacher Selektionsmechanismus herangezogen:

Für eine Mutation kommen generell nur solche Individuen in Betracht, die bereits mindestens fünf Partien absolviert haben, damit sich unbedingt nur solche Individuen reproduzieren, bei man schon eine, wenn auch unsichere, Aussage über den Fitnesswert treffen kann. Es wird aus nach diesem Kriterium geeigneten Individuen ein Turnier zusammengestellt. Kommt ein solches Turnier mangels geeigneter Individuen nicht zustande, wartet der Selektionsprozess, bis genügend Partien zuende gespielt sind. Die Turniergröße ist auf 4 voreingestellt, kann aber über den Parameter *recombinationTournamentsize*

im Strukturfile der Stufe c gesteuert werden. Das Turnier gewinnt dasjenige Individuum mit der höchsten Elo-Zahl.

Die Selektion zur Rekombination wird obiger Mechanismus zweifach benutzt. Es ist möglich, dass zweimal dasselbe Individuum selektiert wird.

– Ersetzung

Unter den zur Ersetzung freigegebenen Individuen, also jenen, die bereits mehr als sieben Partien absolviert haben, werden diejenigen herausgefiltert, welche momentan nicht spielen. Unter diesen werden einige zufällig ausgewählt; die genaue Anzahl wird durch *replacement-Tournamentsize* in der Strukturdatei bestimmt. Das elo-niedrigste von diesen Individuen muss seinen Platz in der Population räumen und wird ersetzt.

Da nur nicht-spielende Individuen ersetzt werden können, muss darauf geachtet werden, dass Verhältnis von Individuen und gleichzeitig ausgetragenen Partien richtig anzusetzen.

Insgesamt wird immer nach zehn Partien ein neues Individuum produziert und ein altes dadurch ersetzt. Die Anzahl der gleichzeitig ausgetragenen Partien ist konstant, und es wird nach einer Einschwingphase genau dann ein neues Spiel gestartet, wenn ein altes beendet wird. Mittels dieser Heuristiken soll verhindert werden, dass der Individuenpool eine unerwünschte Eigendynamik entwickelt.

Die vorgestellten Selektionsmechanismen sind in ähnlicher Form bekannt aus anderen GP-Anwendungen und werden dort auch den an sie gestellten Anforderungen gerecht.

• Selektionsmechanismen für die Spielansetzung

Die in der Stufe c verwirklichte, komplizierte und dynamische Fitnessauswertung, die Individuen des Pools spielen schließlich *gegeneinander*, machte es erforderlich, zusätzliche Selektionsalgorithmen zu entwerfen, die dafür Sorge tragen, dass nur „sinnvolle“ Partien angesetzt werden, und beispielsweise der folgende Fall eben nicht eintritt: Das schlechteste Individuum spielt gegen das zweit schlechteste, und unmittelbar anschließend werden beide gelöscht.

– Erster Spieler

Es wird zunächst ein erstes Individuum gesucht. Mit einer Wahrscheinlichkeit von 90% ist dieses ein „junges“ Individuum, mit einer von 10% ein beliebiges. *Junge Individuen* sind solche, denen es noch an Spielerfahrung mangelt, um ersetzt werden zu können, die gleichsam noch in den Genuss eine „Schonzeit“ kommen.

Soll ein solches junges Individuum gewählt werden, so wird unter allen jungen zufällig eines gezogen. Bei selteneren freien Selektion wird ein

Turnier der Größe *selForGameTournamentsize* (default: 2) gebildet und der Sieger dort über die höhere Elo-Zahl bestimmt.

Nach diesem ersten Schritt steht ein Individuum fest. Dieses sei im folgenden durch *Indi₁* bezeichnet.

– Zweiter Spieler

Sodann wird ein passender Spielpartner für *Indi₁* gesucht. Dies geschieht anhand der Elo-Bewertung von *Indi₁* und der durchschnittlichen Elo-Zahl dessen bisherigen Gegner.

Zunächst wird eine Wunsch-Gegnerdurchschnittselozahl *Elo_Wunsch* für *Indi₁* berechnet. Dieser ergibt sich aus dem Mittel zwischen der Elo-Zahl von *Indi₁* und dem durchschnittlichen Elo-Wert des gesamten Pools. *Elo_Wunsch* motiviert sich dadurch, dass ein Individuum, um maximale Information aus seinen Spielergebnissen ziehen zu können, möglichst gegen etwa gleichstarke Kontrahenten antreten sollte. Die Anforderung „etwa gleichstark“ erweist sich jedoch bei angenommen normalverteilten Elo-Zahlen im Spielerpool gerade für sehr gute oder sehr schlechte Individuen als schwer realisierbar.

Zurück zur Selektion: Falls nun *Elo_Wunsch* größer als der durchschnittliche Elo-Wert der bisherigen Gegner ist, wird (falls möglich) dem ersten Individuum ein Gegner zugewiesen, dessen Elo-Zahl über dem durchschnittlichen Elo-Wert der bisherigen Gegner liegt. Sollte *Elo_Wunsch* kleiner sein als der durchschnittlichen Elo-Wert der bisherigen Gegner, so wird eben nach einem schwächeren Gegner gesucht. Die Idee hinter diesem nur kompliziert scheinenden Verfahren ist es, für n Partien, $n \rightarrow \infty$, die Zahl *Elo_Wunsch* an die tatsächliche Durchschnitts-Elozahl aller Gegner anzunähern.

Bei dieser Selektion wird dafür gesorgt, dass einem Individuum nicht etwa es selbst als Spielpartner zugewiesen wird.

Im Anschluss an die Selektion der beiden Spieler wird noch die Farbverteilung ausgelost, also bestimmt, welches Individuum in der anstehenden Partie Weiß und welches Schwarz haben wird.

6.2 Systemdesign

6.2.1 Grundlegende Überlegungen zum Systemdesign

In diesem Kapitel wird zum einen auf das Konzept der objektorientierten Programmierung (kurz OOP) eingegangen. Insbesondere werden die Vorteile der OOP gegenüber der traditionellen prozeduralen Programmierung dargestellt. Daneben werden die Grundgedanken zum Entwurf unseres Projektes vorgestellt.

6.2.1.1 Objektorientierte Programmierung

Vorteile der OOP

Die OOP ist eine grundsätzlich neue Art der Software-Entwicklung. Sie ist geeig-

net, auf hohem Abstraktionsniveau die Komplexität großer Projekte zu bewältigen. Bei OOP stehen *Objekte* im Mittelpunkt, also die Dinge, um die es bei der jeweiligen Problemstellung geht. Beispielsweise basiert ein Programm zur Realisierung eines GP-basierten Schachsystems u.a. auf Zügen. Dann besitzt das Objekt, das im Programm einen Zug repräsentiert, alle Eigenschaften und Fähigkeiten bzw. Attribute und Methoden, die zu einem Zug gehören. Die Attribute eines Zuges sind beispielsweise Start- und Zielfeld und der zu ziehende Stein, die Methoden sind u.a. Ausführen bzw. Zurücknehmen eines Zuges. Für die prozedurale Programmierung ist es charakteristisch, dass Daten und Funktionen, die diese Daten verarbeiten, keine Einheit bilden. Daraus resultiert eine größere Fehleranfälligkeit, z. B. durch fehlerhafte Zugriffe auf die Daten oder Verwendung nichtinitialisierter Daten. Hinzu kommt ein hoher Wartungsaufwand, z. B. wenn die Daten an neue Anforderungen angepasst werden müssen. Dagegen bilden die Objekte in OOP eine Einheit aus Daten (Eigenschaften) und Funktionen (Methoden). Die Daten werden zusammen mit den auf ihnen operierenden Funktionen implementiert. Daraus ergeben sich für die Software-Qualität entscheidende Vorteile:

- Höhere Zuverlässigkeit
- Geringerer Wartungsaufwand und
- Bessere Wiederverwendbarkeit.

Merkmale der OOP

Eine objektorientierte Programmiersprache, wie C++, ist dadurch gekennzeichnet, dass sie Sprachelemente zur Unterstützung folgender OOP-Paradigmen besitzt:

- Datenabstraktion
Es können Datentypen (Klassen) definiert werden, welche die Eigenschaften und Fähigkeiten von Objekten beschreiben. Dies ermöglicht das Handhaben komplexer Sachverhalte.
- Datenkapselung
Elemente eines Objektes können vor unkontrolliertem Zugriff von außen geschützt werden. Für die Kommunikation mit der Außenwelt, d. h. mit anderen Objekten, besitzt jedes Objekt eine öffentliche Schnittstelle. Über diese Schnittstelle können andere Objekte die Methoden des Objektes aufrufen, ohne dessen interne Struktur zu kennen.
- Vererbung
Neue Objekte lassen sich aus schon vorhandenen Klassen ableiten. Sie erben die vorhandenen Eigenschaften und Fähigkeiten, die ergänzt und verändert werden können. Die Vererbung ist ein wichtiges Mittel, um bereits entwickelten Programmcode wiederverwendbar zu machen.

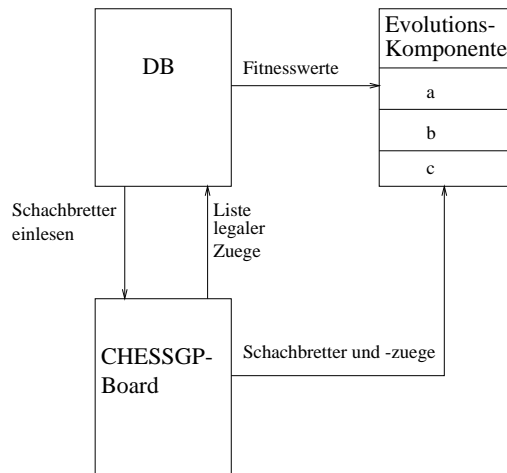


Abbildung 19: Das Drei-Säulen-Konzept

- Polymorphie

Bei der Erstellung eines Programms kann der genaue Typ eines Objektes noch unbekannt sein. Dann wird erst zur Laufzeit entschieden, welche Methode zur Ausführung herangezogen wird. Polymorphie (griech. Vielseitigkeit) bezeichnet die Möglichkeit, dass der Aufruf einer Methode zur Laufzeit verschiedene Wirkungen haben kann, je nach Typ des jeweiligen Objektes.

6.2.1.2 Grundgedanken des Entwurfs

Unserem Entwurf liegt ein Drei-Säulen-Konzept (Abb. 19) zugrunde. Es handelt sich dabei um drei Hauptkomponenten, aus denen unser Schachprogramm besteht:

1. **CHESSGP_Board**: Bereitstellung der grundlegenden Schachfunktionalität.
2. **Datenbank**: Speicherung von Schachstellungen und zugehörigen Bewertungen. Diese stellen die Fitnesscases unseres Evolutionssystems dar.
3. **Evolutionskomponente**: Kapselung der Evolutionsmodule, die mittels der SYSGP-Bibliothek realisiert werden.

Durch das Zusammenwirken dieser Komponenten wird die Evolution von schachspielenden GP-Agenten ermöglicht.

Aufgabe der Datenbank

Die DB kann in zwei Stufen unterteilt werden. Zum einen wird die DB dazu benutzt, Schachpositionen und die dazugehörigen statischen Bewertungen, z. B. die Materialbewertung zu einer gegebenen Schachstellung, abzuspeichern. Das Schreiben von Schachpositionen in die DB erfolgt mit Hilfe eines Parsers. Dieser

konvertiert gespielte, in PGN-Format vorliegende Schachpartien in eine DB-kompatible Form. Kompatibel wird hier in dem Sinne verstanden, dass die Schachpositionen in Form einzeliger Strings gespeichert werden. Genauer werden die Schachpositionen als maximal 64 Byte große Strings notiert. Der Aufbau der DB wird in Kapitel 6.2.3 beschrieben. Die verwendeten Partien entstammen einer *Golden Game Collection*, die die PG benutzt. Statische Bewertungsfunktionen, die in einer Vorverarbeitungsphase berechnet werden, bilden die Grundlagen für statische Fitnesswerte. Diese Fitnesswerte werden dann von der Stufe a der Evolutionskomponente benutzt. Im Kapitel 6.1.1 sowie im Kapitel 6.2.4 steht genaueres dazu.

Zum anderen werden in der DB dynamische Bewertungen gespeichert. Zu einer gegebenen Schachstellung wird eine Liste erlaubter Züge im Hinblick auf einen bestimmten Aspekt bewertet. Dazu wird zu jedem dieser Züge eine Fitnessbewertung abgespeichert, auf die von der Stufe b zurückgegriffen wird. Alle diese Datensätze sind für die Spielfähigkeit bzw. -stärke unseres Schachprogramms von Relevanz. Die DB wird also hauptsächlich dazu benutzt, die Stellungs- und Zugbewertungen während der Evolvierung zu verwalten.

CHESGP_Board:

CHESGP_Board stellt eine Schachbrettklasse mit der jeweils aktuellen Feldbelegung dar, die von allen drei Stufen der Evolutionskomponente benutzt werden kann. Dazu kann CHESGP_Board die in der DB gespeicherten Schachbretter einlesen. Außerdem kann zu der aktuellen Brettposition eine Liste aller legalen Schachzüge ermittelt werden, die dann auch ausgeführt und wieder zurückgenommen werden können. Insbesondere werden Schlagzüge, Rochade, en Passant und die Umwandlung eines Bauers in eine Dame berücksichtigt. Von besonderer Bedeutung ist die Funktionalität dieser Klasse in der Stufe b, wo sie die Kandidatenzuglisten erstellt.

Evolutionskomponente

Unser Ansatz dazu besteht wiederum aus drei Stufen. In Stufe a werden Stellungsbewertungsfunktionen realisiert. In Stufe b werden Zuglisten für jeweils verschiedene Bewertungskriterien generiert, die in Stufe c benutzt werden können. Schließlich wird in Stufe c eine Baumsuche durchgeführt, die als Ergebnis einen legalen Schachzug liefert. Siehe auch Kapitel 6.1 Höhere Stufen können auf tiefere zugreifen. Ein Individuum der Stufe c beinhaltet Instanzen der Stufen a und b. Dieses Individuum enthält als ADF's mehrere Stellungsbewertungsfunktionen der Stufe a sowie Zuglistenlieferanten der Stufe b. Zur Realisierung der Evolutionskomponente wird SYSGP eingesetzt. Dabei muss SYSGP an unser Konzept angepasst werden. Insbesondere müssen *Crossover*, *Mutation* und Interpreter an die Individuentypen angepasst werden. Für den Interpreter der Stufe c wurde das RFOR implementiert. Dabei handelt es sich um einen rekursiven Quantor, der eine clevere Baumsuche ermöglicht, da er den GP-Individuen eine hohe Ausdruckskraft verleiht.

6.2.2 Realisierung

Nachdem in den vorangegangenen Kapiteln das theoretische Grundkonzept unserer Überlegungen dargelegt wurde, werden nun die praktischen Aspekte behandelt. Im Folgenden werden unsere Implementierungen der Klassen der Stufen a, b und c beschrieben. Desweiteren wird das Datenbankschema erläutert, das den Fitnessbewertungen der ersten beiden Stufe zugrunde liegt. Im letzten Teil dieses Abschnittes wird eine typische Lernphase eines Agenten der Stufe a beschrieben, die in der dargestellten Form auch auf Stufe b und – erweitert durch eine parallele Verteilung – Stufe c übertragbar ist.

6.2.2.1 Klassenbeschreibung

Die Abbildungen 20, 21 und 22 zeigen die Klassendiagramme unserer Umsetzung. Einige Klassen, die der SYSGP-Umgebung zuzuordnen sind und in keinem direkten Kontakt zu unseren eigenen stehen, wurden übersichtshalber entfernt. Alle unsere Klassen beginnen zur Abgrenzung zu SYSGP mit der Kennung `CHESSGP_`. Die meisten Klassen wurden mit überladenen Ausgabeoperatoren (`<<`) versehen, um Ergebnisse als Text ausgeben zu können. Klassen, die innerhalb eines Individuums verwendet werden (z.B. die Register) verfügen desweiteren über Einleseoperatoren (`>>`), damit evolvierte Individuen in den Folgestufen eingelesen werden können (z.B. werden Individuen als ASCII-Text zwischen den verschiedenen Stufen „ausgetauscht“).

Folgend werden zuerst die einzelnen von uns implementierten Klassen beschrieben und danach deren Zusammenspiel genauer erläutert. Die Aufzählung orientiert sich an unserem Stufenkonzept, wobei zuerst global verfügbare Klassen, die in allen drei Stufen verwendet werden, aufgeführt werden und danach die Klassen der Stufen a bis c folgen.

- **CHESSGP_Error:**

`CHESSGP_Error` dient uns als allgemeine Fehlerklasse, die für eine einheitliche Fehlerbehandlung (und deren Ausgabe) implementiert wurde.

- **CHESSGP_FitnessArray:**

Diese Klasse stellt ein Array aus zufälligen Zahlen zur Verfügung. Sie wird verwendet, um zu gewährleisten, dass Individuen einer Generation auf die selben Fitnesscases zugreifen. Es wurden Hilfsmethoden implementiert, um je nach Bedarf nur eine bestimmte Anzahl (oder einen Prozentsatz) von Fitnesscases durch neue zu ersetzen.

- **CHESSGP_Environment:**

Das `Environment` stellt die Umgebung dar, in der die Individuen aller Stufen ausgeführt werden. Die Klasse beinhaltet die Register, die von den Individuen zur Ablage von Ergebnissen benutzt werden. Dabei kommt dem Akkumulator als speziell ausgezeichnetes Register eine besondere Bedeutung zu, da er an den meisten Operationen beteiligt ist. Aus ihm werden

oftmals Werte ausgelesen oder zwischengespeichert – auch die Fitness eines Individuums werden in den Stufen a und b aus dem Akku entnommen. Desweiteren wird das jeweilige Schachbrett, auf dem „gespielt“ wird, verwaltet.

- **CHESSGP_RegisterElement:**

Diese Klasse wird als eine Art „Wrapper“ im Programm eingesetzt. Sie beinhaltet Zeiger und Zugriffsmethoden auf Registerelemente der Stufen a, b und c. Diese Form wurde gewählt, da es SYSGP nicht ermöglicht, Registerelemente unterschiedlichen Typs im Environment zu verwalten.

- **CHESSGP_RegisterElementA:**

Diese Klasse enthält die Daten und Methoden für ein Registerelement der Stufe a. Darunter fallen Angaben über die Reihe, Spalte und Farbe des zu betrachtenden Feldes sowie einem Wert, der bei der Fitnessberechnung als Fitnesswert interpretiert wird (siehe auch Kapitel 6.1.4.2).

- **CHESSGP_RegisterElementB:**

Diese Klasse enthält die Daten und Methoden für ein Registerelement der Stufe b. Darunter fallen Angaben zur Suchmaske, eine Zugliste, sowie ein als Fitness interpretierter Wert (siehe auch Kapitel 6.1.5.3).

- **CHESSGP_RegisterElementC:**

Das Registerelement der Stufe c beinhaltet die Werte der jeweils gesetzten Individuen der Stufe a und b. Desweiteren verfügt es über einen Wert, der bei der Berechnung des besten Zuges verwendet wird (siehe auch Kapitel 6.1.6.2).

- **CHESSGP_Board:**

Diese Klasse stellt im Wesentlichen das Schachbrett dar, enthält also alle Schachfiguren und deren Positionen. Desweiteren werden durch sie auch zu setzende Züge realisiert. Dabei „merkt“ sich **CHESSGP_Board** z.B. die Position von König und Turm (um die Rochade-Möglichkeit zu erfassen) oder En-Passant-Bauern. Auch liefert die Klasse alle möglichen Züge aus Sicht eines der beiden Spieler. Diese werden beispielsweise bei der Fitnessbewertung in Stufe b benötigt. Weitere Methoden werden von den Operationen in **functionA.hh** bzw. **functionB.hh** verwendet. Eine Ausgabe des Schachbrettes als ASCII-Text und im \TeX -Format wurde zur einfacheren Überprüfung implementiert.

- **CHESSGP_AbstractFitnessA:**

Dies ist die abstrakte Oberklasse aller Fitnessfunktionen der Stufe a. Die Klasse sollte für jede Fitnessfunktion der Stufe a abgeleitet und die Funktion **normalize()** konkretisiert werden. Allerdings hat sich die Notwendigkeit für normalisierte Fitnesswerte nicht ergeben, so dass diese Möglichkeit

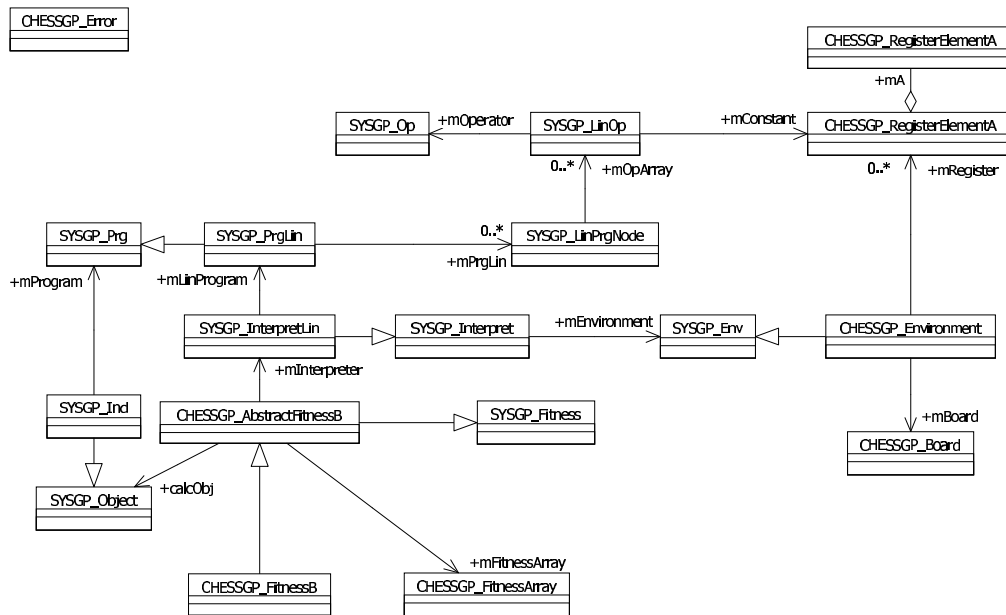


Abbildung 20: Klassendiagramm der Stufe a

nicht ausgenutzt wurde. Die Methode `calc()` implementiert die Fitnessberechnung. Dabei werden die Werte aller Fitnesscases (also aller zu betrachtenden Schachbretter), die zuvor in die Datenbank gespeichert wurden, mit den (Akku-)Werten der Individuen abgeglichen.⁹⁵

- **CHESSGP_FitnessA:**

Diese Klasse implementiert lediglich die Normierungs-Methode der Fitness-Klasse der Stufe a. Sie wird allerdings nicht verwendet (bzw. der Wert wird ohne eine Normierung durchzuführen zurückgegeben).

- **CHESSGP_AbstractFitnessB:**

Dies ist die abstrakte Oberklasse aller Fitnessfunktionen der Stufe b. Die Klasse sollte für jede Fitnessfunktion der Stufe b abgeleitet und die Funktion `normalize()` konkretisiert werden. Allerdings wird die Normalisierung der Fitness nun durch die Klasse `CHESSGP_MoveListDB` realisiert, so dass diese Möglichkeit nicht ausgenutzt wurde. Die Funktion `calc()` implementiert die Fitnessberechnung in ähnlicher Weise, wie in Stufe a. Zusätzlich wurde zur Effizienzsteigerung noch die Funktion `calcX()` hinzugefügt, die auch Verwendung findet. Die Fitness ergibt sich aus dem Abgleich der zuvor in der Datenbank gespeicherten Züge mit den vom Individuum ermittelten Zügen.

⁹⁵ Der Fitnesswert ergibt sich als Summe aller quadratischen Differenzen der Fitnesswerte aus der Datenbank und des Individuums. Der so ermittelte Wert über alle Fitnesscases wird danach auf das Intervall $[-1.000.000; 1.000.000]$ beschränkt und durch die Anzahl der betrachteten Fitnesscases geteilt.

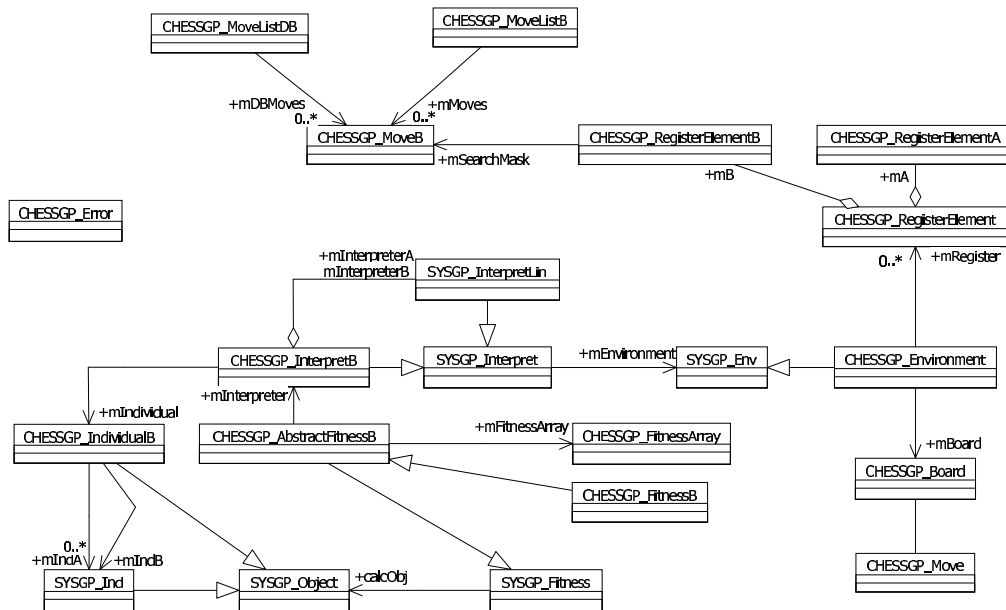


Abbildung 21: Klassendiagramm der Stufe b

- **CHESSGP_FitnessB:**

Diese Klasse implementiert lediglich die Normierungs-Methode der Fitness-Klasse der Stufe b. Sie wird allerdings nicht verwendet, da die Normierung durch die Klasse `CHESSGP_MoveListDB` realisiert wird.

- **CHESSGP_MoveB:**

Diese Klasse repräsentiert einen Zug für Stufe b. Die Registerelemente von Stufe b enthalten eine Zahl, eine Zugliste (von Objekten dieser Klasse) und eine Suchmaske, ebenfalls als Objekt dieser Klasse. Während in der Suchmaske auch Wildcards vorkommen können, müssen die Operationen und die Initialisierung sicherstellen, dass die Zuglisten (vom Typ `CHESSGP_MoveListB`) komplett ausformuliert sind.

- **CHESSGP_MoveListB:**

Objekte dieser Klasse enthalten eine Zugliste, welche ein Baustein der Klasse `CHESSGP_RegisterElementB` der Stufe b ist. Desweiteren sind einige Methoden implementiert, die durch Operationen in `functionB.hh` genutzt werden können. Darunter fallen unter anderem Mengenoperationen wie Vereinigung und Durchschnitt und Methoden zur Bestimmung des kleinsten oder größten Wertes eines Zuges.

- **CHESSGP_MoveListDB:**

Diese Klasse wird zur Fitnessberechnung der Individuen der Stufe b eingesetzt und nimmt die in der Datenbank gespeicherten Schachzüge auf. Mit

- **CHESSGP_Move:**

Diese Klasse repräsentiert in der Stufe c einen vorzunehmenden Zug. Sie ist Bestandteil der Klasse **CHESSGP_MoveListC**.

- **CHESSGP_MoveListC:**

Objekte dieser Klasse enthalten eine Zugliste, welche ein Baustein der vom Interpreter vorgenommen Baumexpansion für Stufe c ist. Eine Zugliste entspricht einem Knoten dieses Baumes, daher kann auch deren Farbe gespeichert werden. Ein Zeiger verweist auf ein aktuelles Element. Es wird eine Zahl zur Verfügung gestellt, um den Minimaxwert des Knotens, zu der die Zugliste gehört, zu berechnen. Darüberhinaus wird der bisher beste Zug gleich mit abgespeichert. Objekte vom Typ **CHESSGP_MoveListC** sollen mit dem Konstruktor immer auch einen Vektor von Zügen übergeben bekommen. Direkt danach soll die Farbe extra gesetzt werden. Nur so wird die Konsistenz der Daten gewährleistet. Im Allgemeinen soll man den Typ **vector** nicht zur Listenbildung verwenden. Da wir aber Operationen, wie Einfügen, insbesondere im Inneren einer Liste, nicht benötigen, reicht der Typ vollkommen aus.

- **CHESSGP_TreeSearch:**

Diese Klasse benötigt der Interpreter zum Aufspannen des Baumes in Stufe c. Auf einfache Weise kann Minimax implementiert werden. Objekte dieser Klasse enthalten einen Stack von Zuglisten, mit denen ein Suchbaum aufgespannt werden kann. Die Listen werden durch Aufrufe der Stufe b erzeugt und auf den Stack gelegt (push). Ist ein Knoten fertig abgearbeitet, nimmt man seine Liste vom Stack herunter (pop). Den Minimaxwert des aktuellen Knotens kann man mit `getValue()` lesen (dieses geschieht zum Ende an der Wurzel). Er ist dann korrekt, wenn alle Züge bereits ausgeführt wurden, also am Ende der Knotenabarbeitung. Die Blattbewertung muss mit `setValueOfMove()` explizit vorgenommen werden. Die Werte werden automatisch hochgereicht.

- **CHESSGP_RFOR:**

Diese Klasse implementiert das bereits in Abschnitt 6.1.6.5 vorgestellte iterative Lernverfahren der Stufe c. Die Funktion `run()` bekommt einen Interpreter der Stufe c übergeben, in dem das auszuführende Individuum gesetzt sein muss, und liefert den vom Individuum berechneten Schachzug zurück. Verschiedene Methoden ermöglichen es, die Ausführung eines Individuums zu beeinflussen, indem z. B. die Suchtiefe oder die Anzahl zu betrachtender Knoten eingeschränkt wird.

- **CHESSGP_SFOR:**

Mit dieser Klasse können verschiedene Baumsuchen realisiert werden. Diese sind zum Beispiel AlphaBeta, MiniMax oder verschiedene NegaScout-

Verfahren. Zum einen findet sie Verwendung bei den Bewertungen der Stufe b. Zum anderen werden die genannten Suchverfahren für Individuen der Stufe c eingesetzt, die nicht als GP-Individuen arbeiten.

- **CHESSGP_Pool:**

Diese Klasse beinhaltet die Hauptschleife der (parallelen) Evolution in Stufe c, die durch den Aufruf der Methode `poolLoop()` gestartet werden kann. Die Überlegungen, die sich an Kapitel 6.1.7 orientieren, sind hier realisiert. Dabei werden neue Spiele mittels `pvm` bzw. der Klasse `PVM_Pool` auf verschiedenen Rechnern parallel gestartet, so dass die (hohe) Rechenzeit kein Einzelsystem und somit die Evolution blockiert. Neue Spiele werden dabei als `CHESSGP_Match` in `CHESSGP_MatchSet` festgehalten und nach Beendigung dort wieder gelöscht. Sowohl die Spieler- als auch die Ersetzungsselektion wird innerhalb verschiedener Methoden realisiert.

- **CHESSGP_PoolStability:**

Die Klasse `CHESSGP_PoolStability` wird in `CHESSGP_Pool` verwendet, um eine bestimmte Anzahl gespielter Spiele zu beobachten und die Rate der erfolgreich beendeten Spiele zu bestimmen.

- **CHESSGP_Player:**

Diese Klasse lässt, nachdem die Operationsmenge und das Strukturfile angegeben wurde, zwei spezifizierte Computerindividuen gegeneinander ein Spiel spielen. Das geschieht durch Aufruf der Funktion `play()`, die das Ergebnis (aus Sicht des weißen Spielers) als Zahl zwischen 0 und 1 zurückgibt, wobei 1 als Sieg für Weiß gewertet wird.

- **PVM_Pool:**

Diese Klasse regelt die Spielkommunikation mit `pvm` und `CHESSGP_Pool`. Es wird eine Instanz von `PVM_Pool` in `CHESSGP_Pool` erzeugt und über diese Klasse mehrere Spiele gestartet (`startGame()`).

- **PVM_Player:**

Diese Klasse regelt die Spielkommunikation mit `pvm` und `CHESSGP_Player`. Die Methode `start()`, die vom `PVM_Pool` aufgerufen wird, legt einen neuen `CHESSGP_Player` an, initialisiert diesen und lässt ihn ein neues Spiel beginnen.

- **CHESSGP_Game:**

Die Klasse organisiert mittels Aufruf der Funktion `playGame()` eine Schachpartie, in der sowohl Individuen der Stufe c als auch menschliche Spieler gegeneinander antreten können. Zurückgeliefert wird der Ausgang des Spiels (aus Sicht des weißen Spielers) als Zahl zwischen 0 und 1, wobei 1 als Sieg für Weiß gewertet wird. Außerdem wird die Klasse zur Fitnessbewertung des sequentiellen Evolutionsprozesses in Stufe c genutzt. Ein Partie kann mittels der Methode `printPGN()` in PGN-Format ausgegeben werden.

- **CHESSGP_Match:**

Die Klasse `CHESSGP_Match` beinhaltet Charakteristika eines Spiels. Es wird festgehalten, welche Individuen am Spiel beteiligt sind und zu welchem Zeitpunkt das Spiel begonnen wurde. Der Zeitpunkt wird angegeben, indem die Anzahl schon begonnener Spiele bis zum Beginn des betrachteten Spiels gemerkt wird. Die Identifikation eines Spiels wird über die `pvm`-Id vorgenommen. Die Klasse wird von `CHESSGP_MatchSet` und `CHESSGP_Pool` verwendet, letzter erstellt neue Spiele und sichert diese bis zur Beendigung in einem `MatchSet`.

- **CHESSGP_MatchSet:**

Die Klasse `CHESSGP_MatchSet` verwaltet die momentan gespielten Spiele. Das Hinzufügen und Entfernen eines Spiels muss explizit angestoßen werden. Beim Entfernen werden die Charakteristika des angegebenen Spiels in einem `CHESSGP_Match`-Objekt zurückgegeben. Die Identifikation eines Spiels erfolgt über die `pvm`-Id.

- **CHESSGP_FitnessC:**

Diese Klasse sollte eigentlich die gleichen Aufgaben wie die entsprechenden Klassen der Stufe a und b wahrnehmen. Allerdings werden die Fitnessveränderungen nun von den Individuen selbst vollzogen, so dass diese Klasse nicht mehr benötigt wird.

- **CHESSGP_IndividualC:**

Diese Klasse beinhaltet neben den in Kapitel 6.1.6 beschriebenen Blatt- und Entscheidungsmodulen auch die Individuen der Stufe a und b, welche bei der Evolution mitverändert werden können. Desweiteren merkt sich ein Individuum sowohl die eigene Fitness (als ELO-Zahl), als auch die durchschnittliche Fitness seiner Gegner. Darüberhinaus weiß ein Individuum auch, an wie vielen Spielen es aktiv teilnimmt und wieviele es bereits beendet hat. Mit diesen Angaben kann in `CHESSGP_Pool` eine effektive Spieler- und Ersetzungsslektion durchgeführt werden.

- **CHESSGP_InterpreterC:**

Diese Klasse stellt den Interpreter der Stufe c dar. Es handelt sich dabei lediglich um eine Wrapper-Klasse, die sowohl Interpreter für das Blatt- und für das Entscheidungsmodul besitzt, als auch Interpreter für die Individuen der Stufe a und b beinhaltet. Letztere werden zur Ausführung von Individuen der Stufe a bzw. b benötigt, da diese von den Individuen der Stufe c aufgerufen werden können.

Zusammenspiel der Klassen

Nachdem die einzelnen Klassen nun vorgestellt wurden, soll in diesem Abschnitt erläutert werden, wie diese zusammenarbeiten. Ausgehend von den Kapiteln 6.1.4, 6.1.5 und 6.1.6 werden die dort beschriebenen Ideen der drei Stufen umgesetzt.

- *Globale Klassen:* Im Mittelpunkt der Evolution aller Klassen steht die Klasse `CHESSGP_Board`, da sich das eigentlich Schachspiel hier abspielt. Diese Klasse verwaltet neben den einzelnen Schachfiguren und deren Positionen auch Zusatzinformationen über den Verlauf des Spiels, hält also zum Beispiel fest, ob ein König noch eine Rochade vollziehen kann oder ob ein Bauer ein En-Passant-Bauer darstellt. `CHESSGP_Board` stellt verschiedene Schnittstellen bereit, die in den drei Stufen Verwendung finden. So kann über die Methode `getAll()` eine Liste aller gültiger Züge bestimmt werden, die den Individuen der Stufe b als Grundlage dienen. Mit `readString()` können Schachbretter, die als ASCII-Text kodiert in der Datenbank liegen, eingelesen und somit eine Brettstellung erzeugt werden. Auch werden verschiedene Operationen der Individuen (siehe Kapitel 6.1.4.4, 6.1.5.5 und 6.1.6.4) mit Hilfe von Methoden des Schachbretts realisiert.⁹⁶ Das Schachbrett stellt somit eine wesentliche Klasse dar, auf dem das gesamte Schachspiel abläuft. Daher ist das Schachbrett auch Bestandteil der Klasse `CHESSGP_Environment`, die wiederum zentraler „Ausführungsort“ der Evolution ist. Daneben umfasst das *Environment* die Registerelemente der Stufen a bis c⁹⁷, die Zwischen- und Endergebnisse der Fitnessberechnungen enthalten.
- *Stufe a:* Individuen der Stufe a stellen sich als lineare Programme dar. Diese werden von dem von SYSGP mitgelieferten *Interpreter* `SYSGP_InterpreterLin` ausgeführt. Dabei wird auf den im *Environment* befindlichen Registerelementen vom Typ `CHESSGP_RegisterElementA` gearbeitet. Diese beinhalten die in Abschnitt 6.1.4.2 beschriebenen Werte für Position und Art einer Figur, sowie eines zusätzlichen Wertes, der zur Fitnessberechnung benutzt wird. Die Fitnessberechnung wird durch die Klasse `CHESSGP_AbstractFitnessA` bzw. `CHESSGP_FitnessA` durchgeführt. Letztere beinhaltet eine Normierungsfunktion, die für jeden Individuentyp der Stufe a implementiert werden muss. Diese greifen auf die in die Datenbank eingetragenen Bewertungen (siehe auch Abschnitt 6.2.4) zurück und gleichen die dort abgelegten Werte mit denen ab, die die Individuen der Stufe a nach ihrer Ausführung im *Interpreter* zurückliefern⁹⁸.
- *Stufe b:* Individuen der Stufe b stellen sich als lineare Programme dar. Diese werden von dem von SYSGP mitgelieferten *Interpreter* `SYSGP_InterpreterLin` ausgeführt. Allerdings benötigen wir in Stufe b ein Individuum, das zusätzlich Individuen der Stufe a aufnehmen oder zumindest aufrufen kann. Dieses ist durch die Klasse `CHESSGP_IndividualB` realisiert worden, die neben dem eigenen linearen Programm auch einen Verweis auf ein Feld von 8 Individuen der Stufe a aufnehmen und diese aufrufen kann. Der

96 So realisiert die Methode `getPiece()` zum Beispiel die Operation *whichPiece* der Individuen der Stufe a.

97 Diese befinden sich wiederum gekapselt in der Klasse `CHESSGP_RegisterElement`.

98 Tatsächlich liefern die Individuen keine Fitnesswerte zurück, sondern der Wert im speziellen Akku-Register des *Environments* wird als Fitnesswert des jeweiligen Individuums angesehen.

entsprechende *Interpreter* wurde ebenfalls angepasst und findet sich in der Klasse `CHESSGP_InterpreterB` wieder. Die Registerelemente der Stufe b beinhalten die in Kapitel 6.1.5.3 beschriebenen Werte für Suchmaske und Zugliste, sowie einen zusätzlichen Wert, der für verschiedene Berechnungen eingesetzt wird. Die Klasse `CHESSGP_MoveB` stellt einen Zug dar, der Werte für einen auszuführenden Spielzug beinhaltet (unter anderem Start- und Endposition der zu ziehenden Figur). Als Suchmaske eingesetzt, kann `CHESSGP_MoveB` auch Wildcards beinhalten. Eine Zugliste wird durch die Klasse `CHESSGP_MoveListB` realisiert, welche eine Liste von Elementen der Klasse `CHESSGP_MoveB` beinhaltet. Neben Methoden, die der Listenverwaltung dienen (Zug einfügen etc.), wurden auch Methoden implementiert, die durch Operationen der Individuen (siehe Kapitel 6.1.5.5) ausgeführt werden können (Mengenoperationen). Die Fitnessberechnung der Individuen der Stufe b wird mit Hilfe der Klasse `CHESSGP_AbstractFitnessB` bzw. `CHESSGP_FitnessB` realisiert. Letztere beinhaltet eine Normierungsfunktion, die für jeden Individuumstyp der Stufe b implementiert werden muss. Zur Fitnessbestimmung wird ein Listenabgleich zwischen den Zügen, die ein Individuum berechnet, und den Zügen, die zuvor in der Datenbank abgelegt wurden, durchgeführt. Die Klasse `CHESSGP_MoveListDB` wird dabei verwendet, um die Züge aus der Datenbank auszulesen, die Methode `compareList()` übernimmt den Listenabgleich.

- *Stufe c*: Das Individuum der Stufe c beinhaltet neben dem Blatt- und Entscheidungsmodul für Weiss und Schwarz auch noch jeweils „eigene“ Individuen der Stufe a und b. Diese können und werden in die Evolution der Stufe c einbezogen werden, so dass jedes Individuum der Stufe c unterschiedliche a- und b-Individuen aufweisen wird. Individuen werden nicht direkt im Interpreter der Stufe c ausgeführt, sondern Schrittweise über RFOR kontrolliert. RFOR sorgt für die Interpretation der Individuen und liefert nach Beendigung eine Zug (`CHESSGP_Move`) zurück, der durch das Individuum als bester Zug zur jeweils gegebene Stellung bestimmt wurde. Dabei wird allerdings auf die Fähigkeiten des Interpreters zurückgegriffen, der für die Entscheidungs- und Blattmodule und auch für die a- und b-Individuen jeweils getrennte Interpreter beinhaltet. Der RFOR-Aufruf wird in `CHESSGP_Game` gemacht, welches wiederum durch einen `CHESSGP_Player` angelegt wurde. Der `CHESSGP_Player` wird durch `PVM_Player` erstellt, der durch das pvm-System auf jedem Rechner erstellt werden kann, auf dem ein pvm-Daemon gestartet wurde. Der `PVM_Player` ist wiederum nur eine Folge des Aufrufes durch den `PVM_Pool`, der seine Anweisungen durch `CHESSGP_Pool` erhält. Abbildung 23 veranschaulicht den Zusammenhang nochmals, der auch in Abschnitt 6.2.2.2 behandelt wird. SFOR kommt zur Verstärkung des `CHESSGP_Pools` ins Spiel: RFOR kann anstelle der Ausführung eines GP-Individuums auch eine „einfache“ Tiefensuche durchführen, die mit Hilfe von SFOR realisiert wird. Dabei gibt ein Typ-Bezeichnung im `CHESSGP_Individuum` an, ob es sich um ein GP-Individuum handelt (0), ob die Zugwahl zufällig erfolgen soll (−1),

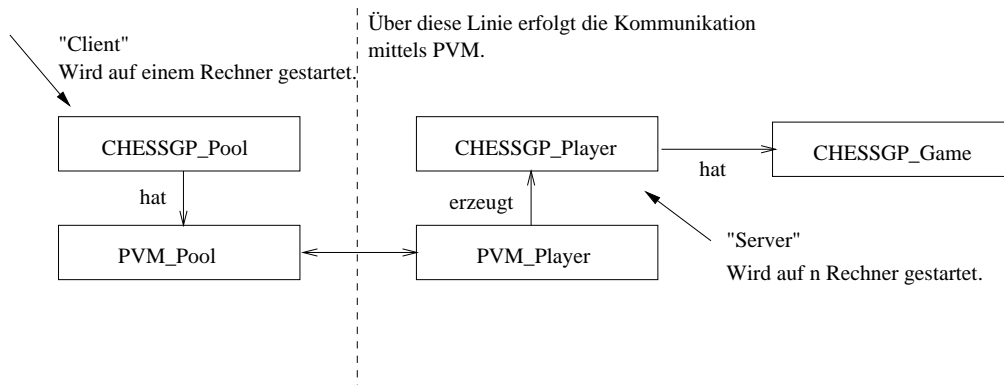


Abbildung 23: PVM-Kommunikationsübersicht

oder aber ob eine Tiefensuche mit SFOR durchgeführt werden soll, wobei der Typ (> 0) dann die Suchtiefe angibt. Somit ist es auch möglich, eine objektive Verbesserung bzw. Bewertung unserer evolvierten Individuen durchzuführen, indem in regelmäßigen Abständen gegen immer gleich starke MiniMax-Individuen gespielt wird.

6.2.2.2 Weitere Implementierungen

Neben den beschriebenen Klassen wurden auch andere Programmierarbeiten vollzogen. Diese werden im Folgenden beschrieben.

functionABC.hh

Für die Stufen a bis c wurden verschiedene Operationen geschrieben, die von den jeweiligen Individuen aufgerufen werden können. Diese befinden sich in den Dateien `functionA.hh`, `functionB.hh` und `functionC.hh` und wurden in Abschnitt 6.1.5.5 beschrieben. Die Operationen der Stufe b umfassen eine Methode, die es ermöglicht, auf Individuen der Stufe a zuzugreifen (`callADFa()`), ebenso ist es einem Individuum der Stufe c möglich, Individuen beider Vorstufen aufzurufen.

Strukturfiles der Stufen a, b und c

Um die Lernphasen der Stufen a, b und c von „außen“ (also ohne sie neu kompilieren zu müssen) zu steuern, wurden einige neue Parameter in die Strukturfiles eingebaut, von denen hier einige exemplarisch aufgeführt werden sollen. Das Strukturfile der Stufe c umfasst dabei auch einige Parameter, die die Individuen der Stufe a und b betreffen, da diese bei der Evolution der letzten Stufe mit inbegriffen sind.

1. fitnessCases: Die Anzahl der zu betrachtenden Fitnesscases.

2. chessboardNumber: Die Anzahl der zu betrachtenden Schachbretter.⁹⁹
3. fitnessType: Dieser Parameter spezifiziert den zu erlernenden Bewertungstyp als String (z. B. „Material“ in Stufe a).
4. probFitnessCases: Die Wahrscheinlichkeit, mit der ein Fitnesscase in der nächsten Generation durch einen neuen ersetzt wird.
5. evaluationType: Dieser Parameter wird in der Fitnessbewertung der Stufe b verwendet.
6. RestartGen: Anzahl der Generationen nach denen geprüft wird, ob ein Restart notwendig ist.
7. RestartProb: Steigung, ab der ein Restart durchgeführt wird.
8. Debug: Mit diesem Parameter lassen sich programmweite Ausgaben, die das Debuggen vereinfachen, an- bzw. abschalten.
9. Mutation/Xover: Es gibt verschiedene Parameter, über die sich XOver- und Mutationsraten der a-, b- und c-Individuen beeinflussen lassen.
10. numberOfParallelGames: Anzahl gleichzeitig parallel auszuführender Spiele.
11. replacementThreshold: Anzahl zu ersetzender Individuen die ersetzbar sein müssen, bevor eine Rekombination ausgeführt wird.

Hauptprogramme der Stufen a und b

Für Stufe a und b wurden zwei Hauptprogramme, die die Evolutionsschleifen beinhalten, implementiert. Hier wird das Strukturfile ausgewertet und die zu verwendenden Operationen bestimmt. Desweiteren wird das FitnessArray für die Evolution angelegt. Es werden verschiedene Ausgaben in eine Datei geschrieben, die über Angaben im Strukturfile parametrisiert werden können. Eine genauere Betrachtung der Hauptprogramme wird durch das in Kapitel 6.2.2.6 dargestellte Ablaufschema ermöglicht.

Hauptprogramme der Stufe c

Stufe c besteht im Grunde aus zwei Hauptprogrammen: Pool und Player, die nicht mit den ähnlich lautenden Klassen zu verwechseln sind! Der Pool stellt das eigentliche Hauptprogramm dar, das vom Benutzer einmal gestartet werden muss. Das Programm legt dann einen `CHESSGP_Pool` an, dessen Methode `poolLoop()` aufgerufen wird. Dieser Aufruf impliziert den (fortlaufenden und parallel ablaufenden) Start von Playern auf verschiedenen Rechnern, die wiederum einen `CHESSGP_Player` erzeugen und deren Methode `play()` aufrufen.

⁹⁹ Es befinden sich etwa 65 00 Schachbretter in der Datenbank. Um nicht auf allen möglichen Schachbrettern trainieren zu müssen, kann die Anzahl mit diesem Parameter eingeschränkt werden.

6.2.2.3 SYSGP-Anpassungen

SYSGP selbst wurde um den „Linearen Baum“ (LinTree) erweitert, der für Individuen der Stufe b eingesetzt werden sollte. Aufgrund starker Speicherprobleme (die Individuen wurden extrem groß), wurde dann aber auf die weitere Verwendung dieser Neuerung verzichtet. Desweiteren mussten einige Stellen in der SYSGP Klassen-Bibliothek angepasst werden, da trotz durchgehender Template-Verwendung (für die Register) an manchen Stellen explizit mit „double“ gearbeitet wurde (wir aber `CHESSGP_RegisterElement` anstelle dessen benutzen). Da die Template-Verwendung weitere Ungereimtheiten zu Tage beförderte und ohnehin nicht benutzt werden musste, wurde im Laufe der Projektgruppe die alte SYSGP-Bibliothek durch eine templatefreie Version („KenoGP“) ersetzt.

6.2.2.4 Crafty

Das Programm Crafty wird benötigt, um die meist sehr frühzeitig aufgegebenen Schachpartien für die Datenbank zu Ende, d. h. bis zum Schachmatt, zu spielen. Außerdem wurde die Bewertung „Beste Züge“ mittels Crafty ermittelt. Es wurden mehrere Klassen geschrieben, um den Austausch des zu verwendenden Schachbretts und das Auslesen der von Crafty ermittelten Züge bzw. Bewertungen zu ermöglichen. Diese werden nun kurz vorgestellt:

- **FDB_Chessinterface:**

Diese Klasse startet die Crafty-Instanzen und regelt die Kommunikation zwischen der Klasse `FDB_Spiel` und Crafty.

- **FDB_DBEndspiele:**

`DBEndspiele` ist die Schnittstelle zur Datenbank. Die Klasse schreibt die neuen (Endspiel-)Schachbretter in die Datenbank.

- **FDB_EndBrett:**

Diese Klasse speichert intern den aktuellen Stand des Schachspiels und wandelt diesen auf Anfrage in ein für die Datenbank taugliches Format um.

- **FDB_Move:**

`FDB_Move` speichert die einzelnen Züge, die zwischen den `FDB_Chessinterface`-Klassen ausgetauscht werden, ab.

- **FDB_Spiel:**

`FDB_Spiel` holt sich eine „Endstellung“ aus der Datenbank, startet die Schachprogramme mittels `FDB_Chessinterface` und schreibt die erhaltenen Bretter wieder in die Datenbank zurück.

6.2.2.5 *xboard*

Wir haben eine Anbindung an die Schachoberfläche *xboard* geschrieben, um gegen unsere eigenen Individuen zu spielen.¹⁰⁰ Im Hintergrund läuft dabei wiederum eine Instanz der Klasse `CHESSGP_Game`, die das Spiel verwaltet.

- **CHESSGP_Interface:**

Diese Klasse schafft die Verbindung von unserem `CHESSGP_Game` zum *xboard*. Die einzelnen Züge werden über Streams ausgetauscht.

- **mainCHESSGP_XBoard:**

Diese Klasse beinhaltet das Hauptprogramm, dass dafür sorgt, dass ein Individuum eingelesen wird, die Verbindung zu *xboard* aufgebaut wird und ein Spiel beginnen kann.

- **programmify und programmify_pool:**

Diese beiden Hilfsklassen ermöglichen auf einfache Weise, die Ausgaben der Evolution (mehrere Dateien) in eine handliche Datei zu wandeln, die vom Hauptprogramm erwartet wird. Dabei werden die Ausgaben lediglich hintereinander geschrieben, was aber zu einer Vereinfachung des Aufrufs führt.

6.2.2.6 *Ablaufschema*

Im Folgenden wird ein Überblick über den Verlauf der Lernphasen gegeben. Das grobe Schema beschreibt dabei den kompletten Ablauf von Stufe a bis Stufe c, wobei letztere zusätzlich parallel ausgeführt wird. Das feine Schema stellt das Trainieren eines einzelnen Stufe-a-Individuums vor. Dieser Ablauf ist dann auf die anderen Lernphasen in ähnlicher Weise übertragbar.

Grobes Schema

Abbildung 24 zeigt den groben Ablauf der Lernphase. Es werden 8 verschiedene Individuen der Stufe a (Individuen, die den bereits beschriebenen statischen Bewertungen zuzuordnen sind) entwickelt. Dabei wird auf die zuvor in die Datenbank geschriebenen Fitnesswerte zurückgegriffen. Zur Entwicklung der Stufe b werden alle 8 a-Individuen eingelesen und so dem jeweiligen b-Individuum bereitgestellt. Die unterschiedlichen b-Individuen greifen bei ihrer Evolution auf die ebenfalls in die Datenbank eingetragenen Züge zurück. Durch eine spezielle Operation (`callADFa()`) können die Individuen der Stufe b auf Individuen der Stufe a zurückgreifen (und deren Bewertungen verwenden).

Für die Evolution des (schachspielenden) c-Individuums werden wiederum die „Agenten“ der ersten beiden Stufen eingelesen. Trainiert wird in dieser Stufe allerdings nicht anhand zuvor berechneter (und in die Datenbank eingetragener) Werte, sondern durch das Spielen gegen sich selbst. Zusätzlich stehen Individuen zur Verfügung, die keine GP-Individuen darstellen, sondern entweder zufällige Schachzüge ziehen oder einem MiniMax-Verfahren mit Suchtiefen 1–4 folgen.

¹⁰⁰ Diese Anbindung war während der Projektgruppe unter dem Namen „Stufe d“ bekannt.

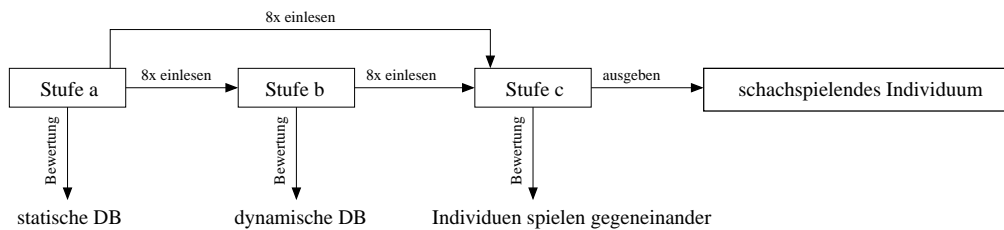


Abbildung 24: Grobes Ablaufschema

Initialisierung	
1	Operationen
2	Strukturfile
3	Ausgabedatei
4	Environment
5	XOver + Mutationsrate bestimmen
6	Pool
7	Interpreter
8	FitnessArray
9	Fitness-Klasse
Evolutionsschleife	
1	FitnessArray update
2	Fitness aller Individuen bestimmen
3	Pool nach Fitness sortieren
CrossOver-Schleife	
1	Tournament-Selektion
2	CrossOver
3	Mutation
4	Fitness-Bestimmung
Ausgabe des besten Individuums	

Abbildung 25: Feines Ablaufschema

Feines Schema

Im Folgenden wird beispielhaft die Entwicklung eines Individuums der Stufe a erläutert. Der Ablauf ist in der Stufe b ähnlich realisiert, es werden zusätzlich noch die (fertig evolvierten) Individuen der Stufe a eingelesen. Entsprechend können Individuen der Stufe c sowohl auf a- als auch auf b-Individuen zugreifen.

Der Abbildung 25 ist der Programmablauf zur Evolution eines a-Agenten zu entnehmen. Er ist für alle Agenten dieser Stufe gleich, lediglich die Werte der jeweiligen über das Strukturfile angegebenen Bewertung (bzw. deren nachzubildende Funktion) unterscheiden sich. Es kann auch notwendig sein, die Parameter der Programmgröße oder andere Einstellungen zu verändern. Die erste Aufgabe für das Starten der Evolution stellt also das Anpassen des Strukturfiles auf die jeweiligen Gegebenheiten dar.

Der Ablauf orientiert sich an dem Hauptprogramm, das die Evolutionsschleife beinhaltet.

1. Alle Operationen werden angelegt und in die Operationsmenge eingefügt.
2. Das Strukturfile wird initialisiert, dazu gehört unter anderem das Auslesen der verschiedenen Parameter (Generationsanzahl, Populationsgröße,

Größe der Individuen, Wahrscheinlichkeiten, Anzahl der Fitnesscases) und die Bestimmung erlaubter Operationen. Die Menge der verfügbaren Operationen kann also durch Angaben im Strukturfile eingeschränkt werden.

3. Anlegen der Ausgabedatei (Parameter „GnuFile“).
4. CrossOver- und Mutationsraten bestimmen.
5. Initialisierung des Environment, also Anlegen und „Säubern“ der Register.
6. Initialisierung des Pools. Es werden neue Individuen anhand der verschiedenen Parameter aus dem Strukturfile angelegt.
7. Initialisierung des (linearen) Interpreters.
8. Initialisierung des `FitnessArrays`. Dabei wird das `FitnessArray` mit Nummern von Schachbrettern (als „Zeiger“ in die Datenbank) gefüllt. Die Anzahl wird ebenfalls im Strukturfile spezifiziert.
9. Initialisierung der Fitness-Klasse. Dabei wird der Fitness-Klasse die Anzahl von Fitnesscases, Schachbrettern, der Bewertungstyp, das `FitnessArray` und der Interpreter bekannt gemacht. Letzter wird benötigt, um die Individuen zur Fitnessbestimmung auszuführen.

Nachdem diese Initialisierungsphase abgeschlossen ist, tritt das Hauptprogramm in die eigentliche Evolutionsschleife ein.

1. `FitnessArray` anpassen (mit der im Strukturfile angegebene Wahrscheinlichkeit werden vorhandene Fitnesscases durch neue ersetzt).
2. Bestimmung der Fitness aller Individuen.
3. Sortieren des Pools nach der Fitness (dieser Schritt ist für eine korrekte Abarbeitung der folgenden Turnierselektion nötig).

Die Anzahl der Wiederholungen der folgenden Schritte wird durch die im Strukturfile angegebene Rekombinationsrate festgelegt:

1. Durchführung der Turnierselektion: Es werden zwei „Turniere“ durchgeführt. Dabei werden jeweils das beste und das schlechteste Individuum aus dem Pool herausgesucht. Dann wird das schlechteste durch das beste Individuum ersetzt.
2. Die neu eingefügten Individuen (die Kopien der besten) werden durch ein (Ein-Punkt-)Crossover miteinander gekreuzt.
3. Danach werden diese Individuen mit einer gewissen Wahrscheinlichkeit mutiert. Die Mutation kann den Austausch einer Operation gegen eine andere oder die Änderung von Konstanten und Registern bedeuten.
4. Zuletzt werden die Fitnesswerte der neu erstellten Individuen bestimmt.

Nachdem die Evolutionsschleife durchlaufen ist, wird das beste Individuum in die Datei geschrieben, und das Programm ist beendet.

6.2.3 Das Datenbankschema

Die Datenbank dient dazu, einen schnellen Zugriff auf die während der Evolution benötigten Fitnesscases zu ermöglichen. Dabei ist zu berücksichtigen, dass das gewählte Stufenkonzept drei Evolutionsphasen beinhaltet: In der ersten Phase werden Programme zur Bewertung einer Brettstellung generiert, die zweite Phase evolviert Agenten, welche eine Liste von Zügen zu einer gegebenen Brettstellung zurückgeben und schließlich soll in der dritten Phase ein Programm erzeugt werden, welches in der Lage ist, Schach zu spielen. Die Evolution greift jedoch nur in den beiden ersten Phasen auf Fitnesscases der Datenbank zurück.

Für den Entwurf eines Datenbankschemas ist festzustellen, welche grundlegenden Einheiten bei der Modellierung der Fitnesscases für diese zwei Phasen auftreten, in welchen Beziehungen die Einheiten zueinander stehen und welche Operationen auf der Datenbank durchgeführt werden sollen.

Als *grundlegende Einheiten*, mit denen die Fitnesscases modelliert werden können, treten zunächst Brettstellungen auf. Diese werden unter verschiedenen Gesichtspunkten bewertet und stellen daneben aber auch die Grundlage für die zu bewertenden Zuglisten dar. Die Zuglisten werden nicht direkt betrachtet. Vielmehr wird eine Zugliste als Folge von Zügen aufgefasst. Die Bewertung der Liste ergibt sich aus der Summe der Bewertungen der in ihr enthaltenen Züge. Als weitere Einheit des Datenbankschemas werden daher die einzelnen Züge betrachtet, aus denen sich die Zuglisten zusammensetzen. Schließlich sind die Gesichtspunkte zu erfassen. Als *Gesichtspunkt* werden die Kriterien bezeichnet, unter denen die Bewertung der Brettstellungen bzw. der Spielzüge erfolgt. Ein möglicher Gesichtspunkt für eine Stellungsbewertung ist zum Beispiel die Materialbewertung. Hierbei wird die Brettstellung mit der Differenz zwischen weißem und schwarzem Spielmaterial bewertet. Als weitere Gesichtspunkte zur Stellungsbewertung werden beispielsweise die Kriterien herangezogen, inwiefern eine Stellung aggressiv ist bzw. inwiefern eine Stellung aus Sicht des weißen Spielers eine Verteidigungsstellung verkörpert. Vergleichbare Gesichtspunkte werden zur Bewertung der Spielzüge herangezogen. Beispiele sind die Gesichtspunkte „Angriffszug“ und „Verteidigungszug“, bei welchen bewertet wird, ob ein Zug einen Angriff bzw. im zweiten Fall eine verteidigende Maßnahme darstellt. Eine vollständige Auflistung der angewandten Gesichtspunkte sowohl der Stellungs- als auch der Zugsbewertung sowie eine Beschreibung der zur Berechnung der zugewiesenen Werte genutzten Funktionen kann in Abschnitt 6.2.4 nachgelesen werden.

Die zu erfassenden *Beziehungen* zwischen diesen Einheiten ergeben sich nunmehr auf natürliche Weise. Zum einen stehen Brettstellungen und Gesichtspunkte der Stellungsbewertung in Beziehung zueinander. Zum anderen besteht eine Beziehung zwischen einem Zug, der Brettstellung, in der er möglich ist, sowie dem Gesichtspunkt, unter welchem der Zug bewertet wird. Eine weitere Beziehung besteht zwischen Brettstellungen und Gesichtspunkten der Zugsbewertung, welche zur Normierung der Fitnesswerte der GP-Programme in der zweiten Evolutionsphase benötigt wird (s. u.).

Als *Operationen*, welche auf der Datenbank laufen, sind das Füllen der Da-

tenbank mit Bewertungen sowie das Auslesen von Bewertungen während der Evolution von Individuen zu nennen. Das Hauptaugenmerk muss beim Entwurf des Datenbankschemas auf das Auslesen der Bewertungen gerichtet sein. Um die Evolution über eine größere Anzahl von Generationen mit einer angemessenen Poolgröße in möglichst kurzer Zeit durchführen zu können, ist es essenziell, dass die zur Bestimmung der Fitness der Individuen benötigten Referenzwerte schnell verfügbar sind. Desweiteren werden die Daten, nachdem sie einmal in die Datenbank eingefügt worden sind, nicht mehr geändert. Nach dem Füllen der Datenbank erfolgt nur noch lesender Zugriff. Aus diesen Gründen wurden die oben als grundlegende Einheit genannten Spielzüge ins Datenbankschema nicht als Einheit, sondern nur als Attribut der Stellungsbewertungs-Beziehung aufgenommen.

Das in Abbildung 26 gezeigte ER-Diagramm stellt das aus den vorhergehenden Überlegungen resultierende Datenbank-Schema dar. Eine generelle Beschreibung des Entity-Relationship-Modells kann u.a. in [BIS95] oder [vos94] nachgelesen werden.

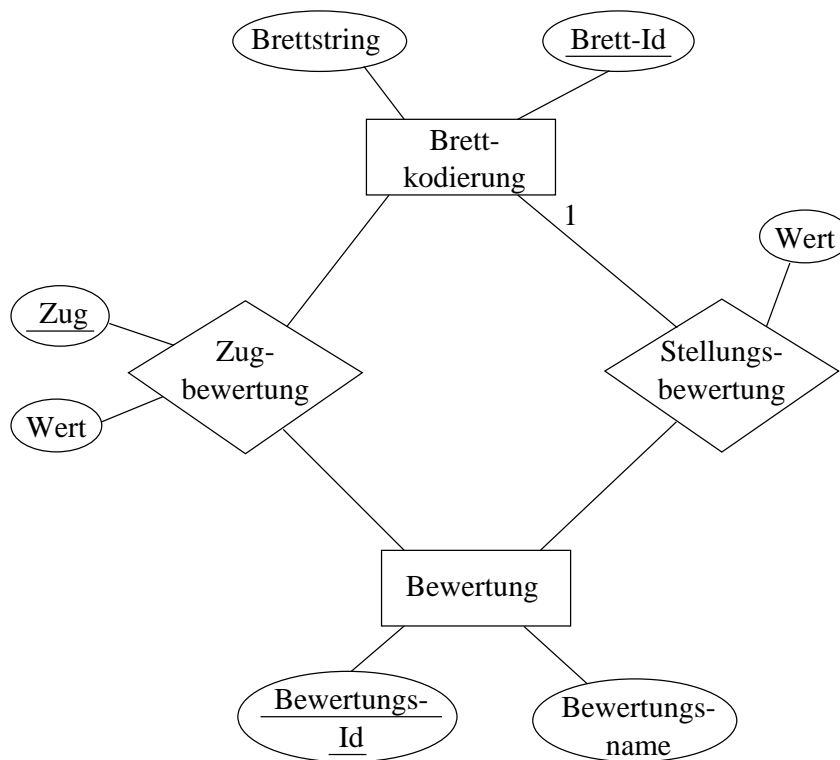


Abbildung 26: ER-Diagramm des Datenbankschemas

In den folgenden Abschnitten werden einige Aspekte näher betrachtet, welche zum Teil nur im weiteren Sinne zum Datenbankschema zu zählen sind. Anstelle der verwendeten Begriffe „Seiendenklasse“ und „Beziehungenklasse“ findet man häufig die englischen Entsprechungen „Entity-Set“ und „Relationship-Set“.

Die Seiendenklasse „Bewertung“

Die Seiendenklasse *Bewertung* besitzt die Attribute *Bewertungs-Id* (Schlüssel) und *Bewertungsname*. Hier werden die Gesichtspunkte aufgeführt, nach denen bewertet wird. Es wird nicht zwischen Zug- und Stellungsbewertungen unterschieden, beide werden in einer Seiendenklasse zusammengefasst. Die korrekte Zuordnung zu Beziehungen muss beim Füllen der Datenbank beachtet werden, sie wird durch das Schema nicht gewährleistet. Der Bewertungsname gibt Aufschluss darüber, welcher Aspekt bewertet wird.

Die Seiendenklasse „Brettkodierung“

Als Attribute enthält diese Seiendenklasse *Brett-Id* (Schlüssel) und *Brettstring*. Der *Brettstring* symbolisiert die Brettstellung (siehe unten), die *Brett-Id* identifiziert diese Brettstellung. Um sich auf eine Brettstellung zu beziehen genügt die Angabe der *Brett-Id*. Das wiederholte Nennen des relativ langen Brettstrings entfällt.

Die Beziehungenklasse „Stellungsbewertung“

Diese Klasse setzt eine Brettstellung und eine Stellungsbewertung in Beziehung zueinander. Das Attribut *Wert* beinhaltet die Bewertung der Stellung unter dem jeweiligen Bewertungsaspekt.

Die Beziehungenklasse „Zugbewertung“

Die Beziehungenklasse *Zugbewertung* bewertet die in einer Brettstellung möglichen Züge (Attribut *Zug*) unter dem jeweiligen Bewertungsaspekt. Die Bewertung wird im Attribut *Wert* abgelegt.

Kodierung der Brettstellungen

Eine Brettstellung wird in der Datenbank als Zeichenkette abgelegt. Grundlage für diese Zeichenkette bilden die englischen Stein-Abkürzungen der PGN-Notation: P wird für Bauer (pawn), N für Springer (knight), B für Läufer (bishop), Q für Dame (queen) und K für König (king) benutzt. Zusätzlich stellt das E einen Bauern dar, welcher en passant geschlagen werden darf, T und I stehen für Turm und König, welche noch rochieren dürfen. Schwarze Spielsteine werden durch einen kleinen, weiße durch einen großen Buchstaben markiert. Für Leerfelder steht der Buchstabe z. Zur Verkürzung werden aufeinanderfolgende Leerfelder durch ein z, an welches die Anzahl der Leerfelder angehängt wird, dargestellt. Der sogenannte *Brettstring* ergibt sich nun aus der Brettstellung, indem von unten nach oben jede Zeile von links nach rechts durchlaufen wird, wobei jedes Feld durch den korrespondierenden Buchstaben zu ersetzen ist. Folgendes Beispiel soll die Kodierung verdeutlichen. Die in Abbildung 27 gezeigte Brettstellung wird durch den Brettstring

`z3NzNz3Pz2Pz2Pz2Pz2PzkzPz3bzpzpz7Rz2rz5BBz5KR`

repräsentiert.

Kodierung von Zügen

Ein Spielzug wird über eine vier- bzw. fünfstellige Zahl kodiert. Die (von links aus

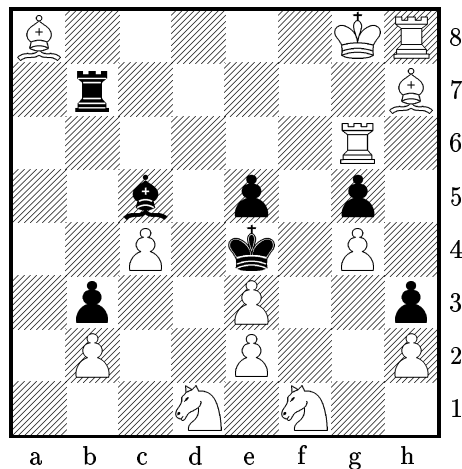


Abbildung 27: Brettstellung und Brettstring

gesehen) erste Ziffer ist optional und steht für die Zielfigur, falls der Zug zu einer Bauernumwandlung führt. Die Bedeutung der Ziffer entspricht den Konventionen der Klasse `CHESSGP_Board` (2: Springer, 3: Läufer, 5: Turm und 9: Dame). Die zweite und dritte Ziffer bezeichnet das Ausgangsfeld des Zuges, wobei zunächst die Spalte und dann die Zeile angegeben wird. „21“ steht somit für das Feld „b1“. In gleicher Weise stellt die vierte und fünfte Ziffer das Zielfeld dar.

6.2.4 Fitnessbewertung

Zur Evolution der Individuen der Stufen a und b sind Fitnessfunktionen implementiert worden, die dazu dienen, den Fitnesswert eines Individuums zu berechnen. Anhand dieser Fitnesswerte werden dann die besten Individuen ausgewählt (selektiert), um am Evolutionsprozess weiter teilzunehmen. Die folgende Beschreibung orientiert sich an den Erläuterungen aus Kapitel 6.1.4.

Zur Fitnessbewertung der Individuen wird in den Stufen a und b auf in der Datenbank abgelegte Fitnesscases zurückgegriffen. Dieses sind Brettstellungen und die zugeordneten Stellungsbewertungen sowie Züge und zugeordnete Zugbewertungen. Die Stellungen- bzw. Zugbewertung erfolgt mittels *Bewertungsfunktionen*, welche in den folgenden Abschnitten dargestellt werden. Die berechneten Werte werden zu der jeweiligen Stellung bzw. dem jeweiligen Zug in die Datenbank eingetragen.

Die Bewertungsfunktionen werden durch verschiedene Methoden realisiert. Dabei wurden für Stufe a und b getrennte Klassen geschrieben, die die Verbindung zur Datenbank aufbauen und das Füllen der Datenbank ermöglichen. Die eigentlichen Bewertungen erfolgen jeweils in einer abgetrennten Methode, die sich nur um eine bestimmte Bewertungsfunktion kümmert. Als einzige Ausnahme stellt sich eine Methode dar, die die verwandten Bewertungen für Komplexität, Aggressivität und Defensivität ermitteln kann (abhängig von einem Parameter).

Die Fitnessbewertung eines Individuums der Stufe a ergibt sich dann durch den Vergleich der Stellungsbewertung des Individuums mit dem in der Datenbank stehenden Wert. Die Fitness eines Individuums entspricht der quadrierten Differenz zwischen der Stellungsbewertung des Individuums und dem in der Datenbank stehenden Referenzwert. Bewertet ein Individuum beispielsweise eine Stellung mit dem Wert 120, welcher laut Datenbank ein Wert von 110 zugeordnet werden sollte, so beträgt die Fitness des Individuums $10^2 = 100$.

Die Fitness eines Individuums der Stufe b wird durch die Bewertung der vom Individuum ermittelten Zugliste bestimmt. Die Bewertung der Zuglisten wird in Kapitel 6.2.4.3 beschrieben.

Im Folgenden werden die Bewertungsfunktionen detailliert beschrieben, die Implementierung ergibt sich aus diesen Erläuterungen.

6.2.4.1 Die Bewertungsfunktionen der Stufe a

Die Bewertungsfunktionen der Stufe a ermöglichen die Bewertung einzelner Brettpositionen. Es wird dabei ausschließlich die vorliegende Brettstellung berücksichtigt, d. h. nachfolgend mögliche Spielzüge werden nicht betrachtet. Die Funktionen werden genutzt, um die während der Evolution der Individuen der Stufe a benötigten Fitnesscases zu erhalten. Daneben werden sie zur Bewertung der Blätter der Spielbäume herangezogen, welche von den Bewertungsfunktionen der Stufe b aufgezogen werden, und können auch von Individuen der Stufe c aufgerufen werden.

Die im Folgenden vorgestellten Bewertungsfunktionen der Stufe a lassen sich in zwei Gruppen unterteilen. Zum einen existieren Bewertungen, welche auf die reine Brettstellung bezogen und farbunabhängig sind. Zum anderen gibt es Stellungsbewertungen, welche aus Sicht eines Spielers vorgenommen werden und somit farbabhängig sind. Farbabhängige Bewertungen werden immer aus Sicht des weißen Spielers vorgenommen.

Abbildung 28 veranschaulicht diese Unterteilung und listet die implementierten Bewertungsfunktionen der Stufe a auf:

Im folgenden Text werden die gezeigten Funktionen näher erläutert:

1. Komplexität

Die Komplexität einer Brettstellung wird ermittelt, indem für jeden Stein (sowohl schwarz als auch weiß) die Anzahl erreichbarer Felder bestimmt wird. Als Komplexitätsmaß wird dann die Summe dieser Zahlen herangezogen. Züge, bei denen geschlagen wird, werden doppelt gewichtet.

$$\sum_{S \in \text{Steine}} (\text{Anzahl der von S erreichbaren Felder} + \text{Anzahl der von S ausführbaren Schlagzüge})$$

2. Aggressivität

Eine Stellung wird als aggressiv eingestuft, wenn die Anzahl an Schlagzügen groß ist. Die Summe der möglichen Schlagzüge wird wieder über weiße und schwarze Figuren gebildet. Es werden jedoch nur solche Schlagzüge

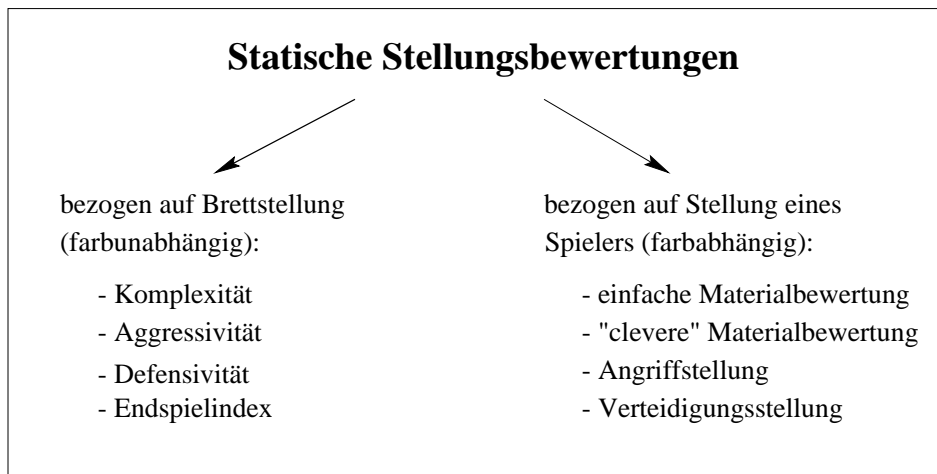


Abbildung 28: Bewertungsfunktionen der Stufe a

betrachtet, bei denen ein (bezogen auf den schlagenden Stein) höher- oder gleichwertiger oder ungedeckter Stein geschlagen wird (sog. *Gewinnschlagzüge*).

$$\sum_{S \in \text{Steine}} \text{Anzahl der von } S \text{ ausführbaren Gewinnschlagzüge}$$

Jeder Gewinnschlagzug fließt in gleicher Höhe in die Summe ein. Auf eine differenzierte Bewertung von Gewinnschlagzügen, beispielsweise die Höherbewertung eines Schlagzuges, bei dem die Differenz zwischen dem Steinwert des schlagenden und des geschlagenen Steins besonders groß ist, wird verzichtet. Zur Bewertung der Aggressivität ist eine solche Unterscheidung unnötig.

3. Defensivität

Eine Brettstellung wird als umso defensiver betrachtet, je mehr Steine gedeckt sind. Pro Stein wird die Anzahl der diesen Stein deckenden Steine bestimmt und darüber summiert:

$$\sum_{S \in \text{Steine}} \text{Anzahl der } S \text{ deckenden Steine}$$

4. Endspielindex

Der Endspielindex soll angeben, in welchem Maße die Brettstellung Endspielcharakter aufweist. Dabei wird die Anzahl der sich noch auf dem Brett befindenden Steine ausgewertet. Ein „hundertprozentiges“ Endspiel soll vorliegen, wenn noch 13 Steine auf dem Brett stehen. Sind weniger Steine vorhanden, steigt der Endspielindex linear. Genauso fällt er bei einer

höheren Anzahl Spielsteinen. Der Endspielindex beträgt 0, wenn alle 32 Steine auf dem Brett stehen. Die Zahl 13 wurde willkürlich gewählt und dient lediglich als Bezugspunkt für ein hundertprozentiges Endspiel. Folgende Funktion berechnet den Endspielindex:

$$\begin{aligned} \text{EIndex}(\text{Anzahl Steine}) &= \frac{32 - \text{Anzahl Steine}}{32 - 13} \cdot 100 \\ &= \frac{3200}{19} - \frac{100}{19} \cdot \text{Anzahl Steine} \end{aligned}$$

5. Einfache Materialbewertung

Die einfache Materialbewertung summiert die Materialwerte der weißen Steine auf und subtrahiert davon die Materialwerte der schwarzen Steine. Die Steine werden wie folgt bewertet:

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

Die einfache Materialbewertung kann mit folgender Formel berechnet werden:

$$\sum_{\text{W weißer Stein}} \text{Materialwert W} - \sum_{\text{S schwarzer Stein}} \text{Materialwert S}$$

6. „Clever“ Materialbewertung

Die clevere Materialbewertung ist eine Erweiterung der einfachen Materialbewertung. Es geht zusätzlich die Berücksichtigung der Position des betrachteten Steines auf dem Spielbrett ein. Außerdem erhalten Bauern mit fortschreitendem Spielverlauf eine höhere Gewichtung.

Die Berücksichtigung der Brettposition eines Steines erfolgt mittels einer Positionsmatrix. Jedem Steintyp wird eine solche Positionsmatrix zugeordnet. Falls sinnvoll wird zwischen schwarzen und weißen Steinen unterschieden. Zeilen und Spalten der Matrix entsprechen den Positionen eines Schachbrettes. Für die weißen Bauern könnte eine solche Matrix wie folgt aussehen:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 8 & 12 & 16 & 16 & 12 & 4 & 4 \\ 4 & 8 & 12 & 16 & 16 & 12 & 4 & 4 \\ 2 & 2 & 12 & 16 & 16 & 12 & 2 & 2 \\ 0 & 0 & 0 & 15 & 25 & 0 & 0 & 0 \\ -2 & -2 & -2 & 6 & 6 & -2 & -2 & -2 \\ -2 & -2 & -2 & -5 & -10 & -2 & -2 & -2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Jeder Bauer, der sich noch in der Ausgangsstellung befindet, erhält einen Abzug von -2 , d- und e-Bauer werden noch schwerer bestraft. Ist der d-Bauer aber in die 7. Zeile vorgerückt, bekommt er einen Positionszuschlag von 16 Punkten.

Die Positionsmatrizen wurden in Anlehnung an GnuChess aufgestellt. GnuChess verwendet eine dem hier vorgestellten ähnlichen Mechanismus zur Stellungsbewertung.

Die höhere Gewichtung eines Bauern während des Spielverlaufs wird realisiert, indem jeder Bauer einen Zuschlag der Höhe $0,5 \cdot \text{Endspielindex}$ erhält.

Es ergibt sich folgende Formel für die „clevere“ Materialbewertung:

$$\begin{aligned} & \text{einfache Materialbewertung} \\ & + \sum_{W \text{ weißer Stein}} \text{Positionswert } W - \sum_{S \text{ schwarzer Stein}} \text{Positionswert } S \\ & + 0,5 \cdot \text{Endspielindex} \cdot \text{Anzahl weißer Bauern} \\ & - 0,5 \cdot \text{Endspielindex} \cdot \text{Anzahl schwarzer Bauern} \end{aligned}$$

7. Angriffsstellung

Die Bewertung unter dem Gesichtspunkt „Angriffsstellung“ spiegelt wider, inwieweit die weißen Spielsteine eine angreifende Position einnehmen. Dabei wird jeder angegriffene schwarze Stein in Höhe seines Materialwertes in die Bewertung einbezogen. Zusätzlich werden gefesselte Steine mit einem Bonus von 300 Punkten bewertet.

Desweiteren wurde dem Angriff auf den König besonderes Augenmerk geschenkt. Steht der schwarze König im Schach, so wird dies als Angriff auf einen schwarzen – besonderen – Stein gewertet und wie oben geschildert berücksichtigt. Der König wird mit einem Materialwert von 3000 Punkten belegt. Zusätzlich werden die den König umgebenden Felder als für den weißen Angriff bedeutsam angesehen. Gelingt es Weiß, ein dem gegnerischen König benachbartes Feld in den Einflussbereich eines seiner Steine zu bringen (dieses also anzugreifen), so geht dies in die Summe mit einem Wert von 75 Punkten ein.

Die sich ergebende Summe wird durch den Materialwert aller schwarzer Steine (exklusive König) geteilt, um die Bewertung in Relation zur materiellen Stärke von Schwarz zu setzen. Ansonsten könnte eine Brettstellung, bei der neben dem König nur noch ein schwarzer Bauer auf dem Brett steht, welcher attackiert wird, bezüglich des Angriffs als schlechter gewertet werden als eine andere, bei der Schwarz noch alle Steine besitzt, von denen einer gefesselt ist.

Da alle Bewertungen in ganzen Zahlen ausgedrückt werden sollen, wird der sich ergebende Wert mit 10000 multipliziert und gerundet.

Zusammenfassend erhält man die Formel (*Steine* bezieht sich hier nur auf schwarze Steine, da die Bewertung immer aus Sicht von Weiß erfolgt):

$$\begin{aligned}
& (\sum_{S \in \text{angegriffene Steine}} \text{Materialwert } S \\
& + 300 \cdot \text{Anzahl gefesselter Steine} \\
& + 75 \cdot \text{Anzahl bedrohter Königsfelder}) \\
& \cdot \frac{10000}{\sum_{S \in \text{Steine}} \text{Materialwert } S}
\end{aligned}$$

Es ist für die Einschätzung einer Stellung als Angriffsstellung ohne Bedeutung, wie stark Schwarz sich verteidigt. Ebenso bleibt unberücksichtigt, wie gut die weiße Position verteidigt ist.

8. Verteidigungsstellung

Die Verteidigungsstellung bewertet eine Brettstellung unter dem Aspekt, inwieweit die weißen Steine eine verteidigende Stellung einnehmen. Ähnlich der Bewertungsfunktion Angriffsstellung erfolgt die Bewertung im Kern über eine Summenbildung.

Zunächst werden alle weißen Steine betrachtet. Ist ein Stein gedeckt, so wird sein Materialwert zur Summe addiert, andernfalls wird dieser von der Summe abgezogen.

Desweiteren wird der Verteidigung des Königs ein hoher Stellenwert eingeräumt. Hierbei werden die direkt um den König gelegenen Felder betrachtet, sofern sie noch zum Spielbrett gehören. Pro Feld, welches von einem weißen Stein besetzt ist, wird die Summe um 100 erhöht. Analog wird pro unbesetztem Feld die Summe um 100 verringert. Falls ein schwarzer Stein auf ein solches Feld vordringen konnte, so werden pro Feld 200 Punkte von der Summe subtrahiert.

Wie bei der Bewertungsfunktion Angriffsstellung wird die resultierende Summe durch den Materialwert aller weißen Steine dividiert, um eine Beziehung zur materiellen Stärke von Weiß herzustellen. Ohne diese Division würde eine (fiktive) Verteidigung, welche nur einen Bauern deckt, unabhängig vom weißen Gesamtmaterial gleich eingeschätzt. Auf diese Art „gesicherte“ Brettstellungen würden gleich bewertet, egal ob der gedeckte Bauer der letzte weiße Stein ist, oder noch sämtliche weißen Steine auf dem Brett stehen.

Um einen ganzzahligen Wert zu erhalten, wird der Quotient mit 10 000 multipliziert und gerundet.

Es ergibt sich folgende Formel:

$$\begin{aligned}
& (\sum_{S \in \text{gedeckte weiße Steine}} \text{Materialwert von } S \\
& - \sum_{S \in \text{ungedekte weiße Steine}} \text{Materialwert von } S \\
& + 100 \cdot \text{Anzahl weiß besetzter Königsfelder} \\
& - 100 \cdot \text{Anzahl unbesetzter Königsfelder} \\
& - 200 \cdot \text{Anzahl schwarz besetzter Königsfelder}) \\
& \cdot \frac{10000}{\sum_{S \in \text{weiße Steine}} \text{Materialwert } S}
\end{aligned}$$

Bei der Bewertung der weißen Verteidigung bleibt unberücksichtigt, in welchem Maße Schwarz attackiert. Ebenso wenig werden weiße Angriffsbemühungen berücksichtigt.

Abschließend sollen die vorgestellten Bewertungen an einem in Abbildung 29 dargestellten *Beispiel* verdeutlicht werden.

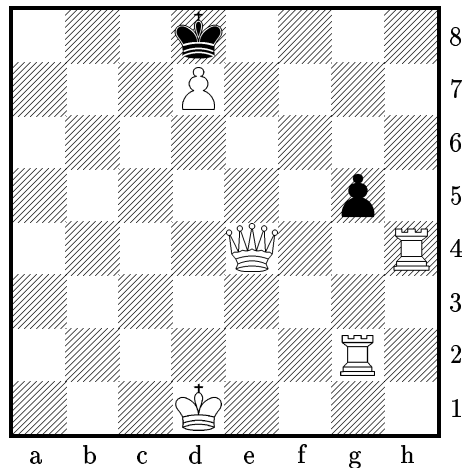


Abbildung 29: Beispielbrett

Bei Summenbildung werden zunächst die weißen Steine (Bauer, Turm g2, Turm h4, Dame, König), danach die schwarzen Steine (Bauer, König) durchlaufen.

- Komplexität: $(0 + 11 + 9 + 24 + 5 + 2 + 5) + (0 + 1 + 0 + 0 + 0 + 1 + 1) = 59$
erste Klammer: Zugmöglichkeiten, zweite Klammer: Schlagzüge
- Aggressivität: $0 + 1 + 0 + 0 + 0 + 1 + 1 = 3$
- Defensivität: $0 + 1 + 1 + 1 + 0 + 0 + 0 = 3$
- Endspielindex: $\frac{3200}{19} - \frac{100}{19} \cdot 7 \approx 131$
- Einfache Materialbewertung: $100 + 500 + 500 + 900 - 100 = 1900$
- „Clever“ Materialbewertung: $16 + 0,5 \cdot 131 \approx 82$
Positionszuschlag + Steigerung des Wertes während des Spielverlaufs
(hier nur Zuschläge des weißen Bauern)
- Angriffsstellung: $\frac{(100+0)+3 \cdot 75+0}{100} \cdot 10000 = 32500$
Der Zähler summiert angegriffene Steine, bedrohte Königsfelder und Fesselungen in dieser Reihenfolge.
- Verteidigungsstellung: $\frac{(-100+500+500+900)+0-5 \cdot 100-0}{(100+500+500+900)} \cdot 10000 = 6500$
Im Zähler werden zunächst gedeckte und ungedeckte Steine, dann von Weiß besetzte, unbesetzte und von Schwarz besetzte Königsfelder berücksichtigt.

6.2.4.2 Die Bewertungsfunktionen der Stufe b (Zugbewertung)

Die Bewertungsfunktionen der Stufe b ermöglichen die Bewertung eines Spielzugs ausgehend von einer gegebenen Brettstellung. Ein Zug wird unter folgenden Aspekten bewertet:

- Tauschzug,
- Opferzug,
- Desperadozug,
- Angriffszug,
- Verteidigungszug,
- bester Zug oder
- gefährlicher Zug.

Im Allgemeinen wird so vorgegangen, dass ausgehend von einer Brettstellung für eine geringe Tiefe der Spielbaum aufgezo- gen wird, wobei der jeweils zu bewertende Zug die erste Kante bestimmt. Die Zugbewertung erfolgt über einen Vergleich der Ausgangsbrettstellung mit den zu erreichenden Spielsituationen. Dabei wird vielfach auf die bereits vorgestellten Bewertungsfunktionen der Stufe a für Brettstellungen zurückgegriffen.

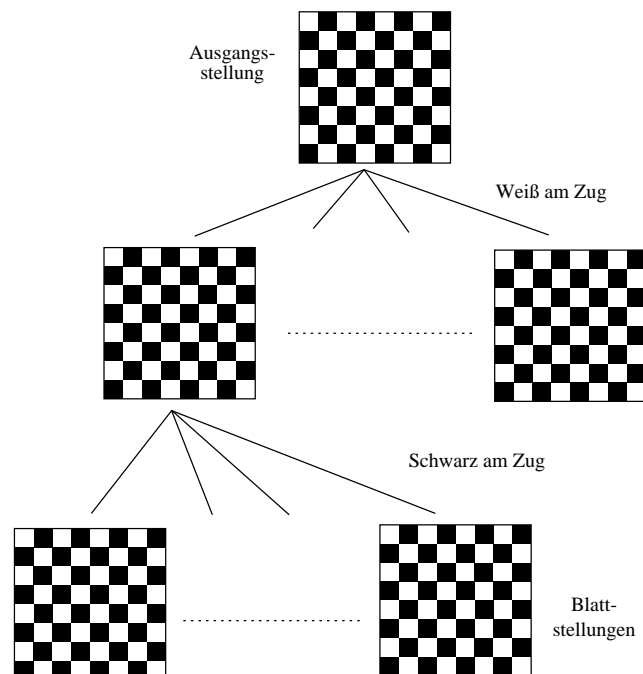


Abbildung 30: Spielbaum der Tiefe 2

Zur Veranschaulichung zeigt Abbildung 30 einen Spielbaum der Tiefe 2. Als Beispiel soll der Angriffswert des ersten weißen Spielzuges (erste linke Kante) ermittelt werden. Dazu wird zunächst die Ausgangsstellung mit der Bewertungsfunktion „Angriffstellung“ bewertet. Angenommen, der resultierende Wert sei 5 250. Es werden nun die Blätter des Teilbaums, welcher dem weißen Zug entspricht, betrachtet. Jedes Blatt wird ebenfalls mit einem Wert versehen. Es sei nun der minimale Angriffswert unter den Blattstellungen 5 425. Es muss der kleinste Wert herangezogen werden, da zuvor Schwarz am Zug war und davon ausgegangen wird, dass Schwarz den für den Gegner ungünstigsten Zug ausführt. Als Angriffswert für den weißen Zug ergibt sich die Differenz der Werte von Ausgangsstellung und Blattstellung, also 175. In gleicher Weise werden die übrigen in der Ausgangsstellung möglichen Züge bewertet.

Bei dem betrachteten Spielbaum wurde eine Tiefe von zwei Halbzügen angenommen, um das Beispiel einfach zu halten. Bei der tatsächlichen Bewertung treten auch Spielbäume mit größerer Tiefe auf. Die Bewertung der Spielzüge wird dann mit der F-Verbesserung des Negascout-Algorithmus vorgenommen. Das grundsätzliche Vorgehen entspricht jedoch auch hier dem des dargestellten Beispiels.

Wie bereits bei der Bewertung der Stufe a wird immer die Sicht von Weiß angenommen, d. h. es werden nur weiße Spielzüge bewertet. Natürlich erfolgt dann auch die Einschätzung der Brettpositionen aus weißer Sicht.

Es werden nicht grundsätzlich alle in einer Spielsituation möglichen Spielzüge bewertet. Obwohl grundsätzlich alle Spielzüge betrachtet werden, können einige Bewertungen verworfen und nicht in die Datenbank aufgenommen werden. Dies ist der Fall, wenn Spielzüge unter Kriterien wie der Gefahr, welche von diesem Spielzug ausgeht, bewertet werden. Wird ein Zug nicht als gefährlich eingestuft, so wird er nicht in die Datenbank aufgenommen. Verwirft eine Bewertungsfunktion gewisse Züge, so wird darauf in der Beschreibung der Funktion hingewiesen.

Im folgenden Text werden die Bewertungsfunktionen der Stufe b detailliert beschrieben.

1. Beste Züge

Unter der Bewertung „Beste Züge“ wird eine Einschätzung verstanden, die eine Aussage über die tatsächliche Qualität eines Zuges treffen soll. Der Zug wird nicht im Hinblick auf bestimmte Kriterien bewertet wie z. B. Angriffs- oder Verteidigungszug, sondern nur dadurch, inwieweit er die weiße Stellung grundsätzlich verbessert.

Diese Bewertung soll durch Crafty vorgenommen werden. Dem Programm wird eine Brettstellung sowie der zu untersuchende Zug übergeben, die von Crafty vorgenommene Bewertung übernommen.

2. Gefährliche Züge

Um potenziell gefährliche Züge zu ermitteln, wird es der weißen Seite erlaubt, zweimal zu ziehen, ohne dass Schwarz einen Zug ausführen darf. Erlangt Weiß durch einen solchen Doppelzug eine signifikant bessere Stellung, so wird der erste weiße Zug dieses Doppelzuges als gefährlich einge-

Stein	Weiß	Schwarz
Bauer	75	100
Springer	320	300
Läufer	280	300
Turm	450	500
Dame	950	900

Tabelle 5: Materialbewertung „Desperado“

stuft. Falls Schwarz in der wirklichen Partie auf einen solchen Zug nicht in angemessener Weise reagiert, kann Weiß einen Vorteil gewinnen.

Ob Weiß durch einen Doppelzug seine Stellung verbessern kann, wird durch einen Vergleich mit der Ausgangsstellung ermittelt. End- und Ausgangsstellung werden mittels „cleverer“ Materialbewertung eingestuft. Ein gefährlicher Zug liegt vor, wenn die Endstellung um mindestens 300 Punkte höher bewertet wird.

Es werden bei der Bewertung „Gefährliche Züge“ nur solche Züge in die Datenbank aufgenommen, die als gefährlich erkannt wurden. Der Wert, der diesen Zügen zugeordnet wird, entspricht der vorher festgestellten Differenz zwischen der Bewertung der Ausgangs- und der Endstellung.

3. Desperado-Züge

Der Begriff „Desperado-Zug“ beschreibt Züge, welche bei besonnener Betrachtung für Weiß an für sich nicht von Vorteil sind. Ein solcher Zug soll in einer Situation, in der Weiß eigentlich schon verloren hat, dazu führen, dass Schwarz in Verwirrung gerät, da mit einem solchen Zug des Gegners nicht zu rechnen war, seine eigentlich geplante Taktik verläßt, Fehler begeht und die Partie wieder öffnet. Es sind Verzweiflungszüge gemeint, die möglicherweise Weiß noch helfen, wenn alle anderen (herkömmlichen) Strategien versagen.

Es werden alle in einer Spielsituation möglichen Züge unter diesem Aspekt bewertet. Dazu wird der betrachtete Zug ausgeführt und die entstehende Brettstellung dahingehend analysiert, inwiefern sie eine Desperadostellung ist. Der Zug wird dann mit dem ermittelten Wert versehen. Die Bewertung der Desperado-Züge erfolgt also, indem ein Spielbaum der Tiefe 1 aufgespannt und die Blätter bewertet werden. Der Wert wird nicht in Relation zur Ausgangsbrettstellung gesetzt.

Die Blattbewertung erfolgt ähnlich der einfachen Materialbewertung. Es wird die Differenz zwischen weißem und schwarzem Material gebildet, wobei jedoch den Steinen andere Materialwerte zugewiesen werden (siehe Tabelle 5).

Zusätzlich zum Materialwert wird die Stellung des schwarzen Königs analysiert: Für jedes den schwarzen König umgebende Feld sowie das Königs-

feld selber wird der schwarze Materialwert um 30 verringert, sofern das Feld von Weiß angegriffen wird. Falls ein solches Feld jedoch von einem schwarzen Bauern besetzt ist, so erhält Schwarz für jedes solche Feld einen Materialbonus von 80. Der weiße Materialwert erhöht sich, falls Weiß noch Damen oder Springer besitzt, in Höhe der Entfernung bzw. Nähe dieser Figuren vom bzw. zum schwarzen König. Für jede dieser Figuren wird die Nähe der Figur zum schwarzen König auf den Materialwert addiert.

Die *Nähe* zweier Felder bewertet das Distanzmaß dieser Felder unter schachlichen Aspekten. Je größer die Distanz eingeschätzt wird, desto näher liegen die betrachteten Felder zueinander. Die größtmögliche Nähe wird mit 100 bewertet und besteht zwischen einem Feld zu sich selbst. Liegen zwei Felder in derselben Reihe, Spalte oder Diagonalen, so wird die Nähe mit Werten von 30 bis 90 angegeben, je nachdem, wie viele Felder die betrachteten Felder trennen. Sind die Felder benachbart, so beträgt die Nähe 90, für jedes dazwischenliegende Feld wird die Nähe um 10 Punkte verringert. Kann das eine Feld vom anderen über einen Springerzug erreicht werden, so beträgt die Nähe 80. In allen übrigen Fällen wird die Nähe ermittelt, indem von einer Grundnähe von 16 die Differenz der Spalten sowie die Differenz der Zeilen der Felder subtrahiert wird. Die Nähe erhöht sich um 5, falls die betrachteten Felder die gleiche Farbe tragen.

Außerdem geht die Anzahl der möglichen Spielzüge in die Bewertung ein. Der Materialwert von Weiß wird für jeden Zug um 12, der von Schwarz um 10 Punkte erhöht.

Zusammenfassend ergibt sich für die Blatt- und somit die Zugbewertung folgende Formel (Betrachtung des schwarzen Königs sowie Anzahl der Spielzüge aus Materialbewertung herausgezogen):

$$\begin{aligned}
 & \text{Materialwert Weiß} - \text{Materialwert Schwarz} \\
 & + \sum_{F \in \text{weiße Damen und Springer}} \text{Nähe}(F, \text{schwarzer König}) \\
 & + 30 \cdot \text{Anzahl angegriffener Königsfelder} \\
 & - 80 \cdot \text{Anzahl mit schwarzem Bauern besetzter Königsfelder} \\
 & + 12 \cdot \text{Anzahl möglicher Spielzüge weiß} \\
 & - 10 \cdot \text{Anzahl möglicher Spielzüge schwarz}
 \end{aligned}$$

4. Angriff / Verteidigung

Die beiden Bewertungen „Angriff“ und „Verteidigung“ führen die entsprechenden Bewertungsfunktionen der Stufe a direkt fort.

Es wird ein Spielbaum aufgespannt, bei welchem die Bewertung der Züge über die F-Verbesserung des Negascout-Suchverfahrens vorgenommen wird. Ein Blattwert ergibt sich aus der Differenz zwischen der Stellungsbewertung des Blattes und der der Ausgangsstellung. Abhängig davon, ob die Züge unter dem Gesichtspunkt der Tauglichkeit zum Angriff oder als Verteidigungszüge bewertet werden sollen, erfolgt die Stellungsbewertung durch die Bewertungsfunktion „Angriffstellung“ oder „Verteidi-

gungsstellung“. Zur Verdeutlichung sei auf Abbildung 30 zu Beginn dieses Abschnitts verwiesen.

Durch beide Bewertungsfunktionen werden alle in einer Stellung möglichen Spielzüge bewertet. Ein Zug, welcher in jedem Fall zu einer Verschlechterung der Stellung führt, erhält einen negativen Wert in der Höhe, in welcher der Stellungswert mindestens abnimmt.

5. Abtausch

Die Einstufung eines Zuges als Abtausch, welche hier beschrieben wird, entspricht nur der Grundidee des Abtausches im Schach. Allgemein wird im Schach eine Zugfolge als Tausch eingestuft, bei der beide Seiten Spielsteine schlagen, das Materialverhältnis jedoch gewahrt wird. Als Beispiel dient häufig der Damentausch: Ein Spieler schlägt mit seiner Dame die Dame des Gegners, welcher jedoch direkt im Gegenzug seinerseits diese schlagen kann.

Im Gegensatz dazu stuft die Bewertungsfunktion „Abtausch“ jeden Schlagzug als Tausch ein, nach welchem sich das Materialverhältnis nicht zugunsten des Gegners verschlechtert. Hierzu wird ausgehend von der zu betrachtenden Stellung ein Spielbaum aufgezo- gen. Die Tiefe des Spielbaums ist auf drei Halbzüge beschränkt, an der Wurzel werden nur Schlagzüge betrachtet. Falls es dem Gegner (also Schwarz) in keinem Fall gelingen kann, durch das Ausführen des Schlagzuges durch Weiß zu einem Materialgewinn zu kommen, wird der Zug als Abtausch aufgefasst.

Zur Feststellung des Materialgewinns wird die Ausgangsstellung mit Bewertungsfunktion „einfache Materialbewertung“ bewertet. Ebenso wird mit den Blättern verfahren. Ist die Differenz zwischen dem Wert einer Blattstellung und dem der Ausgangsposition nicht negativ, so hat sich aus weißer Sicht das Materialverhältnis nicht verschlechtert. Führt jede Zugfolge, an deren Anfang der momentan betrachtete Schlagzug steht, zu diesem Ergebnis, so ist der Zug ein Tauschzug. Er wird mit dem Materialgewinn, den Weiß in jedem Fall erreicht, bewertet.

Wie beschrieben werden also nur solche Züge bewertet, welche als Abtausch klassifiziert wurden.

6. Opferzüge

Die Klassifizierung eines Zuges als Opfer ist die fragwürdigste der hier aufgeführten Bewertungen, da der Begriff des Opfers in der Schachwelt unscharf und generell nur sehr schwer anwendbar ist. Die nachfolgend genannte Definition eines Opferzuges weist daher wahrscheinlich auch Schwächen in vielerlei Hinsicht auf. Dennoch erlaubt sie die Kennzeichnung eines Zuges als Opferzug.

Ein Zug wird als Opferzug aufgefasst, wenn er dem Gegner das Schlagen eines Steins ermöglicht, welches in absehbarer Spielzeit (z. B. zwei / vier Halbzüge) einen

- Gewinn der Partie oder

taktischer Vorteil	Zuschlag
Ansagen von Schach	100 Punkte
Entfernen eines Bauern (Schutz des Königs)	100 Punkte
Entfernen des Königsbauern	200 Punkte

Tabelle 6: Zuschlag für taktische Vorteile

- Materialgewinn von mindestens 400 Punkten nach der cleveren Materialbewertung

für Weiß nach sich zieht.

Neben der Berücksichtigung von Zügen, welche einen materiellen Vorteil verschaffen oder gar den Gewinn der Partie nach sich ziehen, sollen auch Opferzüge erfasst werden, welche einen taktischen Vorteil verschaffen. Um die Klassifizierung eines Zuges als Opferzug möglichst einfach zu halten, werden jedoch nur leicht feststellbare Vorteile betrachtet. Diese sind das Aufbrechen der den König schützenden Bauernreihe (nur in 2. bzw. 7. Reihe) sowie ein direkter Angriff auf den König.

Wird ein solcher Vorteil festgestellt, so wird die Bewertung des weißen Materials erhöht. Dadurch werden auch solche Züge als Opferzüge eingestuft welche unter ausschließlicher Heranziehung der „cleveren“ Materialbewertung nicht als solche gesehen würden.

Tabelle 6 zeigt die Steigerung des Materialwertes durch das Erlangen eines taktischen Vorteils.

Wurde ein Zug als Opferzug erkannt – und nur dann – so wird er bewertet. Der Wert, welcher ihm zugeordnet wird, entspricht im Allgemeinen dem zuvor ermittelten Materialgewinn inklusive Berücksichtigung der taktischen Vorteile. Falls aber die gegnerische Königsverteidigung aufgebrochen werden konnte, erfolgt eine höhere Bewertung. Das Lockern des Bauernschutzes um den gegnerischen König wird für den weiteren Spielverlauf als so bedeutsam gewertet, dass in diesem Fall der Zug mit dem doppelten Materialgewinn bewertet wird.

6.2.4.3 Bewertung der Zuglisten (Stufe b)

Wie in Kapitel 6.1.5 dargestellt wurde, liefern die Individuen in Stufe b Listen von Zügen als Ergebnis. Je nach Art des Individuums sollen diese Listen andere Züge enthalten. Soll das Individuum Züge liefern, welche zum Angreifen geeignet sind, so soll die Zugliste bevorzugt solche Züge enthalten. Darüber hinaus wird gefordert, dass diese Züge nicht nur in der Liste auftreten, sondern nach ihrer Güte sortiert sind. Der stärkste Angriffszug soll also am Kopf der Liste positioniert sein. Desweiteren soll die Länge der Zugliste beschränkt sein. Es sollen nicht alle, sondern nur die besten Züge ermittelt werden.

Da es unter vertretbarem Aufwand nicht möglich ist, im Vorfeld der Evolutionsläufe sämtliche Zuglisten zu betrachten und zu bewerten – dazu müssten zu jeder vorgegebenen Brettstellung alle Teilmengen der legalen Züge betrachtet und sämtliche Permutationen der Züge jeder dieser Teilmengen bewertet werden – enthält die Datenbank, wie im vorangegangenen Teil des Kapitels dargestellt, Bewertungen der einzelnen Züge. Dieser Abschnitt erläutert, wie mittels der in der Datenbank abgelegten Zugbewertungen die Zuglisten bewertet werden und somit die Fitness der b-Individuen berechnet wird.

Die Berechnung der Fitness erfolgt mittels einer Funktion. Diese kann entsprechend den dargestellten Kriterien *Qualität*, *Sortierung* und *Listenlänge* unterteilt werden.

Qualität der Zugliste

Die Qualität der Zugliste ist eine Funktion der Qualitäten der in der Liste enthaltenen Züge. Entsprechen die Züge den gewünschten Kriterien, liefert also das Angriffsindividuum tatsächlich gute Angriffszüge, so wird die Qualität der Zugliste als gut eingestuft. Die Bewertung der einzelnen Züge (im Folgenden *Zugwerte* genannt) kann der Datenbank entnommen werden. Dort liegen die Bewertungen normiert im Intervall $[0,1000]$ vor. Der beste Angriffszug einer Stellung besitzt den Wert 1000, der schlechteste den Wert 0.

Die Qualität der Zugliste wird nun ermittelt, indem die Zugwerte der einzelnen Züge addiert werden:

$$\text{Qualität Liste } L = \sum_{\text{Zug} \in L} \text{Zugwert}$$

Ein kleines Beispiel soll die Qualitätsermittlung veranschaulichen. In der Datenbank sind in einer Stellung fünf Züge möglich. Diese und deren Zugwerte zeigt folgende Tabelle:

Zug	Dg7	Se5	Lh6	d6	h4
Zugwert	1000	800	300	20	0

Ein schon gut trainiertes Individuum liefert vielleicht folgende Zugliste zurück: Dg7, Lh6, d6. Die Qualität dieser Zugliste beträgt $1000+300+20 = 1320$. Ein schlechteres Individuum hingegen bietet möglicherweise die Zugliste d6, h4, Lh6 an. Diese Zugliste wird mit $20 + 0 + 300 = 320$ bewertet.

Was geschieht jedoch nun, wenn ein Zug des Individuums nicht in der Datenbank auftritt? Zunächst einmal wird davon ausgegangen, dass jedes Individuum nur gültige Züge zurückliefert. Da die Datenbank im Allgemeinen alle gültigen Züge abdeckt, können häufig alle Züge der Zuglisten in der Datenbank gefunden werden.

Es gibt jedoch Bewertungskriterien, bei welchen nicht alle legalen Züge in der Datenbank vorhanden sind. So werden unter den Kriterien „Gefährliche Züge“, „Opferzüge“ und „Tauschzüge“ nur solche Züge in die Datenbank aufgenommen, die wirklich als gefährlich, als Tausch- oder als Opferzug eingestuft wurden. Ein

Zug einer Zugliste, welcher nicht in der Datenbank vorkommt, erhält eine negative Konstante, bsplw. -250 , als Zugwert. Diese Konstante kann im Strukturfile angegeben werden, -250 ist der Standardwert. Der Zugwert muss in diesem Fall negativ sein, da sein Auftreten in der Liste die Qualität der Liste senkt.

Sortierung der Zugliste

Als weitere Anforderung an die Individuen sollen die Züge der Zugliste ihrem Zugwert (und damit ihrer Qualität) entsprechend sortiert auftreten. Der Begriff der Sortierung soll sich jedoch nicht nur auf die vom Individuum gelieferte Zugliste beziehen, sondern den Stellenwert eines Zuges unter allen möglichen Zügen – also denen der Datenbank – berücksichtigen. Es erscheint ungerecht, die unsortierte Zugliste d6, h4, Dg7 (vgl. oben) gegenüber der sortierten Zugliste d6, h4 zu bestrafen. Die erste Liste beinhaltet als „unsortierten“ Zug immerhin den besten Zug der Stellung.

Daher wird ein Zug als richtig in die Liste einsortiert aufgefasst, wenn die Position des Zuges in der Liste seiner Position in der Datenbank entspricht oder er in der Zugliste weiter hinten steht als in der Datenbank. Ist dies nicht der Fall, so verdrängt ein Zug einen besseren. Die mangelnde Sortierung der Zugliste wird in Höhe der Differenz zwischen verdrängtem und verdrängendem Zugwert bestraft:

$$\text{Strafe Sortierung} = \sum_{i \leq m} (\text{Zugwert}_i - \text{Zugwert}_{\text{Pos}_i}) \text{ für alle } i < \text{Pos}_i$$

wobei m die Länge der Zugliste bzw. der Anzahl der Datenbankzüge meint. i steht für die Position eines Zuges in der Zugliste, Pos_i beschreibt die Position dieses Zuges in der Datenbank. Schließlich steht Zugwert_k für den Zugwert des in der Datenbank an Position k stehenden Zuges.

Dieses zugegebenermaßen auf den ersten Blick verwirrende Verfahren sei wieder anhand eines Beispiels verdeutlicht. Die Datenbank enthalte dieselben Züge, als Zugliste nehmen wir Dg7, Lh6, d6. Der erste Zug der Liste ist auch der erste Zug der Datenbank. Es gilt also $i = \text{Pos}_i$, die Position des ersten Zuges ist korrekt und wird nicht bestraft. Der zweite Zug jedoch steht in der Datenbank weiter hinten, nämlich erst an dritter Position ($i < \text{Pos}_i$). Er verdrängt mit seinem Zugwert von 300 einen Zug mit Zugwert 800. Dieses wird mit $800 - 300 = 500$ Punkten bestraft. Auch der dritte Zug verdrängt einen besseren, die Bestrafung erfolgt in Höhe von 280 Punkten. Die Zugliste erhält unter Berücksichtigung der Kriterien Qualität und Sortierung also eine Bewertung von $1320 - 780 = 540$.

Das Bestrafen der Sortierung kann in gleicher Weise für Züge erfolgen, welche nicht in der Datenbank auftreten.

Länge der Zugliste

Den ersten zwei Kriterien zufolge ist die beste Zugliste diejenige, welche alle Züge der Datenbank in korrekter Reihenfolge enthält. Wünschenswerterweise sollen die Individuen aber die besten Züge herausfinden und nur die in der Liste zurückliefern. Dieses Verhalten soll durch Bestrafung von Listen erfolgen, deren Längen von der optimalen Listenlänge abweichen. Die Bestrafung erfolgt bei zu

kurzen und zu langen Listen gleichermaßen, da zwar eine Selektion von Zügen erzwungen werden soll, dennoch aber eine gewisse Vielfalt erhalten werden soll.

Zur Bestimmung der *optimalen Listenlänge* wird der Durchschnitt aller in der Datenbank (zu der betrachteten Brettstellung) vorhandenen Zugwerte ermittelt. Die optimale Listenlänge entspricht nun der Anzahl an Zügen der Datenbank, deren Zugwert dem Durchschnitt entspricht oder diesen übertrifft.

Im obigen Beispiel beträgt der Durchschnitt 424. Die optimale Listenlänge ist also zwei, da der Zugwert zweier Züge besser als der Durchschnitt ist.

Um die optimale Listenlänge begrenzen zu können, kann im Strukturfile eine maximale Listenlänge angegeben werden. Die optimale Listenlänge entspricht dann dem Minimum aus dieser maximalen und der wie beschrieben ermittelten Listenlänge.

Wird die optimale Listenlänge (in folgender Formel l_{opt}) unter- oder überschritten, so wird die Bewertung der Zugliste pro Zug Abweichung um einen Wert vermindert. Die Höhe dieses Wertes richtet sich nach dem Zugwert des Zuges, der in der Datenbank an Position l_{opt} steht:

$$\text{Strafe Listenlänge} = |l_{opt} - m| \cdot \text{Zugwert}_{l_{opt}}$$

Wieder steht m für die Länge der Zugliste und Zugwert_k für den Zugwert des an k -ter Position stehenden Zuges der Datenbank.

Im Beispiel würde also die Zugliste Dg7, Lh6, d6 mit einer Bewertung von $1320 - 780 - 800 = -260$ bewertet.

Falls die Datenbank zu einer Stellung keine Züge enthält, ist die optimale Listenlänge offensichtlich 0. In diesem Fall kann der Wert, mit welchem ein Überschreiten der Listenlänge bestraft wird, im Strukturfile angegeben werden. Die Vorgabe ist eine Bestrafung um 250 Punkte.

Zusammenfassend ergibt sich für die Bewertung einer Zugliste:

$$\text{Bewertung Zugliste} = \text{Qualität} - \text{Strafe Sortierung} - \text{Strafe Länge}$$

Normierung

Wie bei den Zügen wird die Bewertung der Zuglisten normiert. Dazu wird eine obere (scharfe) sowie eine untere Schranke je Brettstellung ermittelt, zwischen denen sich die Zuglistenbewertungen bewegen müssen. Relativ zu diesen Schranken wird jede Bewertung im Intervall $[0, 1000]$ normiert. Die beste Liste erhält eine Bewertung von 0, die (theoretisch) schlechteste eine Bewertung von 1000.

6.2.5 Realisierung der Parallelisierung in der Stufe c

Der komfortable und für Benutzer wie High-Level-Programmierer fast unsichtbare Versand von Individuen wird realisiert mit der Bibliothek PVM („Parallel Virtual Machine“, siehe 4.8) sowie geringer selbsterstellter Hilfsfunktionalität. Beispielsweise können Instanzen der Klasse `CHESSGP_PoolStability` per Email Warnungen an die jeweiligen Prozess-Besitzer verschicken, falls der Anteil der erfolgreich beendeten Partien zu tief absinkt.

Einen genauen Architekturüberblick verschaffe sich der Leser anhand Abbildung 22 in Abschnitt 6.2.2.1.

6.2.5.1 CHESSGP_Player, die Server

Genaugenommen bestehen die Server aus pvmd3-Dämonen, aber im weiteren Sinne kann man die Klassen `CHESSGP_Player` und `PVM_Player` ebenfalls dem Server zurechnen. Insbesondere stellt die Klasse `CHESSGP_Player` die für das Schachspielen benötigte Funktionalität bereit.

Eine Instanz der Klasse `CHESSGP_Player` organisiert und überwacht eine Partie. Zu Anfang initialisiert es das „gemeinsame Schachbrett“ der beiden Individuen mit der im Strukturfile angegebenen Position. Während der Partie präsentiert es den teilnehmenden Individuen fortlaufend die aktuelle Schachposition, und fordert erstere auf, einen Zug abzugeben. Dieses `CHESSGP_Player`-Objekt bürgt auch für die Einhaltung der Schachregeln, prüft, ob eine Remis- oder Mattsituation vorliegt, berechnet nach Abschluß der Partie das Resultat und meldet dieses an den Client.

6.2.5.2 CHESSGP_Pool, der Client

Als Client fungiert genau eine Instanz der Klasse `CHESSGP_Pool`. Diese übernimmt die in Abschnitt [6.1.7](#) vorgestellten Selektionen und führt nach Eingang eines neuen Spielergebnisses ein Update der Elo-Zahlen durch.

7 Analyse und Ergebnisse

Dieses Kapitel stellt den Verlauf der Evolutionsläufe und die damit erzielten Ergebnisse dar. Es wird jeweils für die Experimente der einzelnen Stufen nachgezeichnet, welche Parameter dazu gewählt wurden, und welche Schlussfolgerungen aus den Ergebnissen – auch für weitere Experimente – gezogen werden können. Dabei handelt es sich um eine zusammenfassende Darstellung. Der tatsächliche Verlauf der Experimente war an vielen Stellen durch Wiederholungen, kleinere Tests und andere Abweichungen von dem hier im Folgenden skizzierten Verlauf gekennzeichnet. Hier sollen lediglich die wichtigsten Ergebnisse festgehalten werden.

Zunächst werden im folgenden Kapitel einige eigenentwickelte Tools beschrieben, mit denen Läufe durchgeführt und analysiert werden können. Nach einem Überblick über die Partiidatenbank folgt die Beschreibung der Experimente und Analysen für jede einzelne der drei Systemstufen.

7.1 Analysetools

Die Analysetools umfassen *Shellskripte* für Start und Archivierung der Evolutionsläufe und Programme, mit denen evolvierte Individuen getestet oder Bewertungsfunktionen auf ihre Eigenschaften hin überprüft werden können. Der Abschnitt *Datenbanktests* beschreibt diese Testprogramme und der Abschnitt *Random Walk* Untersuchungsmethoden der Fitnesslandschaften, die sich aus den Bewertungsfunktionen ergeben.

7.1.1 Shellskripte

Um den Verlauf einer Evolution im Nachhinein analysieren zu können, wird dieser in Protokolldateien festgehalten. Zur Verwaltung der Protokolldateien wurde eine Verzeichnisstruktur festgelegt, in welcher die Dateien abgelegt werden.

Um die Ablage der Protokolldateien in den vorgesehenen Verzeichnissen sicherzustellen und gleichzeitig den damit verbundenen Aufwand möglichst gering zu halten, wurden Shellskripten erstellt, mit denen ein Evolutionslauf gestartet werden kann, und welche für die Ablage der Protokolldateien sorgen.

7.1.1.1 Verzeichnisstruktur für Protokolldateien

Die Evolution der einzelnen Stufen ist nicht als Prozess zu sehen, welcher einmal angestoßen wird und an dessen Ende die funktionierenden Individuen stehen. Vielmehr sind eine Menge von Evolutionsläufen von Nöten, um zu einem zufriedenstellenden Individuum zu gelangen. Es wird mit den die Evolution beeinflussenden Parametern wie der Größe des Pools oder der Wahrscheinlichkeit eines Crossovers experimentiert, um den Verlauf der Evolution in die gewünschte Richtung zu lenken.

Bei jedem dieser Evolutionsläufe werden Protokolldateien angelegt, welche es ermöglichen, den Verlauf der Evolution nachzuvollziehen und die mögliche Auswirkung von Parametern auf diesen Verlauf zu analysieren. Dazu ist es erforder-

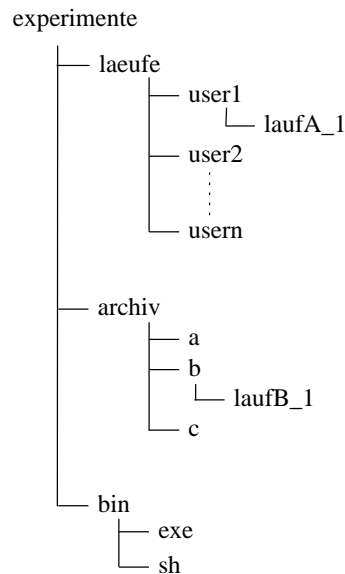


Abbildung 31: Verzeichnisstruktur

lich, Evolutionsläufe miteinander vergleichen zu können. Die Protokolldateien vergangener Läufe sollen strukturiert abgelegt werden, um aufwendiges Suchen oder gar ihren ungewollten Verlust zu vermeiden.

Hinzu kommt, dass nicht nur ein einziges PG-Mitglied Evolutionsläufe starten darf. Dabei ist zu verhindern, dass die Evolution des einen Mitglieds die Protokolldateien eines anderen überschreibt. Dennoch sollten die Protokolldateien allen Mitgliedern der Projektgruppe zugänglich sein.

Die in Abbildung 31 dargestellte Verzeichnisstruktur ist in der Lage, die Protokolldateien einer Vielzahl von Evolutionsläufen verschiedener PG-Mitglieder aufzunehmen. Sie wird im verbleibenden Teil des Abschnitts beschrieben.

Die Wurzel der Struktur ist das Verzeichnis **experimente**. Es existieren die Unterverzeichnisse **laeufer**, **archiv** und **bin**.

Das laeufer-Verzeichnis

Im **laeufer**-Verzeichnis werden die Evolutionsläufe der einzelnen PG-Mitglieder protokolliert. Es existiert für jeden Anwender ein Unterverzeichnis. Dieses enthält für jeden Lauf, den der Anwender gestartet hat, ein Unterverzeichnis, welches die Protokoll-Dateien dieses Laufes beinhaltet. Der Name des Unterverzeichnisses setzt sich zusammen aus dem beim Starten der Evolution angegebenen Laufnamen sowie einer Laufnummer. Diese wird beim Starten der Evolution automatisch ermittelt (vgl. Abschnitt 7.1.1.2) und sorgt für eindeutige Verzeichnisbezeichner.

Beispiel: Das PG-Mitglied Gustav möchte eine Evolution der Stufe a durchführen. Der Lauf soll „Materialzüge“ heißen. Bisher existiert noch kein Evolutionslauf dieses Namens. Später stößt Gustav auf einen weiteren Evolutionslauf mit dem gleichen Laufnamen an. Der erste Lauf erhält die Laufnummer 1, der

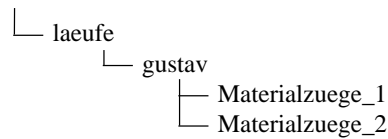


Abbildung 32: Läufe-Verzeichnis

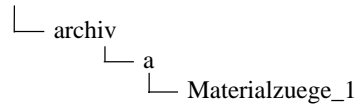


Abbildung 33: Archiv-Verzeichnis

zweite die 2. Das **laeufe**-Verzeichnis stellt sich nun wie in Abbildung 32 gezeigt dar.

Das **laeufe**-Verzeichnis dient also dazu, die Evolutionsläufe der PG-Mitglieder aufzunehmen. Hier werden während der Evolutionen die Protokolldateien – geordnet nach Mitglied und Evolution – angelegt.

*Das **archiv**-Verzeichnis*

Dieses Verzeichnis enthält die Protokolldateien von gesicherten Läufen der Anwender. Wurde ein Lauf eines Anwenders für wertvoll erachtet, weil er beispielsweise besonders gute Individuen hervorgebracht hat, so kann er ins Archiv übernommen werden.

Das **archiv**-Verzeichnis enthält entsprechend den Stufen Unterverzeichnisse **a**, **b** und **c**. Diese Verzeichnisse nehmen die Protokolldateien eines zu archivierenden Evolutionslaufes auf.

Beispiel: Möchte Anwender Gustav den zweiten Lauf „Materialzüge“ sichern, so verschiebt er ihn ins Archiv, so dass dieses danach wie in Abbildung 33 dargestellt aussieht.

Da das Archiv Läufe mehrerer PG-Mitglieder aufnehmen kann, entspricht die Laufnummer des Archivs nicht mehr der Nummer der ursprünglichen Evolution.

*Das **bin**-Verzeichnis*

Im Unterverzeichnis **sh** des **bin**-Verzeichnisses liegen die zur Evolution zur Verfügung stehenden Shellskripten, welche im Abschnitt 7.1.1.2 und 7.1.1.3 beschrieben werden.

Das Unterverzeichnis **exe** dient dazu, die einer Evolution zugrundeliegenden Executables aufzunehmen. Zur Analyse eines Evolutionslaufes muss dieses bekannt sein. Das Executable beeinflusst direkt die Evolution. In ihm ist beispielsweise festgelegt, auf welche Weise die Mutation eines Individuums vorgenommen wird.

Für jedes Executable existiert ein eigenes Unterverzeichnis, so dass der **exe**-Zweig beispielsweise wie in Abbildung 34 gezeigt aussieht.

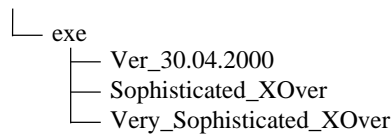


Abbildung 34: Exe-Verzeichnis

In den jeweiligen Unterverzeichnissen befinden sich die auszuführenden Programme `evolChessGP_A`, `evolChessGP_B` und `evolChessGP_C` sowie eine kurze Beschreibung des Executables.

7.1.1.2 Die Shellskripten zum Start eines Evolutionslaufes

Ein Evolutionslauf wird mittels Shellskripten gestartet. Es existieren die Skripten `startEvolA`, `startEvolB` und `startEvolC`, deren Unterschied hauptsächlich im aufgerufenen Executable besteht. Es wird daher hier stellvertretend das Skript `startEvolA` beschrieben. Die beiden anderen laufen weitgehend in gleicher Weise ab.

Die Hauptaufgabe des Shellskripts besteht darin, die während der Evolution erzeugten Protokolldateien automatisch in die in Abschnitt 7.1.1.1 beschriebene Verzeichnisstruktur einzugliedern. Beim Aufruf des Shellskripts muss neben den zu verwendenden Strukturdateien eine Bezeichnung des Evolutionslaufes erfolgen, der Laufname muss angegeben werden. Das Skript legt, falls noch nicht vorhanden, im `laeufer`-Verzeichnis ein Unterverzeichnis für das PG-Mitglied an und erstellt in diesem ein Verzeichnis für die zu startende Evolution. Dieses Verzeichnis wird nach dem Laufnamen und einer Laufnummer bezeichnet. Die Nummer wird vom Skript ermittelt, indem beginnend mit der Nummer 1 getestet wird, ob schon ein Lauf des angegebenen Namens mit dieser Nummer angelegt wurde. Ist dies nicht der Fall, wurde ein eindeutiger Verzeichnisname gefunden, ansonsten wird die nachfolgende Nummer ausprobiert. Es sei darauf hingewiesen, dass die Laufnummer nicht unbedingt die Reihenfolge widerspiegelt, in der die Evolutionsläufe durchgeführt wurden. Die Protokolldateien des Evolutionslaufes werden nun während der Evolution in dem neu erzeugten Verzeichnis angelegt, ein versehentliches Überschreiben älterer Protokolldateien wird verhindert.

Die Auswahl des aufzurufenden Executables wird über die Shell-Variable `CHESSGPEXEC` getroffen. In dem durch die Variable angegebenen Verzeichnis wird nach dem Executable `evolChessGP_A` gesucht. Als Executables sind die unter dem Verzeichnis `experimente/bin/exe/` abgelegten Programme vorgesehen.

Zur Beschreibung des Evolutionslaufes legt das Skript im Verzeichnis, welches die Protokolldateien aufnimmt, zusätzlich die Datei `kommentar.txt` an. In dieser wird der Zeitpunkt des Starts des Evolutionslaufes sowie der Pfad des benutzten Executables festgehalten. Außerdem enthält die Datei einen Kommentar des PG-Mitglieds, welches den Evolutionslauf gestartet hat. Dieser Kommentar kann aus einer Datei eingelesen werden, welche beim Start des Skriptes angegeben wird, oder nach Aufruf des Shellskriptes in einem Editor verfasst werden.

Ein Aufruf des Skriptes, durch welchen ein Evolutionslauf der Stufe a gestartet wird, kann wie folgt aussehen:

```
startEvoLA stufe_a.str Materialzuege komm.txt
```

In diesem Fall wird das Strukturfile `stufe_a.str` verwendet. Der Lauf soll den Namen „Materialzüge“ erhalten, der Kommentar zu diesem Lauf soll der Datei `komm.txt` entnommen werden.

7.1.1.3 Die Shellskripten zum Archivieren von Protokolldateien

Das Archivieren von Protokolldateien erfolgt ebenfalls mittels Shellskripten. Zum Einen erleichtert das den Vorgang des Verschiebens von Protokolldateien ins Archiv, zum Anderen sichert es die Konsistenz des Archivs.

Es existieren die Shellskripten `moveEvoLA`, `moveEvoLB` und `moveEvoLC`, von welchen stellvertretend `moveEvoLA` beschrieben wird.

An für sich führt das Shellskript ein einfaches Verschieben der Protokolldateien eines Evolutionslaufes aus dem `laeufe`-Verzeichnis ins `archiv`-Verzeichnis durch. Der zu verschiebende Lauf wird über seinen Laufnamen sowie seine Laufnummer identifiziert. Falls das aufrufende PG-Mitglied nicht einen seiner Läufe verschieben will, muss außerdem der Name des „Besitzers“ des Laufs angegeben werden. Im Archiv wird nun ein Verzeichnis erstellt, dessen Name sich aus dem Laufnamen sowie einer neu ermittelten (Archiv-)Laufnummer zusammensetzt. Die Laufnummer wird auf gleiche Weise wie in Abschnitt 7.1.1.2 beschrieben ermittelt.

Darüberhinaus wird die Inhaltsdatei `inhalt.txt` des Verzeichnisses `archiv/a/` aktualisiert. In dieser Datei sind alle Läufe des Archivs aufgelistet. Sie enthält neben Angaben wie dem Zeitpunkt des Übernehmens ins Archiv und dem Namen des PG-Mitglieds, welches den Lauf durchgeführt hat, den Inhalt der Kommentardatei `kommentar.txt` dieses Laufes. Anhand der Inhaltsdatei kann ein gesuchter Lauf im Archiv schnell gefunden werden.

Folgendes Beispiel zeigt einen Aufruf des Skripts:

```
moveEvoLA Materialzuege 2
```

Es werden die Protokolldateien zum Lauf „Materialzuege“ mit der Laufnummer 2 aus dem `laeufe`-Verzeichnis des Anwenders ins Verzeichnis `archiv/a/` verschoben.

7.1.2 Datenbanktests

Im Rahmen der *Datenbanktests* können einzelne Individuen auf ihre Leistungsfähigkeit bezüglich bestimmter Schachbretter getestet werden. Während sich der Fitnesswert eines Individuums im Evolutionslauf aus mehreren Komponenten zusammensetzt,¹⁰¹ werden beim direkten Datenbankvergleich die zu

¹⁰¹ Siehe auch die Beschreibungen Fitnessberechnung in Stufe a (Abschnitt 6.2.4) und b (Abschnitt 6.2.4.3).

einem Schachbrett gehörenden Werte aus der Datenbank den von dem Individuum berechneten Ergebnissen gegenübergestellt. Da ein Datenbanklernen nur in den Stufen a und b stattfindet, wird die genaue Vorgehensweise nur für diese beiden Stufen im Folgenden näher erläutert.

7.1.2.1 Stufe a

Das Lernziel für die Individuen der Stufe a bestand in der Zuordnung eines numerischen Wertes zu einem Schachbrett. Dieser Wert stellt die Rückgabe einer Bewertungsfunktion zu einem Schachbrett dar.¹⁰² Dazu standen acht verschiedene Bewertungsfunktionen zur Verfügung. Jedem Brett in der Datenbank sind in der Stufe a somit acht Werte zugeordnet. Das trainierte Individuum soll dazu in der Lage sein, zu einem beliebigen Schachbrett unter Vorgabe einer Bewertungsfunktion den jeweils zugehörigen Wert zurückzugeben.

Zu Beginn des Datenbanktests müssen die Bewertungsfunktionen und die Anzahl der zu durchlaufenden Schachbretter in der Datenbank festgelegt und das zu testende Individuum eingelesen werden. Der Datenbanktest für die Stufe a durchläuft nun alle gewünschten Schachbretter und übergibt jedes einzelne an das Individuum, um dessen Rückgabewert zu diesem Brett für die zu lernende Bewertungsfunktion abzufragen. Die Differenz zwischen dem Wert der Bewertungsfunktion in der Datenbank und dem Rückgabewert des Individuums ist der *Abweichungsfehler*.

Die Ausgabe besteht aus einer Tabelle, welche die Nummer des Schachbrettes, den dazu in der Datenbank abgelegten Wert der gewählten Bewertungsfunktion, den Rückgabewert des Individuums und den Abweichungsfehler enthält. Zusätzlich wird der durchschnittliche Gesamtfehler auf allen untersuchten Brettern gebildet.¹⁰³

Mit dem Gesamtfehler und den Einzelabweichungen ist es möglich, das Verhalten des Individuums – bezüglich der Zuordnung einer Bewertung zu einem Schachbrett – darzustellen. Das kann sowohl eine Untersuchung einiger einzelner Bretter (durch direkten Vergleich von Datenbank- und Rückgabewert), als auch das Verhalten des Individuums auf allen Schachbrettern sein. Einfache Skripte ermöglichen eine grafische Ausgabe der Abweichungen.

7.1.2.2 Stufe b

In Stufe b sollen die GP-Individuen lernen, zu den Schachstellungen in der Datenbank Zugvorschläge zu tätigen.¹⁰⁴ Die Rückgabe des Individuums besteht in einer sortierten Liste der vorgeschlagenen Züge.

Ein einfacher numerischer Vergleich wie im Datenbanktest der Stufe a, der lediglich einen abweichenden Fehler durch einfache Subtraktion berechnet, ist hier nicht mehr möglich. Stattdessen wird dem Benutzer die Möglichkeit gegeben, genau anzugeben, auf welchen Schachbrettern das Individuum getestet

¹⁰² Siehe auch hierzu die Erläuterungen zum Konzept der Stufe a in Abschnitt 6.1.4

¹⁰³ Dazu werden alle Absolutwerte des Abweichungsfehlers summiert und am Schluss durch die Anzahl der untersuchten Bretter geteilt.

¹⁰⁴ Siehe auch hierzu die Erläuterungen zum Konzept der Stufe b in Abschnitt 6.1.5

werden soll. Für alle diese Schachbretter werden dann die Stellung und die Zugvorschläge des Individuums in eine Ausgabedatei geschrieben. So kann von einem menschlichen Schachexperten die Qualität der zurückgelieferten Zuglisten untersucht werden. Quantitative Auswertungen, die als Graphen dargestellt werden könnten, entfallen.

7.1.3 Random Walk

Ein *Random Walk* gibt Einblicke in die Struktur der *Fitnesslandschaft* des untersuchten Problems. Eine *Fitnesslandschaft* ist eine Zuordnung der Genome zu den Fitnesswerten. Alle möglichen Genome sind dabei als Knoten in einem gerichteten Graphen so angeordnet, dass man durch die Anwendung eines genetischen Operators (z. B. Mutation) von einem Knoten zu seinen Nachbarn gelangen kann. Interessant sind dabei die Bereiche der Fitnesslandschaft, in welchen optimale Fitnesswerte liegen. Das Ziel der Untersuchungen ist es, herauszufinden, wie stark die Landschaft „zerklüftet“ ist, und ob Regionen mit optimalen Werten einfach zu finden oder – auf „Felsspitzen“ oder in „schmalen Tälern“ – versteckt sind.

Ein *Random Walk* ist nun eine zufällige Bewegung auf dieser Landschaft. Je nach Differenz der Fitnesswerte benachbarter Genotypen lassen sich Rückschlüsse auf den Grad der „Zerklüftung“ der Landschaft und somit auf die Komplexität des zu lernenden Problems ziehen. Der Ablauf des *Random Walks* erfolgt als zufällige Veränderung eines einzelnen Individuums ohne anschließende Selektion. Dadurch werden nicht nur die besten Individuen betrachtet, sondern ein Pfad durch die Fitnesslandschaft, der sowohl Verbesserungen als auch Verschlechterungen zulässt. Jede Veränderung durch einen genetischen Operator führt zu einem benachbarten Knoten und dessen zugeordnetem Fitnesswert. So kann beispielsweise ein Random Walk auf einer Binärstringrepräsentation ein einzelner Bitflip in jedem Schritt sein.

Sind die Veränderung der zugehörigen Fitnesswerte bei diesen Schritten sehr groß, ist die Landschaft stark zerklüftet. Eine Methode zur Erkennung der Zerklüftung ist die Anwendung einer Autokorrelationsfunktion auf einzelne Zeitreihen, die durch Random Walks produziert wurden. Hierbei wird der Zusammenhang der Fitnesswerte in unterschiedlichen Zeitabständen untersucht. Da wir in unserem Fall die Zeitreihen aus den Random Walks nicht näher statistisch untersucht und nur zum Überblick genutzt haben, soll hier auf eine detailliertere Beschreibung verzichtet werden. Einen guten Einblick und weitere Einstiegspunkte findet sich z. B. in [KAU93].

Unser Tool zur Erzeugung von Zeitreihen mit Random Walks verfügt über zwei Modi:

- *Mutation*: Ausgehend von einem Startindividuum wird in jedem Schritt eine gewünschte Anzahl Knoten mutiert. Im einfachsten Fall ist das genau ein Knoten, und anhand der Fitnesswerte über der Zeit kann beobachtet werden, wie stark sich diese nach den durchgeführten Mutationen verändern.

- *Crossover*: In dieser Variation des Random Walk wird mit zwei Individuen gearbeitet. In jedem Schritt werden die Fitnesswerte der Individuen berechnet, beide miteinander rekombiniert und anschliessend jedes der aus der Rekombination entstandenen neuen Individuen mutiert. Anschliessend werden die Eltern mit den neu entstandenen Nachkommen überschrieben. Die Anzahl der zu verändernden Knoten wird auch hierbei wieder durch einen Parameter vom Benutzer vorgegeben. Ziel dabei ist es, den Einfluss des Crossovers auf den Evolutionsverlauf näher zu untersuchen. Dabei konnte nicht ganz auf die Mutation verzichtet werden, da die Anwendung des Crossovers alleine nicht genug Veränderung verursacht, um Pfade in der Fitnesslandschaft zu erzeugen.

In beiden Modi besteht die Möglichkeit, durch die Angabe einer Poolgröße beliebig viele Random Walks mit den gewünschten Einstellungen gleichzeitig durchzuführen. Dabei ist in diesem Fall mit *Pool* nicht die Menge der Individuen, die untereinander verknüpft werden, gemeint, sondern lediglich die Speicherung der *getrennt* arbeitenden Random Walks. Shellskripte ermöglichen darüberhinaus eine bequeme Anzeige und Auswertung.

7.2 Analyse der Partiidatenbank

In diesem Abschnitt wird die statistische Verteilung der Datenbankeinträge, auf denen die Fitness-Berechnung beruht, einmal genauer betrachtet. In der Datenbank sind für Stufe a jeder Stellung genau acht Bewertungsfunktionen¹⁰⁵ zugeteilt, die teilweise sehr unterschiedliche Wertebereiche haben.

Die „normale“ *Materialbewertung* bildet eine gewichtete Summe aller Schachsteine wobei die gegnerischen (also schwarzen) Steine negativ in die Summe eingehen. In Abbildung 35 sieht man, dass fast die Hälfte der ca. 40 000 in der Datenbank vorhandenen Stellungen einen Materialwert von 0 haben, und weitere 25% der Stellungen mit ± 100 bewertet sind. Da bei der Materialbewertung ein Bauer mit ± 100 gezählt wird (und die anderen Steine entsprechend höher) heißt das also, dass sich in drei Vierteln der Stellungen das Materialverhältnis zwischen den beiden Spielern um höchstens einen Bauern unterscheidet. Ein Materialunterschied von mehr als drei Bauern ist schon extrem selten, und ein größerer Unterschied als fünf Bauern (was einem Turm entspricht) kommt nur noch relativ selten vor. Dieser Umstand ist dadurch zu erklären, dass die Stellungen allesamt realen Partien¹⁰⁶ entstammen, die schon „sehr früh“, d. h. oft schon bei Verlust eines Bauern, aufgegeben wurden. Auf den Lernerfolg unserer GP-Individuen wirkt sich dieses jedoch sehr nachteilig aus, da ein Individuum, welches unabhängig von der jeweiligen Schachstellung konstant „0“ berechnet, schon „ziemlich gut“ ist. – Es „verrechnet“ sich auf zwei Drittel der Stellungen um höchstens einen Bauern. Solche Individuen verdrängen also jene, die eine „wirkliche“ Stellungsbewertung durchführen und sich dabei um mehr als einen Bauern „verrechnen“.

¹⁰⁵ Siehe auch Abschnitt 6.1.4.1.

¹⁰⁶ Siehe auch Abschnitt 5.2.

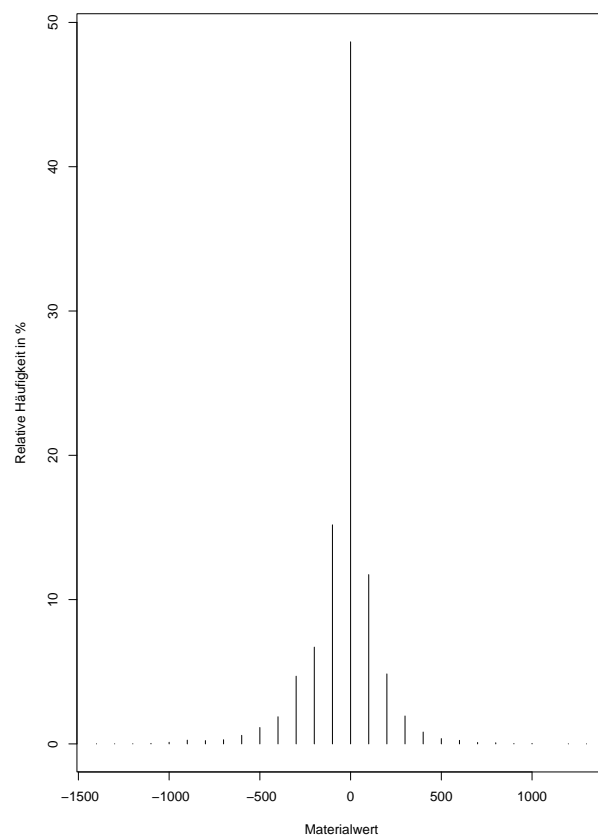


Abbildung 35: Relative Häufigkeiten der Materialwerte in der Datenbank

Es gibt verschiedene Möglichkeiten, diesem Dilemma zu begegnen:

- Zusätzliches Einfügen „neuer Schachstellungen“ in die Datenbank, die aus den bereits vorhandenen dadurch erzeugt werden, dass auf einer Seite an zufälligen Positionen weitere Schachsteine hinzugefügt werden.
- Aufgegebene Partien bis zum Matt weiter spielen und die sich ergebenden Stellungen in die Datenbank aufnehmen.
- Bei der Auswahl der Fitness-Cases aus der Datenbank darauf achten, dass seltenere Stellungstypen häufiger gewählt werden.

Die erste Möglichkeit ist eher nicht zu empfehlen, da nicht ausgeschlossen werden kann, dass dadurch Stellungen entstehen, die in einem realen Schachspiel nie vorkommen würden. Für die Materialbewertung mag das zwar nicht zwangsläufig nachteilig sein, allerdings werden die Stellungen auch für andere Bewertungen benutzt¹⁰⁷ und da kann es sehr gut passieren, dass das Hinzufügen eines Steines z. B. eine eindeutig defensive Stellung zerstört. Die Stellungen für die Datenbank wurden ja auch mit Bedacht ausgewählt, da wir die Hoffnung haben, dass die GP-Individuen eigene Heuristiken erlernen, um nicht wie Brute-force-Schachprogramme alle prinzipiell möglichen Züge betrachten zu müssen, sondern nur diejenigen, die 1. in richtigen Schachspielen vorkommen und 2. möglichst vielversprechend aussehen.

Bedeutend besser ist da schon die zweite Möglichkeit, weil hierbei nur „reale“ Schachstellungen entstehen und eigentlich nur das explizit ausgeführt, bzw. in die Datenbank aufgenommen wird, was die an der jeweiligen Partie beteiligten Spieler als offensichtlich ansahen, und folglich wegließen. Daher haben wir uns auch entschieden, diese Methode zu verfolgen, wobei das Zuendespielen der Partien von dem Schachprogramm Crafty übernommen wird, dass abwechselnd für Schwarz und für Weiß zieht.

Die dritte Methode besteht darin, dass man durch das Hinzufügen von Wahrscheinlichkeiten für die Wahl einer Partie als Fitness-Case sozusagen künstlich eine Gleichverteilung der Datensätze erzeugt. Sie ist vor allem als Ergänzung zu der Möglichkeit des Zuendespiels von Partien sehr gut geeignet und wurde von uns in Stufe b verwendet.

Die *clevere Materialbewertung* unterscheidet sich von der oben genannten „normalen“ Materialbewertung dadurch, dass hierbei nun auch die Position der einzelnen Schachsteine auf dem Brett mit in die Bewertung eingeht. Abbildung 36 zeigt die Verteilung der Bewertungen in der Datenbank: Auf den ersten Blick unterscheidet sie sich nicht allzusehr von der Verteilung der „normalen“ Materialwerte. – Bis auf die Peaks bei ± 100 (also einem Bauernwert) und ± 300 (also einem Läufer- oder Springerwert). Wenn man das Diagramm jedoch genauer betrachtet, fällt auf, dass die Ordinatenachse gänzlich anders skaliert ist. So ist zwar immer noch die Bewertung „0“ häufiger als jede andere Bewertung vorhanden, allerdings macht das nicht mehr fast die Hälfte der Bewertungen aus, sondern nur noch gut 0,9%. Eine Erklärung dafür ist, dass die Bewertung nun

¹⁰⁷ Siehe auch Abschnitt 6.1.4.1.

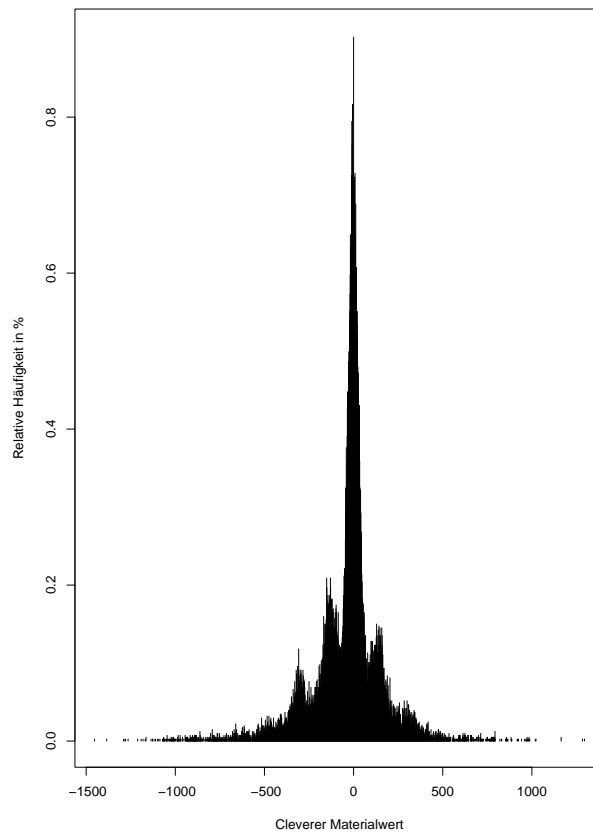


Abbildung 36: Relative Häufigkeiten der cleveren Materialwerte in der Datenbank

nicht mehr nur diskrete Werte (in 100er-Schritten) annehmen kann, und daher jetzt viele Werte zwar $\neq 0$ sind, aber trotzdem sehr nah bei 0 liegen. Die oben genannten Verbesserungen sind also auch hier notwendig.

Die *Aggressivitätsbewertung*, die aus Sicht des Spielers am Zug bewertet, wie aggressiv eine Stellung ist, ist, wie man in Abbildung 37 sieht, ebenfalls nicht sonderlich gut verteilt: Beinahe ein Viertel der Werte ist 0, und insgesamt fast 70% der Werte sind 0, 1 oder 2. Hier muss man allerdings beachten, dass die Bewertungen im Vergleich zu den anderen Bewertungsfunktionen einem sehr kleinen Wertebereich (mit nur 14 möglichen Werten) entstammen.

Auch bei der Verteilung der *Angriffsbewertung*, die in Abbildung 38 dargestellt ist, sind relativ viele Werte gleich 0 bzw. nah bei 0 angesiedelt, was auch hier zu Problemen führt. Die Angriffsbewertung gibt an, inwiefern die Stellung des Spielers, der gerade am Zug ist, eine Angriffsstellung ist.

Nach diesen problematischen Verteilungen seien der Vollständigkeit wegen noch kurz die restlichen Verteilungen der Bewertungsfunktionen aufgezählt: Die

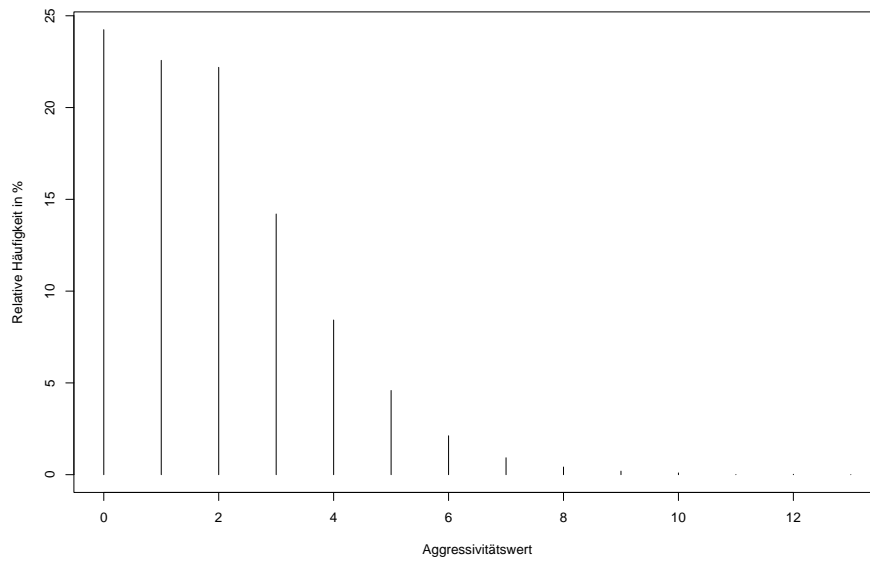


Abbildung 37: Relative Häufigkeiten der Aggressivitätswerte in der Datenbank

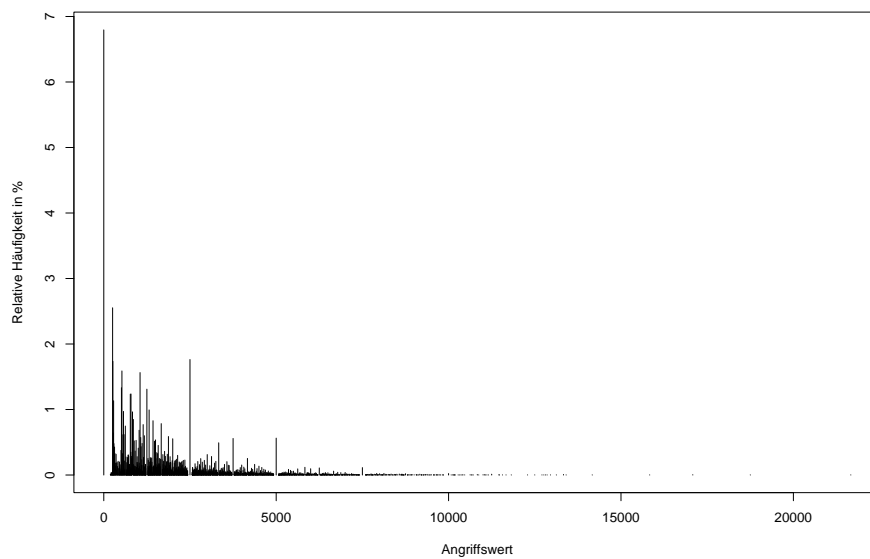


Abbildung 38: Relative Häufigkeiten der Angriffswerte in der Datenbank

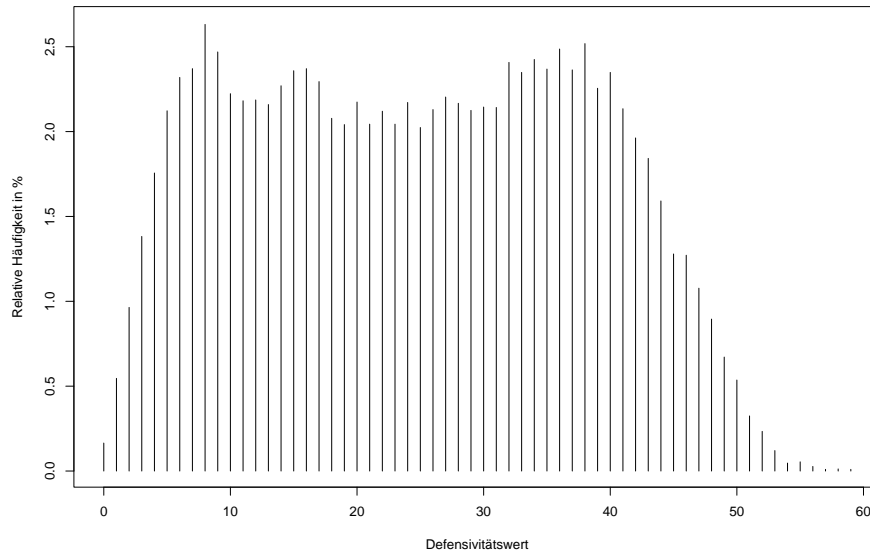


Abbildung 39: Relative Häufigkeiten der Defensivitätswerte in der Datenbank

Defensivitätsbewertung, die ein Maß dafür liefert, wie defensiv eine Stellung des Spielers am Zug ist, hat für unsere Zwecke die schönste Verteilung, da sie sich relativ gleichmäßig über den gesamten Wertebereich erstreckt (Abbildung 39). Auch die *Komplexitätsbewertung*, die – wie man in Abbildung 40 sieht – annähernd normalverteilt um den Wert 80 ist, bringt keine großen Probleme mit sich. Ebenfalls unproblematisch sind die Verteilungen der *Endspielbewertung*, die angibt, in welchem Maße ein Endspielstatus vorliegt (Abbildung 41) und der *Verteidigungsbewertung*, die ein Maßstab dafür ist, ob und inwieweit der Spieler am Zug verteidigend steht (Abbildung 42).

7.3 Analyse der Stufe a

7.3.1 Vorgehen

Im Folgenden wird ein Überblick über die Experimente und Analysen zur Evolution von Bewertungsfunktionen der Stufe gegeben. Dies erfolgt weitestgehend in der chronologischen Reihenfolge der Durchführung dieser Experimente. Allerdings werden nicht alle Läufe dokumentiert, sondern eine Auswahl, anhand welcher einige Zwischenergebnisse nachgezeichnet werden.

Zunächst einmal wird in Abschnitt 7.3.2 der zugrundeliegende Parametersatz tabellarisch dargestellt und die wichtigsten Parameter werden erläutert. Für unsere Evolutionsläufe unrelevante Parameter werden nicht berücksichtigt. Grundsätzlich lassen sich die Parameter in solche des zugrundeliegenden GP-Systems SYSGP und eigene Parameter (hier als *Userparameter* bezeichnet) unterscheiden. Die ersten Läufe erfolgten mit identischen Parametersätzen. Die

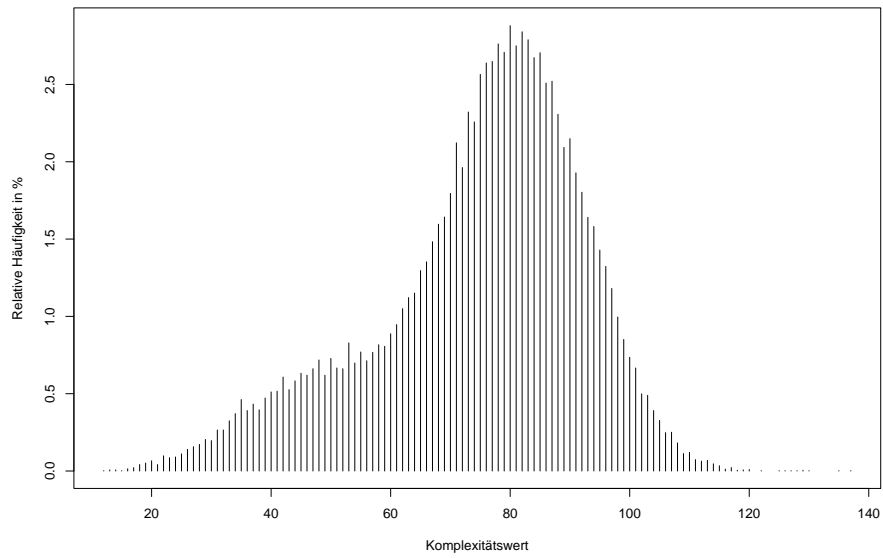


Abbildung 40: Relative Häufigkeiten der Komplexitätswerte in der Datenbank

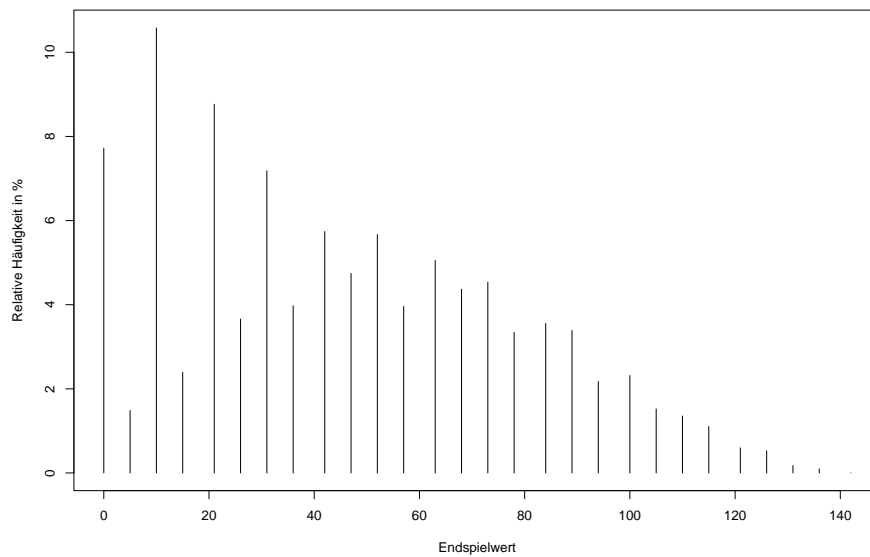


Abbildung 41: Relative Häufigkeiten der Endspielwerte in der Datenbank

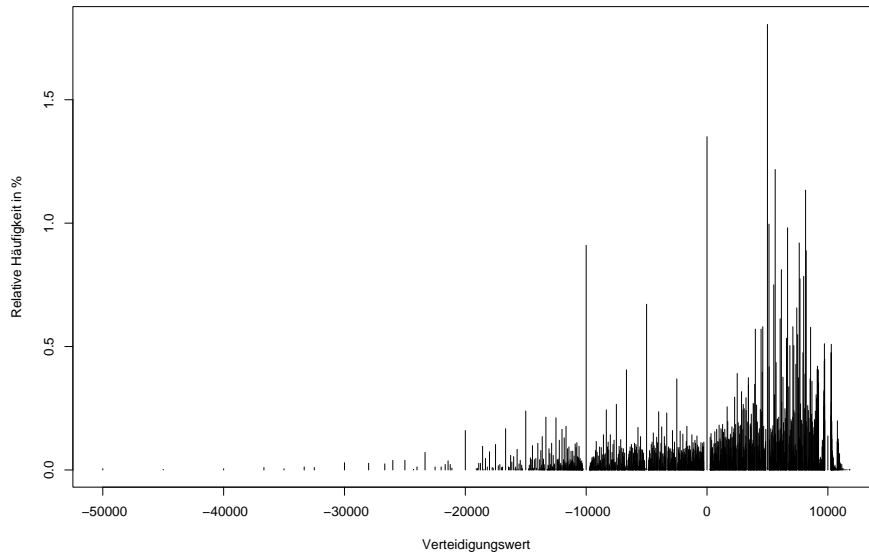


Abbildung 42: Relative Häufigkeiten der Verteidigungswerte in der Datenbank

späteren Parameteränderungen betrafen hauptsächlich den Bereich der Userparameter. Änderungen der SYSGP-Parameter erfolgten erst in späteren Phasen und hauptsächlich in Stufe b.

Schwerpunktmässig werden – nach einem Überblick über die Testphase der Stufe a (Abschnitt 7.3.3) – drei Serien aus den Evolutionsläufen betrachtet, von denen die zugehörigen Graphen sämtlicher acht Bewertungsfunktionen angegeben werden: Eine Serie *ohne* Restart-Mechanismus (in Abschnitt 7.3.4), eine Serie *mit* Restart-Mechanismus (in Abschnitt 7.3.5) und eine Serie mit Restart und eingeschränkter Operationenmenge (in Abschnitt 7.3.6).¹⁰⁸ Anschließend folgen zwei Abschnitte mit Analyse-Läufen: Datenbanktests in Abschnitt 7.3.7 und Random Walks in Abschnitt 7.3.8.¹⁰⁹ Innerhalb der Beschreibung der Serien werden auch einzelne Versuche und deren Ergebnisse vorgestellt.¹¹⁰

7.3.2 Parameter der Stufe a

Hier erfolgt eine Auflistung aller wichtigen Parameter. Die SYSGP-Parameter sind in Tabelle 7 auf der nächsten Seite erläutert.

Auf diesen Parametern beruhen im Wesentlichen die folgenden Serien aus Evolutionsläufen der Stufe a. Verändert wurden in den später dargestellten Läufen lediglich die Populationsgröße von 100 auf den hier schon dargestellten Wert von 1000 und im weiteren Verlauf der Experimente die Grenzen der Kon-

¹⁰⁸ Zur Bedeutung des Restart-Mechanismus siehe Abschnitt 6.1.3.

¹⁰⁹ Eine Beschreibung der Analysetools findet sich in 7.1.

¹¹⁰ Die Bedeutung der Fitnesswerte aller folgenden Kurven wird in Abschnitt 6.2.4 erläutert.

Bezeichnung	Wert	Erklärung
Population	1 000	Größe der Population
Generation	1 000	Anzahl der Generationen (Laufzeit)
RegisterSize	4	Anzahl der Register im Environment
Crossoverrate	80	Häufigkeit des Crossover in Prozent
Mutationrate	30	Häufigkeit der Mutationen in Prozent
mSeed	0	Seed des Zufallszahlengenerators (0: Seed erzeugen)
MainTyp	L	Programmtyp (L: Linear)
MainMutCnt	1	Anzahl der Knoten, die während einer Mutation verändert werden
MainInitTyp	halfAndHalf	Initialisierungsart
MainInitNodeCnt	25	maximale Knotenanzahl während der Initialisierung
MainInitMinNodeCnt	5	minimale Knotenanzahl während der Initialisierung
MainNodeCnt	50	maximale Knotenanzahl der Individuen während der Evolution
MainMinNodeCnt	10	minimale Knotenanzahl der Individuen während der Evolution
MainConstProb	50	Prozentuale Wahrscheinlichkeit, dass ein Terminal eine Konstante ist
MainTermProb	20	Prozentuale Wahrscheinlichkeit, dass ein Knoten ein Terminal ist
MinConstant	-1 000	untere Schranke der Konstanten
MaxConstant	1 000	obere Schranke der Konstanten

Tabelle 7: Grundlegende SYSGP-Parameter

Bezeichnung	Wert	Erklärung
probFitnessCases	30	Wahrscheinlichkeit, mit der ein Fitness-Case pro Generation zufällig neu gezogen wird
fitnessType	Aggressiv	Bezeichner der Bewertungsfunktion
chessboardNumber	40 000	Anzahl der in der Datenbank vorhandenen Schachbretter
fitnessCasesA	100	Anzahl der zufällig zum Lernen zu wählenden Bretter

Tabelle 8: Grundlegende User-Parameter

stanten (*MinConstant*, *MaxConstant*). Veränderungen der *Crossoverrate* und der *Mutationrate* und der Individuengröße wurden erst in Stufe b durchgeführt. Die vollständigen Mengen der Operationen und Terminale beinhalten¹¹¹:

- *Operationen*: boardLoop rowLoop lineLoop pieceLoop add sub mul div setAll setRow setLine setPiece setValue ifThen rowShift lineShift northWestShift northEastShift addequal subequal mulequal divequal whichPiece absolute loadAll loadPiece loadLine loadValue loadRow storeAll storePiece storeValue storeLine storeRow swap
- *Terminale*: pieceValue attackedByValue attackedByNumber legalMovesNumber

Die Userparameter der ersten Läufe sind in Tabelle 8 erklärt.

In den nachfolgenden Läufen wurden hier einige Ergänzungen vorgenommen u.a. zur Durchführung des Restarts. Diese werden an den entsprechenden Stellen im Verlauf erläutert. Hier ist zunächst nur festzuhalten, dass sich der Lernvorgang am Anfang auf 100 Schachbretter aus einem Gesamtvolumen von 40 000 Schachbrettern in der Datenbank beschränkte. Später wurde die Datenbank um von Crafty gespielte Endspielstellungen erweitert.¹¹² Darauf wird auch bei der Beschreibung der Experimente an entsprechender Stelle verwiesen. Der Wert *probFitnessCases* von 30 sagt aus, dass in jeder Generation jeder Fitnesscase mit einer Wahrscheinlichkeit von 30% neu gewürfelt wird. Der Wertebereich der Fitnesscases liegt innerhalb der Anzahl der zur Verfügung stehenden Schachbretter in der Datenbank (*chessboardNumber*).¹¹³

¹¹¹ Die Operationen werden in 6.1.4.4 näher erläutert.

¹¹² Eine Beschreibung von Crafty findet sich in Abschnitt 4.10.

¹¹³ Eine genauere Beschreibung der Fitnesscase-Auswahl findet sich in Abschnitt 5.2.1.

7.3.3 Die ersten Testläufe

7.3.3.1 Zielsetzung

Die ersten durchgeführten Experimente mit der Evolution von Individuen der Stufe a für eine einfache statische Materialbewertung verfolgten hauptsächlich drei Ziele, nämlich

- die Korrektur semantischer Fehler,
- die Verbesserung der Evolutionsgeschwindigkeit, sowie
- allgemeine GP-Verbesserungen im Hinblick auf das Evolutionsziel¹¹⁴.

Da man bei hinreichend komplexen Computerprogrammen semantische Fehler nie ganz ausschließen kann, müssen diese ausgiebig getestet werden, um alle Fehler zu finden. Die Funktionen/Befehle, die den zu evolvierenden Schachprogrammen zur Verfügung gestellt werden, können am besten getestet werden, indem man sie „einfach benutzt“, d. h. indem man Programme evolviert, und untersucht, ob die verwendeten Befehle das bewirken, was der Programmierer intendiert hat. Weil bei diesen Tests schon alles benutzt wird, was später bei der „richtigen“ Evolution benötigt wird, also z. B. die schachspezifischen Funktionen der Klasse `CHESSGP_Board`, der Zugriff auf die Datenbank, die Generierung von Zufallszahlen für die benötigten Konstanten, die genetischen Operatoren wie Selektion, Crossover und Mutation, und vieles weitere mehr, können auch diese Komponenten – und vor allem auch deren Zusammenspiel – getestet werden.

Bei dem zweiten Ziel, der Verbesserung der Evolutionsgeschwindigkeit, geht es darum, gegebenenfalls Schwachpunkte auszumachen, die zu einer schlechten Performance führen.

Unter den dritten Punkt fallen vor allem solche Läufe, in denen zwar keine „direkten“ Fehler auftraten, aber dennoch nicht die erwartete Entwicklung der Individuen stattfand.

7.3.3.2 Durchführung

Den Anfang machten einfache Tests, bei denen die Populationsgröße (also die Anzahl der gleichzeitig existierenden Stellungsbewertungsprogramme) und die Anzahl der Fitness-Cases (also die Anzahl der Schachstellungen, auf denen sich die Programme zu bewähren haben) verändert wurden, da die Vermutung besteht, dass größere Populationen besser dazu in der Lage sind, das Evolutionsziel zu erreichen als kleinere. Es zeigte sich hierbei allerdings auch das wenig überraschende Ergebnis, dass die Geschwindigkeit sehr stark von der Populationsgröße abhängt. Die Anzahl der Fitness-Cases hat ebenfalls einen – wenn auch nicht ganz so starken – Einfluss auf die Laufzeit.

¹¹⁴ Mit „Evolutionsziel“ ist die Senkung des Fitnesswertes auf 0 und somit die Verbesserung der Fitness gemeint. Wenn die Bewertungs- und Fitnessfunktionen korrekt sind, ist ein Individuum mit Fitnesswert 0 eines, welches eine „gute“ Bewertungsfunktion darstellt.

Bei weiteren Versuchen stellte sich heraus, dass eine unerwünschte Abhängigkeit zwischen der Anzahl der Fitness-Cases und der Fitness bestand: Bei wenigen Fitness-Cases ging der Fitnesswert extrem schnell gegen 0, bei mehr (über 100) Fitness-Cases verbesserte er sich jedoch fast gar nicht. Außerdem hing der Startwert der Fitnesswerte mit der Anzahl der Fitness-Cases zusammen. Diese Umstände konnten durch die Korrektur semantischer Fehler und die Konstanthaltung der Fitness-Cases innerhalb einer Generation (s. u.) beseitigt werden.

Weiterhin wurde die Abhängigkeit des Erreichens des Evolutionszieles von der Auswahl der Fitness-Cases untersucht. Dabei stellte sich heraus, dass es günstiger ist, wenn sich die Individuen einer Generation auf die *selbe* Teilmenge aus allen Fitness-Cases beziehen, anstatt dass sie jedes Mal neu gewählt wird. Diese Teilmenge wird daher genau einmal pro Generation zufällig bestimmt.

Nach einigen Testläufen wurde festgestellt, dass keine Terminale¹¹⁵ verwendet wurden. Dieses lag daran, dass die Terminalwahrscheinlichkeit aus Gründen, die mit SYSGP zu tun hatten, auf 0 gestellt worden war. Als Lösung dieses Problems wurden (durch Änderungen in SYSGP und der Datei `Function_A.hh`) bei den Operationen nun auch Rückgabewerte zugelassen, so dass Terminale verwendet werden konnten.

Als nächstes wurde nach dem Grund gesucht, warum nur extrem hohe Fitnesswerte vorkamen und sich kaum verbesserten. Dazu wurden die Populationsgröße, die Fitness-Cases, sowie die Mutations- und Crossoverrate verändert. Dabei wurden Fehler bei der Konvertierung des Formates, in dem die Schachstellungen in der Datenbank gespeichert werden, in das Format der Klasse `CHESSGP_Board` entdeckt und beseitigt.¹¹⁶ Außerdem stellte sich heraus, dass die Rückgabewerte der Operationen allesamt mit „0“ überschrieben worden waren. Die Korrektur dieser Fehler führte zu grundsätzlich „vernünftigerem“ Verhalten.

Weitere Versuche sollten klären, warum nun die Fitnesswerte des besten Individuums einer jeden Generation so sehr schwankten. Eine Veränderung der Mutations- und Crossoverraten führte zu keinem Erfolg. Sowohl bei Abschaltung von Crossover und Mutation, als auch bei Höchstwerten für diese blieben die Schwankungen gleich. Jedoch zeigten sich Erfolge bei Experimenten mit den Fitness-Cases. Daher wurden nun nicht mehr in jeder Generation alle Fitness-Cases neu gewählt, sondern es blieb immer ein Teil der Fitness-Cases erhalten. Das Ergebnis war, dass die Fitnesswerte nun ziemlich stetig (mit seltenen Ausreißern) fielen; das Erreichen des Evolutionszieles in einem Lauf ist aber nicht garantiert. 2 von 4 Testläufen erreichten sehr schnell die Fitness 1, die anderen blieben ziemlich schlecht. Es sei hier noch angemerkt, dass die Erfolge *nur* in einem eingeschränkten Suchraum erzielt werden konnten.

In verschiedenen Versuchen wurde die Größe der Individuen sowie die Anzahl der Operationen und Terminale stark beschränkt, um den Suchraum zu verkleinern und ein gewünschtes Ergebnis wahrscheinlicher zu machen. Diese sehr starken Beschränkungen waren nur für das Testen vorgesehen, allerdings

¹¹⁵ Siehe Abschnitt 75.

¹¹⁶ Die Fehler bei der Konvertierung zeigten sich z. B. dadurch, dass in gewissen Stellungen 7 Könige auf dem Brett waren.

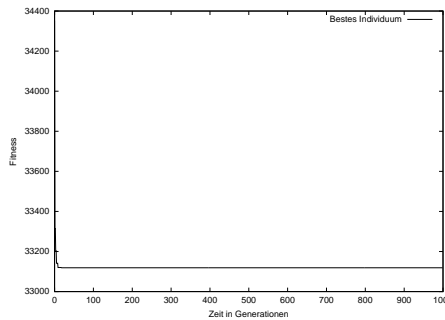


Abbildung 43: Evolutionslauf der Bewertungsfunktion *Clever* mit einer Populationsgröße von 100.

hat sich gezeigt, dass man auch bei der späteren „richtigen“ Evolution nicht um gewisse Beschränkungen herumkam, da die Vielfältigkeit der Operationen den Suchraum dermaßen stark vergrößerte, dass man einige Operationen, die sich in Experimenten für die jeweilige Bewertungsfunktion als ungeeignet erwiesen, wegließ.

7.3.4 Evolution ohne Restart

Erste Versuche einer Evolution von Stufe a wurden mit einer Populationsgröße von 100 Individuen durchgeführt. Aufgrund der geringen Verbesserung des Fitnesswertes des besten Individuums wurde die Populationsgröße auf 1000 verzehnfacht. Der direkte Vergleich zwischen dem Lauf mit der kleineren Population (Abb. 43) und demjenigen mit der größeren Population (Abb. 46) zeigt die Veränderung, die wohl in erster Linie durch die Vergrößerung der Individuenanzahl hervorgerufen wurde.

Auf diesen beiden und allen folgenden Kurven in diesem Abschnitt ist jeweils der Verlauf des Fitnesswertes des besten Individuums der jeweiligen Generation aufgezeichnet. Dabei ist zu beachten, dass die Fitnesswerte in dieser Stufe nicht normiert sind. Die Fitnesswerte bilden die Quadratsumme der Fehlerabweichung über alle Fitnesscases.¹¹⁷ Da sie direkt von den Werten der Bewertungsfunktionen abhängen, fallen die Bereiche der Fitnesswerte je nach Bewertungsfunktion unterschiedlich aus. Eine Ausnahme bilden hierbei die Bewertungen *Angriff* (Abb. 45) und *Verteidigung* (Abb. 51). Bei beiden sind die von den Bewertungsfunktionen vergebenen Werte so hoch, dass aus der Quadratsumme der Fehlerabweichung noch die Wurzel gezogen werden musste, um Werte im gültigen Bereich zu bekommen.

Insgesamt ist zu beobachten, dass bei fast allen Bewertungen ein anfänglich abwärts zum Optimum tendierendes Verhalten des besten Individuums zu beobachten war. Doch nach relativ kurzer Zeit entwickelten sich die Fitnesswerte nur noch schleppend bzw. stagnierten. In dieser Serie war die Evolutionszeit noch

¹¹⁷ Zur genauen Beschreibung der Berechnung des Fitnesswertes siehe Abschnitt 6.2.4.

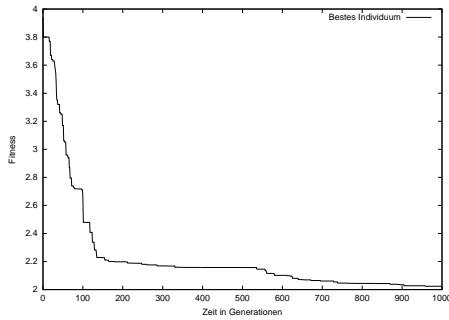


Abbildung 44: Bewertung *Aggressiv* ohne Restart, Populationsgröße 1000

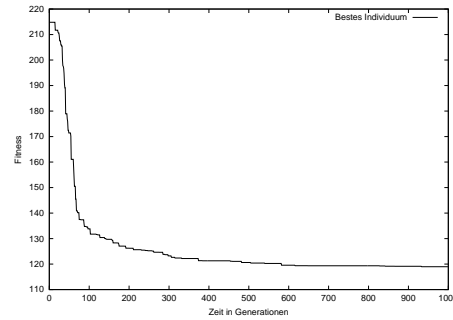


Abbildung 45: Bewertung *Angriff* ohne Restart, Populationsgröße 1000

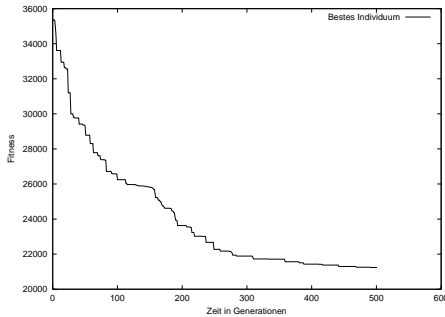


Abbildung 46: Bewertung *Clever* ohne Restart, Populationsgröße 1000

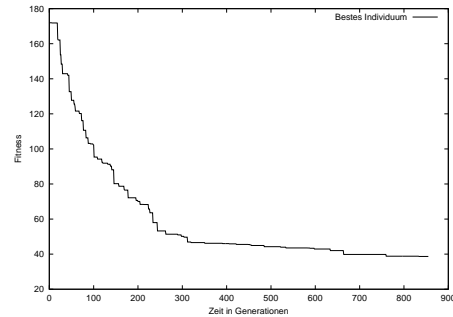


Abbildung 47: Bewertung *Defensiv* ohne Restart, Populationsgröße 1000

auf eine vorher festgelegte maximale Generation festgelegt. Daher der Abbruch bei einigen Fitnesskurven, deren Verlauf noch einen Abwärtstrend zeigt (z. B. in Abb. 51).

Alle Läufe ohne Restart sind in den folgenden Abbildungen 44 bis 51 dargestellt.

7.3.5 Evolution mit Restart

Neue Userparameter (siehe Tabelle 9) sorgen nun für das Einlesen von Individuen und die Steuerung des Restart. Ein weiterer hier nicht angezeigter Parameter erlaubt die Einstellung, ob im Anschluss an die Fitnesswertberechnung aus diesem Fitnesswert noch die Wurzel gezogen werden soll, da dies auch weiterhin für die Bewertungen *Angriff* und *Verteidigung* notwendig ist.

Die Parameter *ToPoolInit* und *IndFile* legen fest, ob ein und welches Individuum zu Beginn geladen wird und wie oft es in den Pool geschrieben wird. Die Auswahl der Poolstellen erfolgt dabei zufällig. Eine ähnliche Funktion hat der Parameter *ToPoolRestart*, nur dass es sich hierbei um die Verteilung von Individuen aus dem bisherigen Lauf in einen neu initialisierten Pool handelt. Die Restart-Generation *RestartGen* legt die Häufigkeit der Überprüfung auf einen

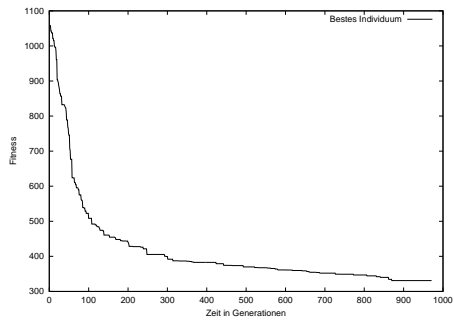


Abbildung 48: Bewertung *Endspiel* ohne Restart, Populationsgröße 1000

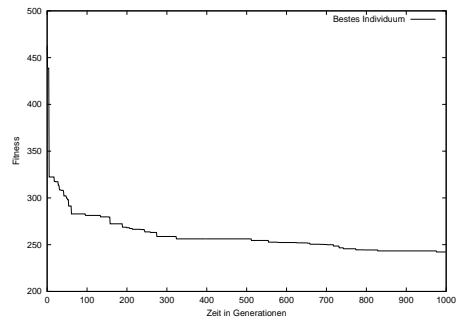


Abbildung 49: Bewertung *Komplex* ohne Restart, Populationsgröße 1000

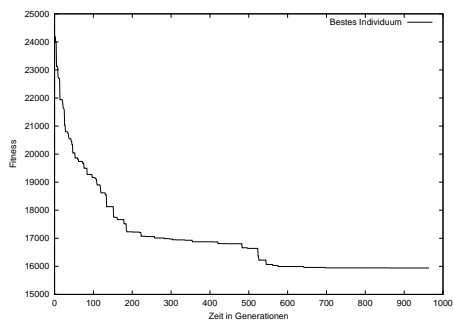


Abbildung 50: Bewertung *Material* ohne Restart, Populationsgröße 1000

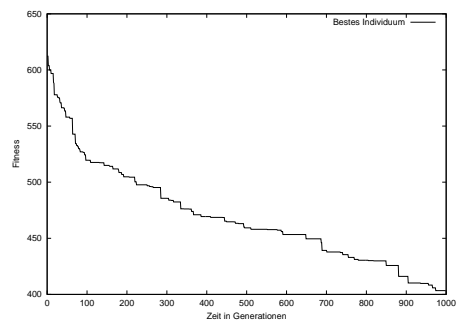


Abbildung 51: Bewertung *Verteidigung* ohne Restart, Populationsgröße 1000

Restart fest und der Parameter *RestartProb* gibt die Steigung der Fitnesskurve an, die nicht unterschritten werden darf. Ist dies der Fall, wird ein Restart durchgeführt.

Die Auswirkungen dieses Vorgehens sind in den Abbildungen 52 bis 59 zu sehen. Wie in der Serie des letzten Abschnittes wurden diese Läufe „von vorne“ gestartet (also ohne ein Individuum zu Beginn zu laden), lernten mit 40 000 Schachbrettern mit einer Poolgröße von 1 000. Geändert wurde lediglich die Anzahl der Fitnesscases auf 500. Die Parameter des Restarts sind so eingestellt, dass in jeder 50. Generation die Steigung der letzten 50 Generationen überprüft und mit dem „Minimalwert“ -1 verglichen wird.¹¹⁸ Bei einem Restart werden in diesem Fall (Poolgröße 1 000) 100 Individuen des neuen Pools durch alte gesicherte Individuen ersetzt.

Die Hauptschleife ist nicht mehr auf eine maximale Generationenzahl beschränkt, sondern bricht bei Erreichen des optimalen Fitnesswertes ab. Einige Läufe sind durch externe Einflüsse abgebrochen worden. Da eine erste Überprüfung auf einen Restart frühestens in der 50. Generation stattfand, sind dort die Auswirkungen des Restarts nicht unbedingt sichtbar. Bei gutem Fortschritt in den ersten 50 Generationen ist dieser auch eventuell gar nicht durchgeführt worden. Eindeutige Auswirkungen des Restarts sind in den Abbildungen 53, 54 und 59 zu sehen. Das oszillierende Verhalten entsteht durch das Fehlen eines Individuums der aktuellen Generation im neuen Pool nach dem Restart. Hier wurde immer auf ein Individuum der ersten Generationen zurückgegriffen, was sich auch in den meisten Fällen als neues bestes Individuum durchsetzte. Dieses war aber in der Regel schlechter als das beste Individuum der letzten Generation. In den folgenden Läufen wurde dieses Verhalten dadurch behoben, dass bei einem Restart *immer* ein Individuum der vorherigen Generation bewahrt wird. Sind die Ausschläge in den Restartgenerationen nicht so deutlich oder gar nicht vorhanden, wie in Abb. 57, so liegt das daran, dass unter bestimmten Bedingungen¹¹⁹ automatisch ein Individuum aus den näherliegenden letzten Generationen gewählt wird. Eine wahrscheinlich auf den Restart zurückzuführende Verbesserung kann in Abb. 55 beobachtet werden.

Es folgen die Abbildungen der Entwicklung des Fitnesswertes des besten Individuums jeder Bewertungsfunktion über der Zeit:

7.3.6 Eingeschränkte Operationenmenge

7.3.6.1 Beschreibung der eingeschränkten Operationenmengen

Um den Lösungsraum des Problems einzuschränken und damit die Suche möglicherweise zu beschleunigen, wurde für eine jede Bewertungsfunktion in der Stufe A eine Untermenge von Operationen spezifiziert, die anstelle der gesamten Operationenmenge benutzt wurden. Bei der Festlegung dieser Teilmengen wurde jeweils darauf geachtet, für die dazugehörigen Bewertungsfunktionen möglichst

¹¹⁸ Ein Wert größer als -1 würde bedeuten, dass die Fitnesskurve „zu flach“ verlaufen ist.

Gesucht ist eine hohe *negative* Steigung, da das Optimum bei 0 liegt.

¹¹⁹ Genaueres dazu in 6.1.3.

Bezeichnung	Wert	Erklärung
ToPoolInit	0	Anzahl der Stellen an denen ein eingelesenes Individuum in den Pool gesetzt wird
IndFile	indA1	Datei, die ein einzulesendes Individuum enthält
ToPoolRestart	1	Anzahl der Stellen an denen ein beim Restart gesichertes Individuum in den Pool gesetzt wird
RestartGen	50	Anzahl der Generationen nach denen geprüft wird, ob ein Restart notwendig ist
RestartProb	-1	Steigung, ab der ein Restart durchgeführt wird

Tabelle 9: Erweiterte User-Parameter

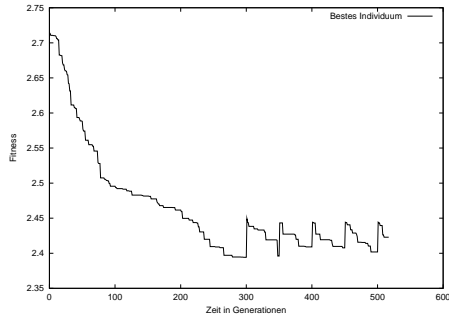


Abbildung 52: Bewertung *Aggressiv* mit Restart in jeder 50. Gen., Populationsgröße 1000

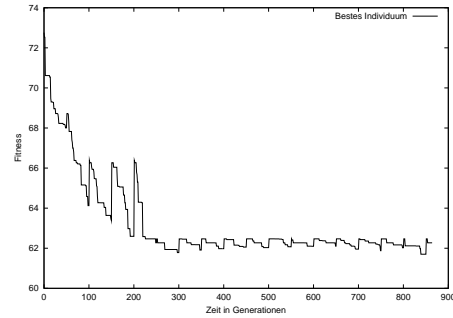


Abbildung 53: Bewertung *Angriff* mit Restart in jeder 50. Gen., Populationsgröße 1000

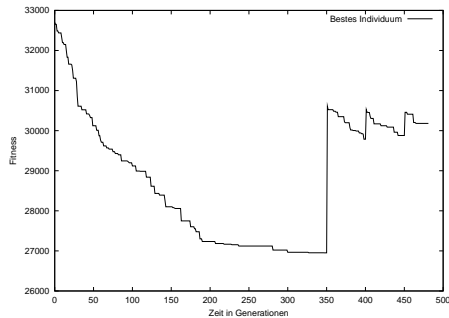


Abbildung 54: Bewertung *Clever* mit Restart in jeder 50. Gen., Populationsgröße 1000

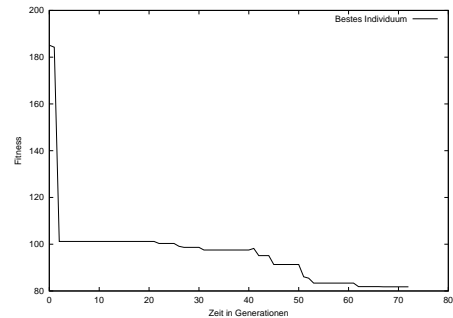


Abbildung 55: Bewertung *Defensiv* mit Restart in jeder 50. Gen., Populationsgröße 1000

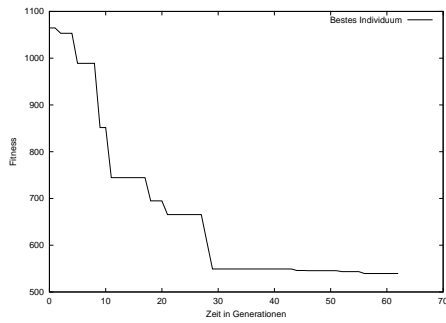


Abbildung 56: Bewertung *Endspiel* mit Restart in jeder 50. Gen., Populationsgröße 1000

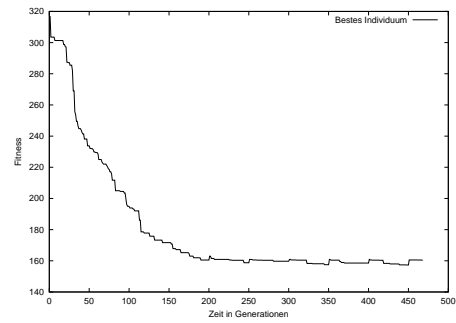


Abbildung 57: Bewertung *Komplex* mit Restart in jeder 50. Gen., Populationsgröße 1000

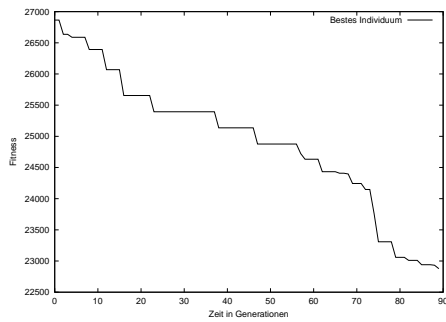


Abbildung 58: Bewertung *Material* mit Restart in jeder 50. Gen., Populationsgröße 1000

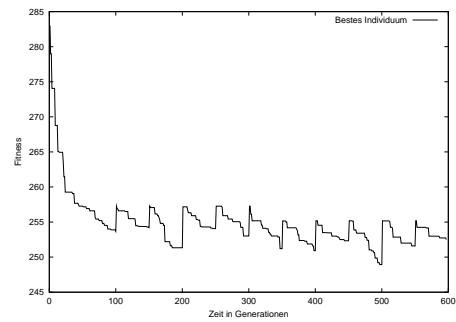


Abbildung 59: Bewertung *Verteidigung* mit Restart in jeder 50. Gen., Populationsgröße 1000

„hilfreiche“ Konstrukte in der aktiven Sprachmenge zu verwenden.

Für alle Bewertungsfunktionen sollen die vier arithmetischen Standardoperationen sowohl in der Drei-Adress- als auch in der Zwei-Adress-Version zur Verfügung gestellt werden. Außerdem werden die Operationen *absolute* und *ifThen* zugelassen, um einerseits die im Vorzeichen liegende Farbe ausblenden und andererseits auch nach Farben unterscheiden zu können.

Diese für alle Bewertungsfunktionen gemeinsamen Operationen lauten: *add*, *addEqual*, *sub*, *subEqual*, *mul*, *mulEqual*, *div*, *divEqual*, *absolute*, *ifThen*.

Im Folgenden werden die Operationen aufgelistet und beschrieben, die für die einzelnen Bewertungsfunktionen spezifisch sind und zu den gemeinsamen Funktionen noch hinzugefügt werden:

1. Material

Operationen: *boardLoop*, *whichPiece*

Terminale: *pieceValue*

Die Bewertungsfunktion „Material“ ist die einfachste aller Bewertungsfunktionen. Es können sogar mit einer Untermenge dieser Sprachausdrücke die von der Fitnessfunktion geforderten Ergebnisse exakt berechnet werden. Dafür muss zu Beginn ein Register auf Null gesetzt werden, auf das dann im Verlauf eines *Boardloops* die Figurenwerte aller Felder summiert werden.

2. Clever

Operationen: *boardLoop*, *whichPiece*, *setRow*, *setLine*

Terminale: *pieceValue*, *legalMovesNumber*

Die „Clever“-Bewertung erweitert die obige Material-Bewertung um einige kleinere Terme. Insbesondere erhalten Figuren individuelle, in einer eigenen Matrix vermerkte Boni für ihre Standplätze. Deshalb wird es hier mit *setRow* und *setLine* zusätzlich ermöglicht, das aktuelle betrachtete Feld explizit zu setzen. Insgesamt ist hier eine Fitness von 0 aber nicht zu erreichen, weil beispielsweise auch die Bewertung „Endspiel“ in den Zielwert einfließt.

3. Angriff

Operationen: *boardLoop*, *rowLoop*, *setRow*, *whichPiece*, *swap*

Terminale: *pieceValue*, *attackedByValue*, *attackedByNumber*

Diese Bewertung ist um einiges komplizierter als die vorangegangenen. Hier werden zusätzlich die angriffsrelevanten Terminale zugelassen und das gezielte Setzen der aktuell betrachteten Reihe. Letzteres geschieht unter der Annahme, dass ein Angriff eher auf der gegnerischen Bretthälfte, also etwa jenseits der vierten Reihe (von Weiß aus gesehen) stattfinden wird.

4. Verteidigung

Operationen: *boardLoop*, *rowLoop*, *setRow*, *whichPiece*, *swap*

Terminale: *pieceValue*, *attackedByValue*, *attackedByNumber*

Die Bewertungsfunktion „Verteidigung“ wurde als Gegenstück zum „Angriff“ entworfen, und folglich kommen dieselben Sprachbausteine zum Einsatz.

5. Aggressiv

Operationen: *boardLoop*, *rowLoop*, *setRow*, *whichPiece*

Terminale: *pieceValue*, *attackedByValue*, *attackedByNumber*

Betrachteten die letzten vier Bewertungen das Schachbrett jeweils aus der Perspektive des sich am Zuge befindlichen Spielers, so wird von dieser Bewertung an das Brett unabhängig von der Seite am Zug bewertet. Eine Stellung soll „aggressiv“ sein, wenn eine hohe Zahl von Steinen einer Schlagbedrohung durch gegnerische Steine ausgesetzt ist. Dem wird wiederum durch das Hinzufügen der Terminale *attackedByValue* und vor allem *attackedByNumber* Rechnung getragen.

6. Defensiv

Operationen: *boardLoop*, *rowLoop*, *setRow*, *whichPiece*

Terminale: *pieceValue*, *attackedByValue*, *attackedByNumber*

Die Bewertung „Defensiv“ versteht sich als Analogon zur Aggressivität. Es soll festgestellt werden, zu welchem Grad sich die Steine einer Partei gegenseitig *decken*, also eine schlagende Figur des Gegners unmittelbar zurückschlagen könnten. Dazu werden auch hier *attackedByValue* und *attackedByNumber* nutzbar gemacht. Man bedenke hierbei, dass diese Terminale, wenn mit einem „eigenen“ Feld im Akku ausgelesen, sich nicht auf die Angreifer, sondern auf die Deckung beziehen.

7. Komplex

Operationen: *boardLoop*, *whichPiece*

Terminale: *pieceValue*, *legalMovesNumber*, *attackedByNumber*

Als „komplex“ sollen Stellungen erkannt werden, in denen ein aufzuspannender Spielbaum einen hohen Verzweigungsgrad hätte und auch ein skeltierter Spielbaum, der nur aus „wichtigen“ Zügen, wie Schlagfolgen, bestünde, sich noch hinreichend auffächerte. Von daher ist hier insbesondere die Aufnahme von *legalMovesNumber* naheliegend.

8. Endspiel

Operationen: *boardLoop*, *whichPiece*

Terminale: *pieceValue*

Die Endspielbewertung soll feststellen, ob es sich bei der aktuellen Brettstellung um ein Endspiel handelt. Die vorgegebene Fitnessfunktion besteht dazu aus einer linearen Abbildung der Anzahl aller Steine auf dem Brett und kann durch wenige elementare Operationen exakt realisiert werden. Neben der einfachen Materialbewertung ist dies die einzige Bewertung, bei der davon auszugehen ist, dass ein Fitnesswert von 0 in der Praxis wirklich erreicht werden kann.

7.3.6.2 Erfolgreich evolvierte Materialbewertung

Das in der Evolution der Stufe für die Bewertung „Material“ evolvierte Individuum ist optimal. Als einziges Individuum aller drei Stufen und 16 Bewertungen konnte es einen Trainings- und Testfehler von Null erreichen. Der um Introns, welche die Ausführung nicht beeinflussen, bereinigte Quellcode ist im Listing 7 abgebildet.

Listing 7 Quellcode des optimal evolvierten Individuums der Materialbewertung in Stufe a

```

BeginInd
Main Linear -----
{ R 0 = R 3 subequal R0 }
{ R 1 = R 3 boardLoop _Const
      Line      Row      Piece  Value
        7        2        2    162.167
}
{ R 2 = R 1 whichPiece }
{ R 3 = R 2 addequal pieceValue }
{ R 3 = R 2 addequal pieceValue }
{ R 1 = R 1 add pieceValue }
{ R 3 = R 2 addequal pieceValue }
{ R 3 = R 2 addequal pieceValue }
{ R 3 = R 2 addequal pieceValue }
{ R 3 = R 2 addequal pieceValue }
{ R 2 = R 1 div R2 }
{ R 3 = R 2 addequal pieceValue }
{ R 1 = R 3 boardLoop _Const
      Line      Row      Piece  Value
        7        2        2    162.167
}
{ R 3 = R 2 addequal pieceValue }
{ R 3 = R 2 addequal pieceValue }
{ R 2 = R 1 div R2 }
{ R 3 = R 2 addequal pieceValue }
{ R 0 = R 2 boardLoop R1 }

EndInd -----

```

Um Nullfehler zu erreichen, ist es lediglich erforderlich, alle Felder des Schachbretts jeweils genau einmal zu betrachten, und die Werte der sich darauf befindlichen Schachsteine aufzusummieren. Dies ist erfolgsbringend, weil den Werten der schwarzen Figuren ein negatives Vorzeichen vorangestellt ist. Allerdings gilt es, noch eine weitere Klippe zu umschiffen: Die in der Datenbank abgelegten Werte für die Materialsituation sind in $\frac{1}{10}$ -Bauerneinheiten gemessen, sodass noch eine Multiplikation des beim Schleifendurchlauf erhaltenen Wertes mit 10 erforderlich

ist.

Die von CHESSGP gefundene Lösung sieht folgendermaßen aus:

1. Von der Ausführungsumgebung werden alle Register mit dem Wert 1 initialisiert. Vom Akkumulator (Inhalt: 1) wird der Wert des Registers 0 (ebenfalls 1) subtrahiert, wonach der Akkumulator mit dem Wert 0 belegt ist.
2. Danach wird in einer Schleife das Schachbrett durchlaufen. Diese Schleife müsste eigentlich 162 Zeilen umfassen, wie im Value-Feld der Konstanten angegeben, aber um ineinander verschachtelte Schleife zu vermeiden, wird, wenn der Beginn einer weiteren Schleife innerhalb eines Boardloops liegt, der Umkehrpunkt des Boardloops auf die letzte Zeile vor der zweiten Schleife vorgezogen. Man betrachte die in diesem Boardloop ausgeführten Befehle:
 - Mit dem Befehl „addequal“ wird der Wert des auf einem Feld stehenden Materials siebenmal auf den Akkumulatorwert addiert.
 - Die Befehle „whichpiece“, „add“ und „div“ manipulieren unbeteiligte Register und können als Introns aufgefasst werden.
3. In einer zweiten Schleife über das ganze Schachbrett wird der Materialwert noch dreimal aufaddiert.

Insgesamt befindet sich jetzt, wie gefordert, der zehnfache Wert der Materialdifferenz im Value-Feld des Akkumulators.

Die Darstellung dieser Lösung kommt ohne die in jedem Register vorhandenen Feldern „Line“, „Row“ und „Piece“ aus. Vermutlich, weil es bei den anderen Zielfunktionen keine solche „einfache“ Darstellung gibt, konnte für keine von ihnen eine ebenfalls optimale Lösung gefunden werden.

7.3.6.3 Weitere Ergebnisse

In den Läufen mit eingeschränkter Operationenmenge (Abb. 60 bis 66) wurden einige Parameter verändert:

Die Datenbank wurde erweitert und um Endspielsituationen der oft nicht zu Ende gespielten Partien ergänzt. Die neuen Läufe lernen auf einer Auswahl von weiterhin 500 Fitnesscases aus nunmehr 65 304 Schachbrettern. Die Populationsgröße ist 500 und der Wertebereich der Konstanten wurde auf $[-15; 15]$ beschränkt. Wird ein Restart durchgeführt, so wird die Hälfte des neu initialisierten Pools (an 250 zufällig ausgewählten Stellen) mit alten Individuen gefüllt. Die Materialbewertung ist in den folgenden Abbildungen nicht mehr zu sehen, da hier, wie im vorigen Abschnitt beschrieben, ein Evolutionserfolg erzielt wurde.

Zu beachten ist bei den Graphen zum Einen die oft kurze Laufzeit, welche durch häufige externe Rechnerstörungen und Abbrüche aufgrund von Datenbankproblemen zurückzuführen ist. Die hier abgebildeten Läufe zeigen störungsfreie Sequenzen mit einer korrekten Datenbankbasis. Zum Anderen ist zu beachten, dass es in einigen Verläufen nach einer Konvergenz der Kurve zum Optimum

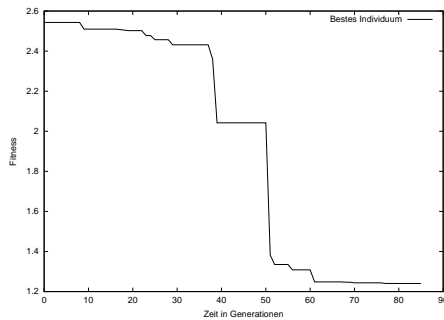


Abbildung 60: Bewertung *Aggressiv* mit eingeschränkter Operationenmenge

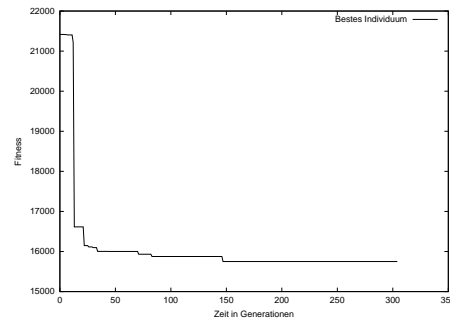


Abbildung 61: Bewertung *Angriff* mit eingeschränkter Operationenmenge

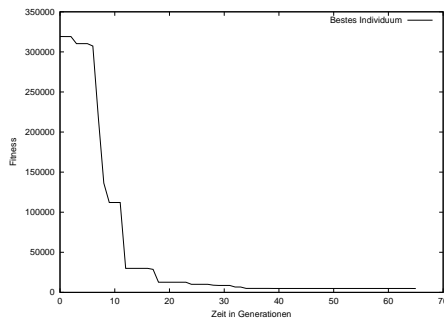


Abbildung 62: Bewertung *Clever* mit eingeschränkter Operationenmenge

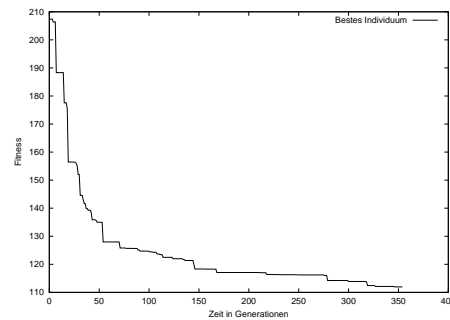


Abbildung 63: Bewertung *Defensiv* mit eingeschränkter Operationenmenge

hin aussieht. Dies ist aufgrund der Skalierung aber nur scheinbar der Fall. Oft sind die Fitnesswerte zu Beginn des Laufes so hoch, dass die Auflösung der Graphik dadurch relativ grob wird.

7.3.7 Datenbanktest der evolvierten Individuen

In diesem Abschnitt sind in den Abbildungen 67 bis 73 die Fehlerwerte der besten Individuen der Stufe a auf der gesamten Datenbank aufgezeichnet. Die X-Achse repräsentiert die Nummern sämtlicher Schachbretter in der Datenbank und die Y-Achse steht für die Abweichung der Rückgabe des Individuums vom Wert in der Datenbank. Dazu gibt es wieder zu allen Bewertungen einen Graphen, mit Ausnahme der Materialbewertung, da dort ein Individuum evolviert wurde, das auf allen Schachbrettern keinen Fehler mehr produzierte und den korrekten Wert berechnete. Unter jeder Abbildung wird zu jedem dort dargestellten Individuum angegeben, welches sein *Fitnesswert* während der Evolution (auf den Fitnesscases) war und welchen *durchschnittlichen Gesamtfehler* es auf der *gesamten* Datenbank verursacht.

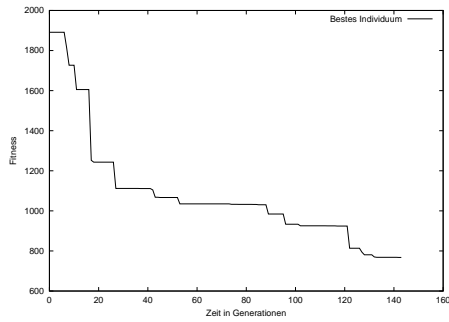


Abbildung 64: Bewertung *Endspiel* mit eingeschränkter Operationenmenge

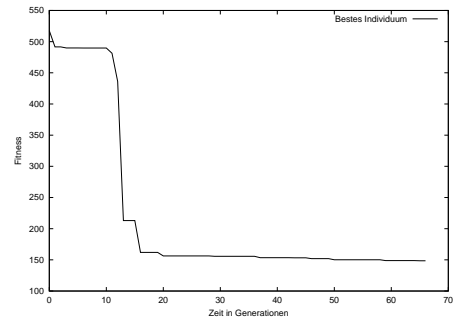


Abbildung 65: Bewertung *Komplex* mit eingeschränkter Operationenmenge

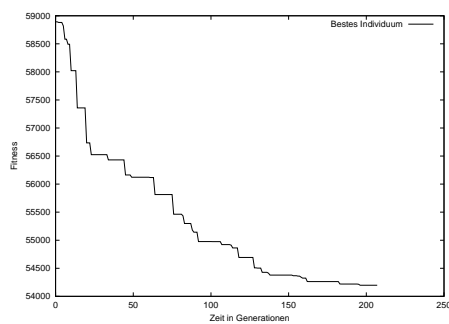


Abbildung 66: Bewertung *Verteidigung* mit eingeschränkter Operationenmenge

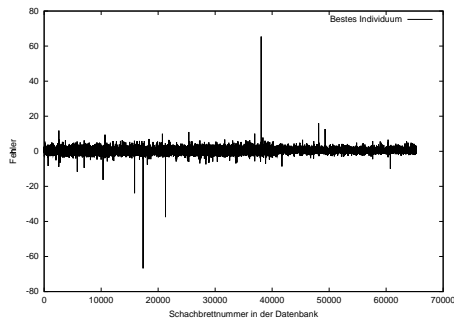


Abbildung 67: Datenbankfehler bei *Aggressiv*. Fitness: 1,15326; Gesamtfehler: 0,883851

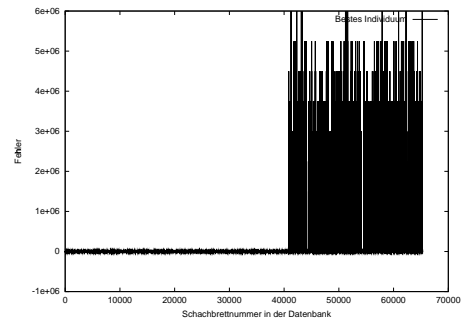


Abbildung 68: Datenbankfehler bei *Angriff*. Fitness: 15749,6; Gesamtfehler: 61108,7

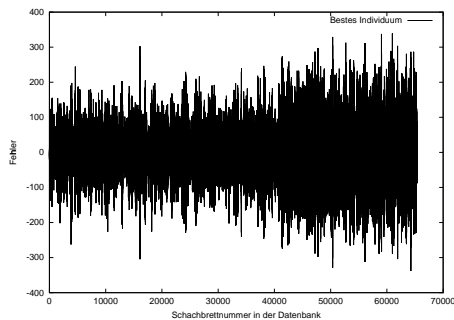


Abbildung 69: Datenbankfehler bei *Clever*. Fitness: 4729,12; Gesamtfehler: 50,868

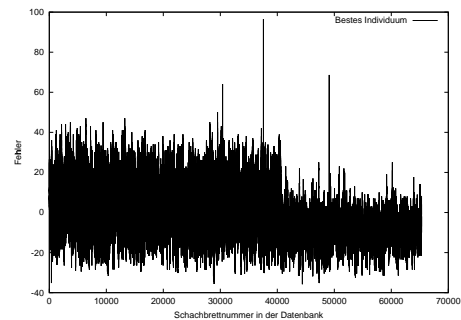


Abbildung 70: Datenbankfehler bei *Defensiv*. Fitness: 105,178; Gesamtfehler: 8,266

Bemerkenswert dabei ist, dass einige Individuen auf der gesamten Datenbank einen durchschnittlich gleichen Fehler produzieren (siehe Abbildungen [67](#), [69](#), [70](#), [72](#)), während bei anderen Bewertungen deutlich die beiden Datenbankbereiche unterhalb und oberhalb der Marke von 40 000 Brettern zu erkennen sind. Im Bereich ab Brett 40 000 finden sich die neu hinzugefügten Endspiele.¹²⁰ In einigen Bewertungen (z.B. *Angriff*) sind also die bei normalen Spielsituationen erfolgreichen Vorgehensweisen nicht mehr bei Endspielen anzuwenden. Insgesamt kann festgehalten werden, dass es gelingen kann, aufgrund von Datenbankbeispielen GP-Individuen zu erzeugen, die auf der gesamten Datenbank ein gleichmässiges Verhalten zeigen (auch wenn dieses Verhalten hier noch nicht dem Optimum entspricht).

¹²⁰ Die Ergänzung der Endspiele wurde mit dem Programm Crafty vorgenommen. Eine Beschreibung dazu findet sich in [4.10](#). Es ist ferner zu beachten, dass es sich bei den Endspielen in der Datenbank um von Crafty erzeugte Endspiele handelt. Diese können sich von real gespielten unterscheiden.

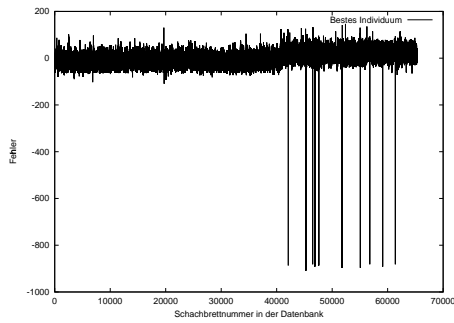


Abbildung 71: Datenbankfehler bei *Endspiel*. Fitness: 716,121; Gesamtfehler: 22,583

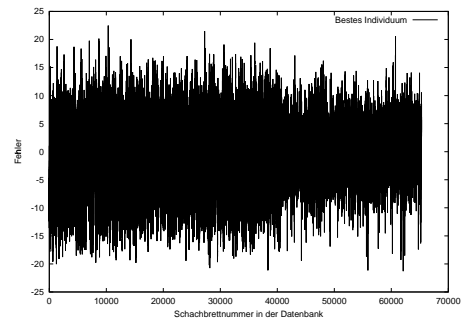


Abbildung 72: Datenbankfehler bei *Komplex*. Fitness: 25,2514 Gesamtfehler: 3,9244

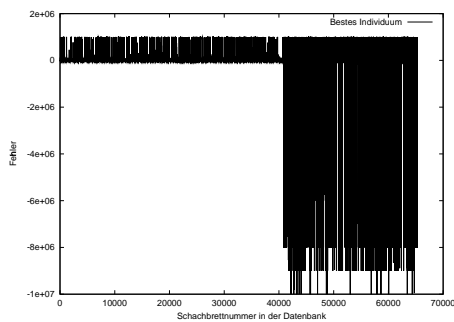


Abbildung 73: Datenbankfehler bei *Verteidigung*. Fitness: 43212,5 Gesamtfehler: 213685

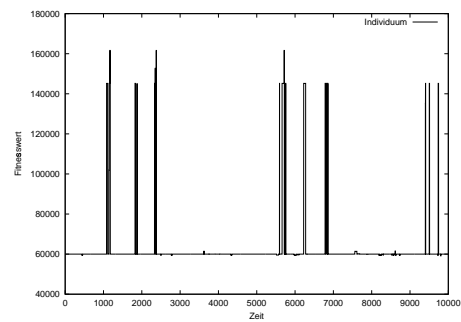


Abbildung 74: Random Walk der Bewertung *Angriff*. Typ: Mutation, Anzahl Mutationen pro Schritt: 1

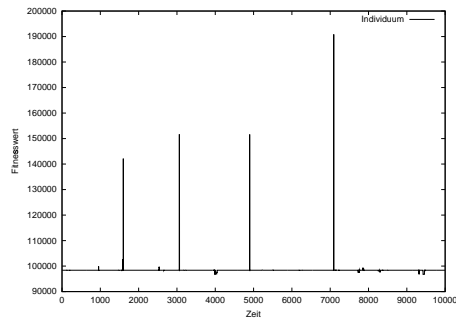


Abbildung 75: Random Walk der Bewertung *Angriff*. Typ: Mutation, Anzahl Mutationen pro Schritt: 1

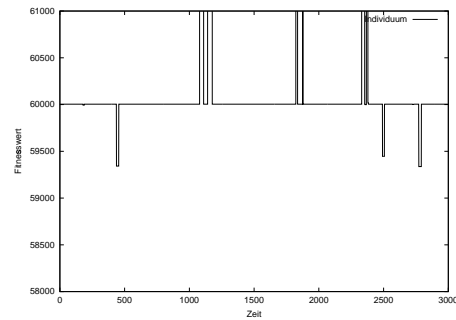


Abbildung 76: Ausschnitt aus Abbildung 74, Gen. 0 bis 3000

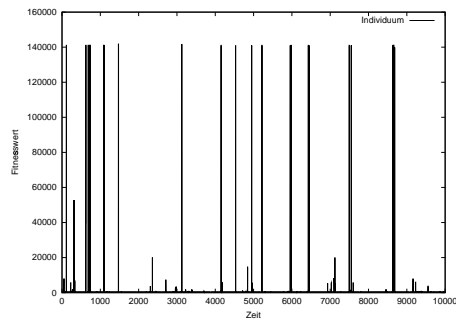


Abbildung 77: Random Walk der Bewertung *Angriff*. Typ: Mutation, Anzahl Mutationen pro Schritt: 2

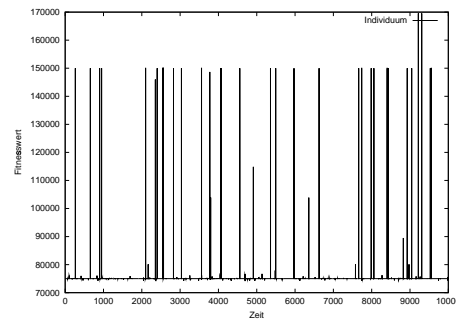


Abbildung 78: Random Walk der Bewertung *Angriff*. Typ: Mutation, Anzahl Mutationen pro Schritt: 4

7.3.8 Random Walks

Anhand der Bewertungsfunktion *Angriff*, die sich als schwierig zu lernen herausstellte, soll gezeigt werden, wie Random-Walk-Analysen genutzt werden können und welche Schlussfolgerungen daraus für die Evolution in Stufe b gezogen wurden.¹²¹

Die Abbildungen 74 und 75 zeigen zwei verschiedene Random Walks mit gleichen Parametern. Es wurde genau einmal pro Schritt ein Knoten mutiert. Jeweils ein Random Walk erstreckte sich über 10 000 Zeitschritte, was 10 000 durchgeführten Mutationen entspricht. Ein Ausschnitt über die ersten 3000 Generationen aus Abb. 74 ist in Abb. 76 zu sehen. Auffällig ist, dass einen konstanten Fitnesswert zu geben scheint, der nur für sehr kurze Zeit hohe Abweichung aufweist. Diese Abweichungen sind nie von langer Dauer. Trotz andauernder Veränderung bleibt die Fitnessbewertung des Individuums gleich.

¹²¹ Die Funktionsweise des Random Walk wird im Kapitel über die Analysetools (7.1) erläutert.

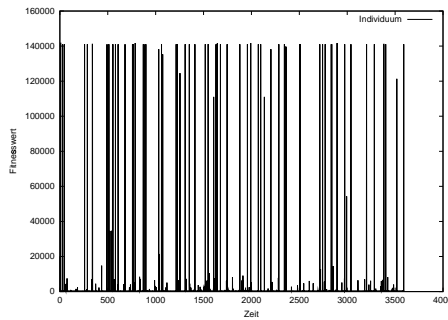


Abbildung 79: Random Walk der Bewertung *Angriff*. Typ: Mutation, Anzahl Mutationen pro Schritt: 10

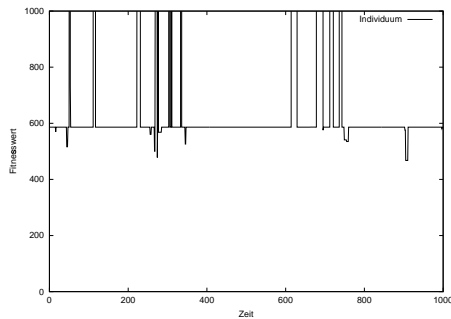


Abbildung 80: Ausschnitt aus Abbildung 77, Gen. 0 bis 1000

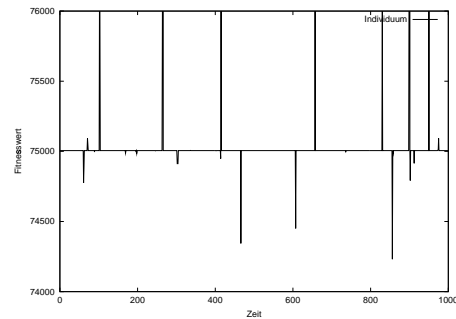


Abbildung 81: Ausschnitt aus Abbildung 78, Gen. 0 bis 1000

Die SYSGP-Parameter des Random Walks entsprechen denen der Evolutionsläufe. Um festzustellen, ob sich größere Veränderungen erzwingen lassen, wurde der *Mutationscounter*, also die Anzahl der pro Mutation veränderten Knoten, erhöht. In den Abbildungen 77 bis 79 sind Random Walks mit 2, 4 und 10 veränderten Knoten pro Mutation (also in jedem Schritt) zu sehen. An der Zunahme der Peaks lässt sich die zunehmende Veränderung bereits erkennen. Die in den Abbildungen 80 bis 82 dargestellten Ausschnitte aus diesen Läufen machen deutlich, dass die Veränderungen nicht mehr nur sprunghaft sind und sich auch in Richtung des Optimums bewegen können.

Eine Schlussfolgerung aus diesen Ergebnissen ist die Erhöhung der Mutationsrate oder der Anzahl der bei jeder Mutation zu verändernden Knoten. Offensichtlich finden Veränderungen in größerem Maße nur bei verstärkter Mutation statt.

Da in diesen Untersuchungen der Einfluss der Mutation separiert wurde, sollte auch der Crossover-Operator getrennt untersucht werden. Wenn man sich dabei auf zwei Individuen und deren durch die Rekombination entstehende Nachfahren beschränkt, ist jedoch auch eine Mutation erforderlich, um überhaupt Veränderung zu erzeugen. Diese Mutation bestand aber nur aus einer Veränderung in einem Knoten der neuen Individuen nach der Rekombination. Dass der

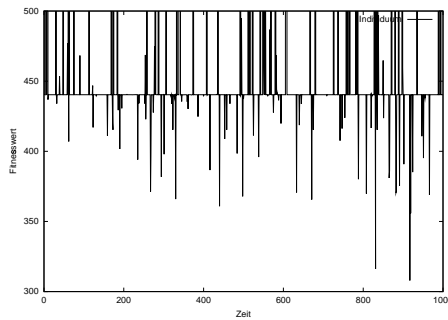


Abbildung 82: Ausschnitt aus Abbildung 79, Gen. 0 bis 1000

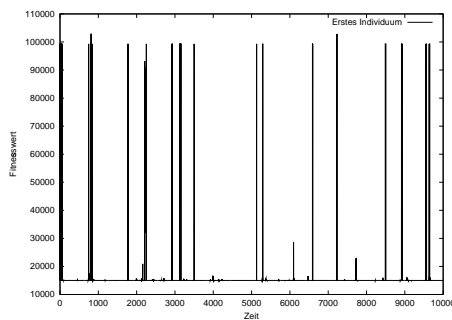


Abbildung 83: Random Walk der Bewertung *Angriff*. Typ: Crossover, Anzahl Mutationen pro Schritt: 1

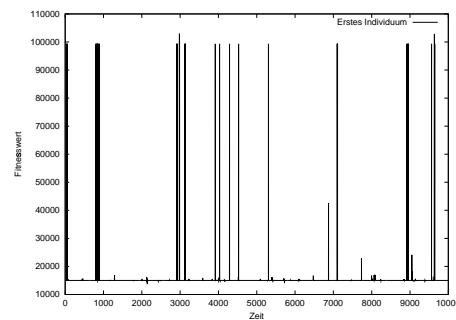


Abbildung 84: Random Walk der Bewertung *Angriff*. Typ: Crossover, Anzahl Mutationen pro Schritt: 1

Einfluss der einfachen Mutation für sich genommen relativ gering ist, wurde aus den vorhergehenden Beispielen ersichtlich. Dieser Analyselauf erstreckte sich ebenfalls über 10 000 Schritte.

Die Abbildungen 83 und 84 zeigen den Fitnessverlauf der beiden Individuen eines solchen Random Walk mit Crossover. In den Abbildungen 85 und 86 wird jeweils ein in der Fitness-Skala eingeschränkter Ausschnitt über den gesamten Verlauf der beiden Läufe gezeigt.

An diesen wenigen Versuchen mit einem Crossover kann bereits abgelesen werden, dass die Variabilität der Fitnesswerte der Individuen im Vergleich zu den Läufen mit nur einer Mutation an einem Knoten bereits lediglich durch die Hinzunahme des Crossovers wesentlich größer wird. Natürlich können aufgrund einiger weniger Versuche noch keine endgültigen Schlussfolgerungen über den tatsächlichen Einfluss oder gar eine qualitative Verbesserung durch den Crossover-Operator gezogen werden. Dass jedoch ein anscheinend nicht unerheblicher Einfluss vorliegt, ist schon hier ersichtlich. Eine Schlussfolgerung aus den Random Walks für die kommenden Läufe in Stufe b war es, den Einfluss der Mutation durch Erhöhung der Mutationsrate und der Anzahl der pro Mutation zu verändernden Knoten zu erhöhen. Es hatte sich gezeigt, dass die Mutation stärker betont werden muss, um gegenüber dem Crossover zur Geltung zu

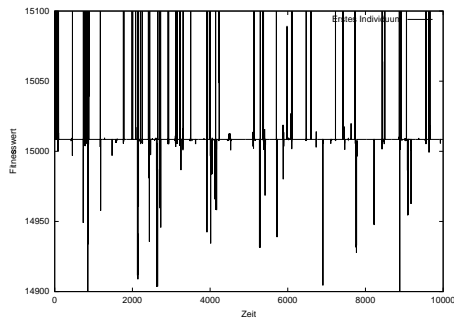


Abbildung 85: Ausschnitt aus Abbildung 83; alle Generationen, aber Beschränkung auf der Fitness-Skala

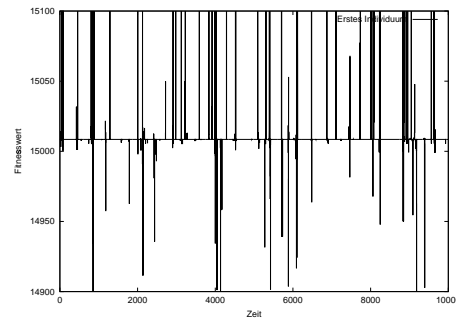


Abbildung 86: Ausschnitt aus Abbildung 84; alle Generationen, aber Beschränkung auf der Fitness-Skala

kommen und Veränderungen hervorzurufen.¹²²

7.4 Analyse der Stufe b

Im Folgenden werden Experimente und Analysen der Stufe b beschrieben. Zunächst wird ein kurzer Überblick über die Parameter der Stufe b gegeben (Abschnitt 7.4.1). Da die Individuen der Stufe b nicht mehr nur noch einfache Werte zur Bewertung eines Schachbretters, sondern Zugvorschläge zurückliefern sollten, befasst sich die Analyse der Ergebnisse in Abschnitt 7.4.3 mit diesen Zuglisten. Zuvor (Abschnitt 7.4.2) wird das Verhalten der Fitnesskurven während der Evolution beschrieben. Abschliessend wird noch ein Versuch beschrieben, das Verhalten der Stufe b durch die Zulassung beliebig langer Zuglisten zu verbessern (Abschnitt 7.4.4) und eine Schlussbemerkung zur Evolution beider Stufen – a und b – gegeben.

7.4.1 Parameter der Stufe b

Die verwendeten Parameter entsprechen zunächst einmal denen der Stufe a (siehe auch Abschnitt 7.3.2). Abweichungen werden an den entsprechenden Stellen angegeben.

Die vollständigen Mengen der Operationen und Teminale beinhalten¹²³:

- *Operationen*: b.intersection b.unionize b.difference b.cardinality
b.sortByValue b.number b.numberByBorder b.subset b.subsetByBorder
b.incrementIfEqual b.set2Mask b.increaseAllValues b.setAllValues
b.setAll2WildCard b.setSourceRow b.setSourceRow2WildCard
b.setSourceLine b.setSourceLine2WildCard b.setDestinationRow

¹²² Dabei wurde die Annahme gemacht, dass sich die in Stufe a gewonnenen Erkenntnisse auf Stufe b übertragen lassen. Eine Validierung dieser These wurde jedoch nicht vorgenommen.

¹²³ Die Operationen werden in 6.1.5.5 näher erläutert.

Parameter Name	Parameter Value
Population Size	100
Crossoverrate	80
Mutationrate	50
MainMutCnt	10
MinConstant	−15
MaxConstant	15
ToPoolRestart	10
RestartProb	−1
RestartGen	30
probFitnessCases	30
chessboardNumber	65304
fitnessCases	100

Tabelle 10: Neue Parameter

b_setDestinationRow2WildCard b_setDestinationLine
 b_setDestinationLine2WildCard b_setSearchMaskValue b_setDirection
 b_setDirection2WildCard b_setPiece b_setPiece2WildCard b_setColour
 b_setColour2WildCard b_setHitPiece b_setHitPiece2WildCard
 b_copyPiece2HitPiece b_copyHitPiece2Piece b_callIndividualA b_loadAll
 b_loadAllIndirect b_storeAll b_storeAllIndirect b_loadValue
 b_loadValueIndirect b_storeValue b_storeValueIndirect
 b_loadSearchmaskIndirect b_loadSearchmask b_storeSearchMask
 b_storeSearchmaskIndirect b_loadMoveList b_loadMoveListIndirect
 b_storeMoveList b_storeMoveListIndirect b_loadMoveValue
 b_loadMaxMoveValue b_loadMinMoveValue b_loadMedianMoveValue
 b_storeMoveValue b_add b_sub b_mul b_div b_swap b_getAllMoves
 b_ifThen

- *Terminale*: Keine Terminale in Stufe b

7.4.2 Typisches Verhalten der Evolutionsläufe

Die erste Serie der Läufe in Stufe b ist in den Abbildungen 87 bis 94 dargestellt. Die wesentlichen neuen Parametereinstellungen sind in Tabelle 10 dargestellt.

Die geringe Populationsgröße resultiert zum Einen auf der Vermutung, durch den Restart-Mechanismus über eine genügend hohe Diversität zu verfügen, und zum Anderen auf vorausgegangenen Tests, in denen sich eine hohe Individuenzahl nur auf die Rechenzeit, nicht aber auf den Verlauf der Fitnesskurve auswirkte. Die Parameter *Mutationrate* und *MainMutCnt* (Anzahl der veränderten Knoten pro Mutation) wurden erhöht (Mutationrate von 30 auf 50, MainMutCnt von 1 auf 10), um die Erfahrungen der Random Walks aus Stufe a mit einzubeziehen (siehe auch 7.3.8). Es zeigte sich jedoch, dass sich weder hiermit, noch mit einer später erfolgten Vergrößerung der Individuen der Verlauf Fitnesskurven

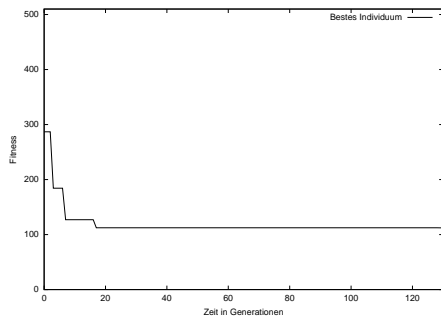


Abbildung 87: Bewertung *Angriffszüge* mit Restart in jeder 30. Gen., Populationsgröße 100

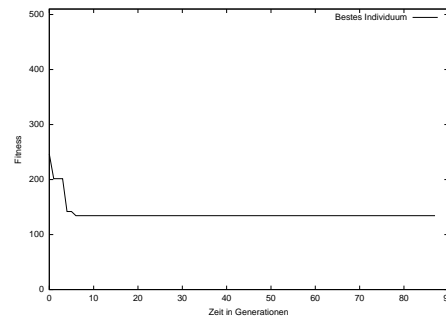


Abbildung 88: Bewertung *Beste Züge* mit Restart in jeder 30. Gen., Populationsgröße 100

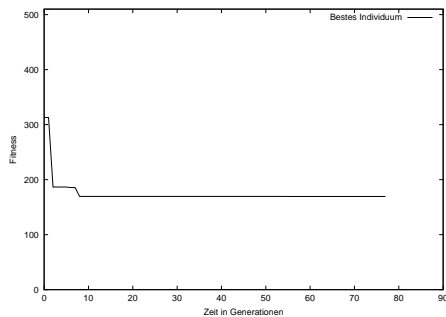


Abbildung 89: Bewertung *Desperadozüge* mit Restart in jeder 30. Gen., Populationsgröße 100

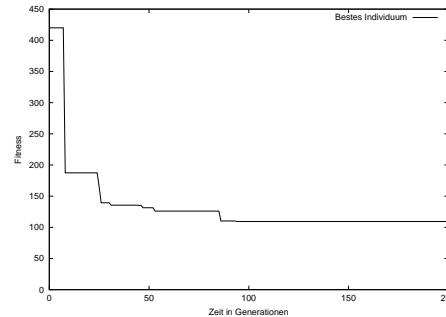


Abbildung 90: Bewertung *Gefährliche Züge* mit Restart in jeder 30. Gen., Populationsgröße 100

(in Richtung Optimum) nicht wesentlich beeinflussen ließen. Bei den restlichen Parametern ist nur zu beachten, dass die Zeitpunkte der Restart-Überprüfung (*RestartGen* von 50 Generationen auf 30 Generationen und die Wahrscheinlichkeit des Austausches von Fitnesscases von 70% auf 30% herabgesetzt wurden. Gelernt wurde mit insgesamt 100 Fitnesscases aus der gesamten Datenbank.¹²⁴

Das Verhalten sämtlicher Läufe der Stufe b ist bis auf wenige Ausnahmen recht ähnlich. Einem starken Abfall (Richtung Optimum) in einer relativ kurzen Zeit zu Beginn folgt eine stagnierende Phase. Trotz der teilweise geringen Generationenzahl war der Rechenaufwand mitunter erheblich.

Auffällig ist dabei die Kurve der Bewertung *Opferzüge* (Abb. 92). Hier finden kaum Veränderungen statt. Das liegt an den in der Datenbank nur in sehr geringem Maße vorhandenen Opferzügen, anhand derer gelernt werden konnte. Dazu wurden die Opferzüge nur auf speziellen Brettern aus der Datenbank trainiert (zur Fitnesscase-Auswahl siehe auch 5.2.1). Das Ergebnis (in Abb. 95)

¹²⁴ Zum Restart-Mechanismus siehe auch 6.1.3 und zum Vorgehen bei der Auswahl der Fitnesscases siehe 5.2.1.

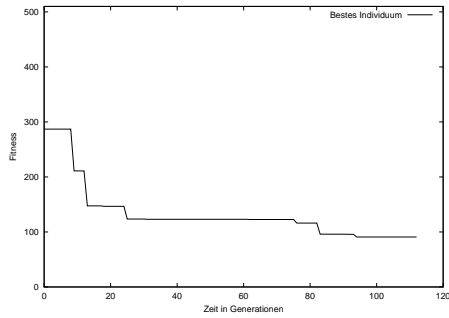


Abbildung 91: Bewertung *Materialzüge* mit Restart in jeder 30. Gen., Populationsgröße 100

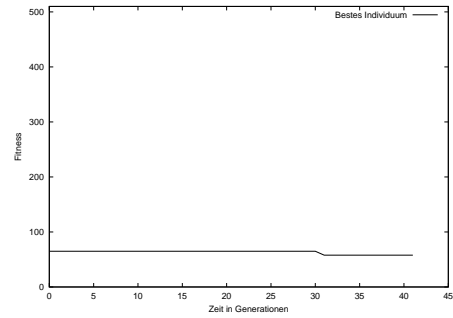


Abbildung 92: Bewertung *Opferzüge* mit Restart in jeder 30. Gen., Populationsgröße 100

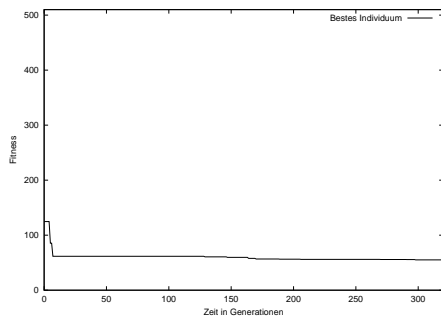


Abbildung 93: Bewertung *Tauschzüge* mit Restart in jeder 30. Gen., Populationsgröße 100

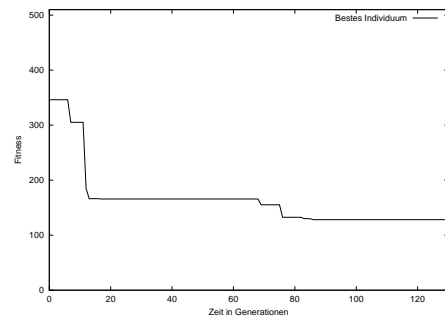


Abbildung 94: Bewertung *Verteidigungszüge* mit Restart in jeder 30. Gen., Populationsgröße 100

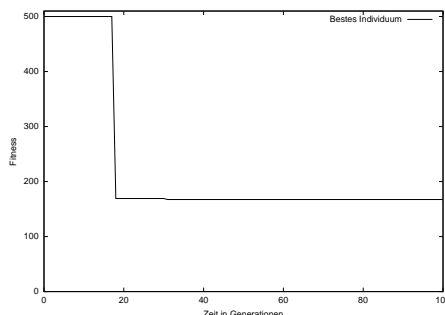


Abbildung 95: Bewertung *Opferzüge* mit spezieller Fitnesscase-Auswahl

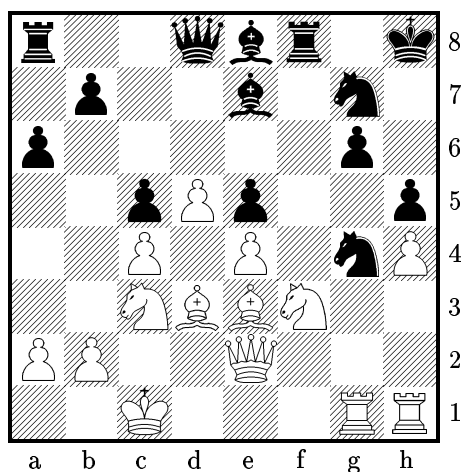
zeigt ein den anderen Kurven ähnliches Verhalten der Opferzugbewertung. In diesem Lauf wurden zusätzlich die Populationsgröße auf 500 und die Anzahl der Fitnesscases auf 500 vergrößert.

Weitere Versuche mit Veränderungen verschiedenster Parameter, hier vor allem die Individuengröße, brachten keine wesentlichen Veränderungen im Verlauf der Fitnesskurve mit sich. Eine Analyse der von den hierbei entstandenen besten Individuen zurückgegebenen Zuglisten findet sich im folgenden Abschnitt.

7.4.3 Analyse der Zuglisten in Stufe b

Zur Beurteilung des Erfolgs der verschiedenen Individuen der Stufe b wurde zu vier ausgewählten Stellungen aus der Datenbank von allen evolvierten b-Modulen eine Zugliste ermittelt. Die Testpositionen wurden von einem Schachexperten so ausgewählt, dass verschiedene Qualitäten der unterschiedlichen b-Module sichtbar werden sollten, und sie entstammen den Partien 22, 64, 81 und 90 aus [MAT96]. Die zurückgelieferten Zuglisten wurden sodann von ebendiesem Schachexperten unter menschlichen Augenschein genommen.

Brett: 1734 – Spassky-Ghitescu, Beverwijk 1967



Dies ist eine Stellung mit geschlossenem Zentrum, d.h. in der Brettmitte blockieren sich weiße und schwarze Bauern gegenseitig. Solche Stellungen sind für

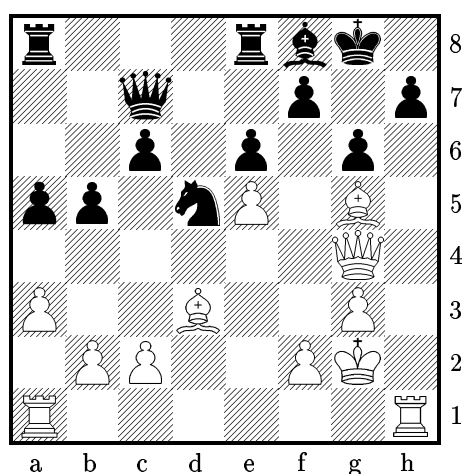
auf Baumsuche basierende Schachprogramme schwer zu spielen, weil konkrete Aktionen meist jenseits ihres Suchhorizonts liegen. In dieser konkreten Stellung jedenfalls kann der Weißspieler am Zug mit 22.Tg1×g4, einem sogenannten *Qualitätsoffer*, d. h. der Hergabe eines Turmes gegen einen generell als schwächer eingeschätzten Springer, schon in deutlichen Vorteil kommen. Dieser Vorteil ist jedoch langfristig-dynamischer Natur und wird von Computerschachprogrammen in der Regel nicht erkannt. Nichtsdestotrotz gibt es hier auch andere Züge und damit verbundene Pläne.

Zu dieser wie auch später zu den folgenden Positionen seien hier die zurueckgelieferten Züge der jeweiligen evolvierten Individuen aufgeführt.

- Verteidigungszüge: Kc1–b1, Kc1–c2, Kc1–d1, Kc1–d2
- Tauschzüge: Le3×c5, Sf3×e5
- Opferzüge: Kc1–b1, Kc1–c2, Kc1–d1, Kc1–d2
- Materialzüge: Kc1–b1, Kc1–c2, Kc1–d1, Kc1–d2
- Gefährliche Züge: Le3–d2, Le3–d4, Sf3–d2, Sf3–d4, Tg1–d1, Kc1–d1, Kc1–d2, Sc3–d1, d5–d6, De2–d1, De2–d2
- Desperadozüge: a2–a3, a2–a4, Sc3–a4
- Beste Züge: Kc1–b1, Kc1–c2, Kc1–d1, Kc1–d2
- Angriffszüge: Sc3–b5, Le3–g5, Sf3–g5, Le3×c5, Sf3×e5

Richtig „interessante“ Züge wurden nicht ermittelt. Daher wird auf eine gesonderte Besprechung der einzelnen Stellungen verzichtet und hinterher ein Gesamt-Resumee gezogen.

Brett: 4532 – Tal-Gurgenidze, SSSR 1969

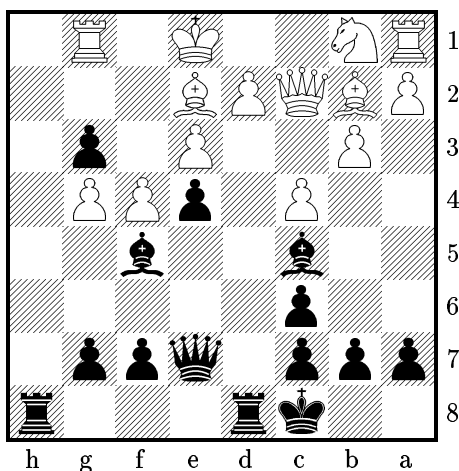


In dieser Stellung konnte der selige „Schachzauberer“ Tal seinerzeit mit der guten Möglichkeit 21.Th1×h7 einen Mattangriff starten. Dieser Zug opfert einen

Turm gegen einen Bauern und erfordert präzise Variantenberechnung. Ein einfacherer Plan wäre es gewesen, vor einem Einschlag die beiden Türme in der h-Linie zu *verdoppeln*, also beispielsweise auf den Feldern h1 und h2 eine Batterie aus den beiden Türmen aufzubauen.

- Verteidigungszüge: Kg2–f1, Kg2–f3, Kg2–g1, Kg2–h2, Kg2–h3
- Tauschzüge: Ld3×b5, Ld3×g6, Dg4×e6, Th1×h7
- Opferzüge: Kg2–f1, Kg2–f3, Kg2–g1, Kg2–h2, Kg2–h3
- Materialzüge: Kg2–f1, Kg2–f3, Kg2–g1, Kg2–h2, Kg2–h3
- Gefährliche Züge: Dg4–a4, Ta1–a2, a3–a4
- Desperadozüge: Dg4–a4, Ta1–b1, a3–a4
- Beste Züge: Kg2–f1, Kg2–f3, Kg2–g1, Kg2–h2, Kg2–h3
- Angriffszüge: Ld3–f5, Dg4–f5, Dg4–h5, Th1–h5, Ld3×b5

Brett: 5755 – Larsen-Spassky, Beograd 1970

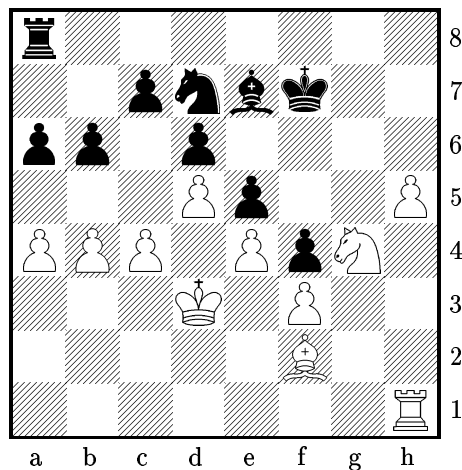


In dieser Stellung (Schwarz am Zug!) hat der Weißspieler seinen König noch nicht mit der Rochade in Sicherheit gebracht, und Schwarz verfügt obendrein über einen gefährlichen Freibauern auf g3, der sich sowohl in eine neue Dame zu verwandeln droht, als auch dem weißen König das Fluchtfeld f2 nimmt. Dafür hat der Schwarze jedoch mit einem Springer einen hohen Preis bezahlt. Das Spiel wird kurzfristig entschieden, denn Weiß, so der Spieler denn rasch seine Stellung befestigen kann, wird den Mehrspringer sicherlich in einen Sieg verwandeln. Derartige Stellungen werden von den derzeitigen herkömmlichen Schachprogrammen bereits meisterhaft behandelt. Der stärkste Zug in dieser Stellung ist 14. ... Th8–h1.

- Verteidigungszüge: Kc8–b8, Kc8–d7

- Tauschzüge: Lc5×e3, Td8×d2, Lf5×g4
- Opferzüge: Kc8–b8, Kc8–d7
- Materialzüge: Kc8–b8, Kc8–d7
- Gefährliche Züge: Lf5–d7, Kc8–d7, Lc5–d6, Lc5–d4, Td8–d7, Td8–d6, Td8–d5, Td8–d4, Td8–d3, Td8×d2, De7–d6, De7–d7
- Desperadozüge: a7–a6, a7–a5, Lc5–a3
- Beste Züge: Kc8–b8, Kc8–d7
- Angriffszüge: Lc5–b4, Lc5–d4, Td8–d4, De7–h4, Th8–h4, Lf5×g4

Brett: 6289 – Larsen-Kavalek, Solingen (m/5) 1970



Diese Stellung ist unspektakulär im Verhältnis zu den vorangegangenen. Weiß verfügt über sogenannten *positionellen Vorteil*. Sein Bauer auf h5 ist seines schwarzen Gegenübers bar, und Weiß kann ihn langfristig in eine Dame umwandeln. Eine ganze Strauß von Zügen kommt hier grundsätzlich in Betracht, zielstrebig wäre z. B. 36. h5–h6, und Bent Larsen spielte 36. Sg4–h6+.

- Verteidigungszüge: Kd3–c2, Kd3–c3, Kd3–d2, Kd3–e2
- Tauschzüge: Lf2×b6, Sg4×e5+
- Opferzüge: Kd3–e2, Kd3–d2, Kd3–c3, Kd3–c2
- Materialzüge: Kd3–e2, Kd3–d2, Kd3–c3, Kd3–c2
- Gefährliche Züge: Th1–c1, Lf2–c5, Kd3–c3, Kd3–c2, c4–c5
- Desperadozüge: Th1–a1, a4–a5
- Beste Züge: Kd3–c2, Kd3–c3, Kd3–d2, Kd3–e2
- Angriffszüge: a4–a5, b4–b5, c4–c5, Lf2–c5, Sg4×e5

Betrachtet man die Stellungen und die vorgeschlagenen Züge, kann man das Ergebnis als erschreckend schlecht beurteilen. Von den zu den Teststellungen vorgeschlagenen 125 Zügen wurde nur ein einziger wirklich gespielt, und die meisten sehen sehr „geraten“ aus. Das Ziel, mit den Individuen der Stufe B den Gesamtsuchraum plausibel beschränkende Zuglistengeneratoren zu evolvieren, ist somit fehlgeschlagen.

Betrachten wir nun die Individuen im Zusammenhang mit ihrer direkten Zielvorgabe, der jeweiligen Fitnessfunktion.

1. Beste Züge, Materialzüge, Opferzüge, Verteidigungszüge

Alle diese Individuen liefern zu den obigen Positionen genau die Liste aller legalen Königszüge zurück, immerhin in teilweise unterschiedlicher Reihung. Man kann davon ausgehen, dass dies auf eine Tendenz in der Auswahl der Lernbeispiele zurückzuführen ist. In den Stellungen der Datenbank befinden sich nur Stellungen, die aus dem Spiel von Experten entstanden sind, daher ist ihnen eine gewisse „Wohlgeformtheit“ zu eigen. Dazu gehört, dass beide Parteien in einer „normalen“ Stellung, den eigenen König hinter einem Wall aus Bauern verbergen. In derartigen Positionen kann der König in der Regel höchstens ein Feld nach rechts oder eines nach links gehen, und dann im nächsten Zug wieder zurück. Durch derartige Züge wird die Stellung nicht wesentlich verändert, aber die Möglichkeit, einen schlechten Zug zu machen, an den Gegner weitergereicht. Ein derartiges „Vorgehen“ des Individuums schützt es also vor Fehlern, hat ergo gleichsam eine hervorragende „Richtig-Negativ-Ausbeute“.

2. Desperado-Züge

Es fällt auf, dass sich die Zielfelder aller von diesem Individuum vorgeschlagenen Züge auf der a-Linie befinden. Eine besondere „Desperado-Affinität“ der a-Linie wäre dem Schachexperten bekannt. Wahrscheinlich liegt die Wurzel des Problems in der internen Darstellung der Schachbrettlinie „a“ durch die Integerzahl 1, mit der Initialisierung aller Register durchgeführt wird.

3. Gefährliche Züge

Dieses Modul ermittelt stets eine Menge von Zügen, deren Zielfeld jeweils auf der gleichen Linie liegt. Die Ziellinie variiert aber je nach Stellung. Ein derartiges Konzept könnte schachliche Raffinesse niederer Art beinhalten, wenn die Linie des Zielfeldes beispielsweise mit der Linie des gegnerischen Königs zusammenfiel. Ein von einem Schachexperten leicht erkennbarer Zusammenhang zwischen der gewählten Linie und der Ausgangsposition liegt jedoch hier nicht vor.

4. Angriffszüge

Das Individuum liefert genau alle Züge mit Zielfeld auf der fünften Reihe (bzw. vierten Reihe aus der Sicht von Schwarz). Dieses primitive Konzept scheint sich im Verhältnis zu seiner Beschreibungslänge sehr gut als Filter für Angriffszüge zu eignen. Mit dem anspruchsvollen Design der Operationenmenge wurden jedoch ursprünglich ehrgeizigere Ambitionen verbunden.

5. Tauschzüge

Das Individuum für Tauschzüge schlägt genau all die Züge vor, welche einen gegnerischen Bauern schlagen. Ähnlich wie bei den Angriffszügen ist diese simple Heuristik recht brauchbar, um den Erfordernissen der Fitnessfunktion gerecht zu werden. Rückblickend auf die Entwurfsphase der GP-Operationen stellt sich die Frage, warum bei der Operation *Intersection* nicht auch ein „Größer-Gleich-Modus“ ermöglicht wurde, mittels dessen sich hier beliebige Schlagzüge oder oben bei den Angriffszügen Züge auf die „mindestens fünfte“ Reihe kurz beschreiben ließen.

Als Fundament eines anspruchsvollen Schachsystems scheinen jedoch auch diese beiden letzten Zuglistenmodule tönern.

Im Hinblick auf die für die Stufe C zu erbringende Vorauswahl-Funktionalität kann festgestellt werden, dass alle hier vorgestellten Module häufig kurze Zuglisten mit oft nur drei oder vier Zügen aufstellen. Eine so kleine Auswahl mag, so sie denn eine von den „vernünftigen“ Zügen ist, genügen, aber bei den evolvierten Individuen ist von einer Plausibilität dieser Grundmenge nicht auszugehen.

Zumindest in den der Jetzt-Stellung nahen Astgabeln des Spielbaums könnte es sicherlich zu besseren Spielergebnissen führen, stellte man dem C-Individuum eine komplette Liste aller legalen Züge zur Verfügung.

Der Grad des positiven Beitrags der Stufe B zum Gesamterfolg des geschaffenen Schachsystems ist also insgesamt kritisch zu beurteilen.

7.4.4 Versuche ohne Längenbegrenzung der Zuglisten

Aufgrund der enttäuschenden Ergebnisse, die mit den Individuen der Stufe b nur erzielt werden konnten, wurden die Fitnessfunktionen der Stufe b geändert. Als besonders störend erwies sich bei den erhaltenen Ergebnissen die Kürze der Listen, die die Individuen der Stufe c stark zu beschränken drohte.

Dadurch motiviert, wurde bei den Bewertungsfunktionen der Stufe b auf die Anwendung der Strafterme für zu große Listenlänge verzichtet.

Es war zu erwarten, dass nach diesem Eingriff die Module mit hoher Fitness allesamt die komplette Liste aller in einer Stellung möglichen Züge in einer individuellen Permutation zurücklieferten, was auch eintrat.

Diese Permutationen erwiesen sich als einer menschlichen Analyse unzugänglich. Innerhalb der Liste fand sich kein einfaches Gruppierungskriterium der Züge.

Dieses mag daran liegen, dass die Listen hier offensichtlich nicht inkrementell mit dem *subset*-Kommando aufgebaut werden, sondern mit *getAll* direkt in Gänze geladen und anschließend permutiert werden.

Permutationen bereits fertig aufgebauter Listen werden jedoch von der Operationenmenge nicht besonders unterstützt, und sinnvolle Umordnungen der Liste scheinen überhaupt nur mit langen Befehlssequenzen möglich, deren Evolution nach allem zuvor von CHESGP Erreichten man zumindest als unwahrscheinlich ansehen würde.

Da die ohne Beschränkung der Listenlängen evolvierten Individuen nur kom-

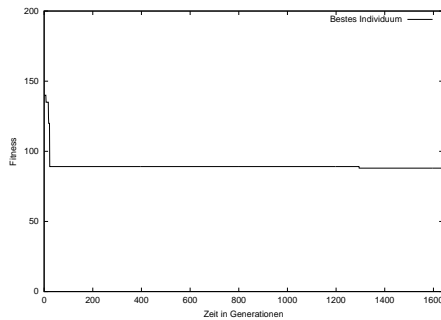


Abbildung 96: Bewertung *Tauschzüge* ohne Beschränkung der Listenlänge

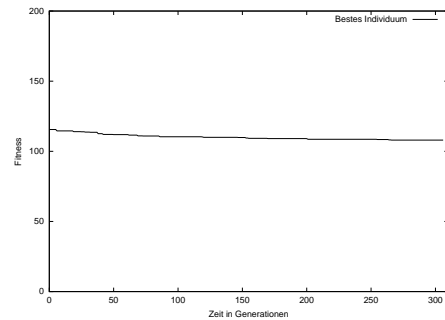


Abbildung 97: Bewertung *Materialzüge* ohne Beschränkung der Listenlänge

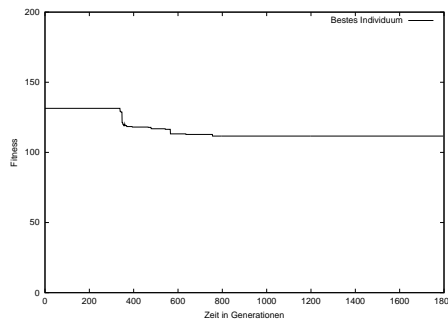


Abbildung 98: Bewertung *Opferzüge* ohne Beschränkung der Listenlänge

plette Zuglisten in intuitiv willkürlich anmutender Reihung erzeugten, wurde auf ein weitergehendes Beschreiten dieses Pfades verzichtet.

Zur Verdeutlichung des Evolutionsvorganges seien hier noch drei Graphen für Fitnessverläufe des jeweils besten Individuums ohne Beschränkung der Listenlänge angefügt:

Dabei wurden die Bewertungen *Tauschzüge* (Abb. 96) und *Beste Züge* (Abb. 98) mit einer Populationsgröße von 50 und doppelt so großen Individuen wie bisher auf der gesamten Datenbank mit 100 Fitnesscases (und einer Austauschwahrscheinlichkeit der Fitnesscases von 70%) trainiert. Der Lauf der Bewertung *Materialzüge* (Abb. 97) basiert auf einer Populationsgröße von 50 und ebenfalls großen Individuen. Die 100 Fitnesscases aus der gesamten Datenbank werden aber nicht ausgetauscht, so dass sich die Lernbedingungen während des Laufs nicht verändert haben.

Es fällt auf, dass in den Kurven lediglich die typische steile Abwärtsphase zu Beginn fehlt. Ansonsten zeigt sich das stagnierende Verhalten – sogar in den gleichen Wertebereichen, wie in den Läufen ohne Beschränkung der Listenlänge – direkt von Anfang an. Das legt die Vermutung nahe, dass der schnelle Fitnessfortschritt in der Anfangsphase der anderen Läufe lediglich auf die Verkürzung und nicht auf eine inhaltliche Verbesserung der zurückgegebenen Listen zurück-

zuführen ist.

7.4.5 Schlussbemerkung zur Evolution der Stufen a und b

Insgesamt lässt sich für die Evolution der beiden „Hilfsstufen“ festhalten, dass in beiden Fällen das Ziel – Evolvierung von Programmen, die für bestimmte Schachbretter mit einiger Sicherheit eine Bewertung oder eine Zugfolge zurückliefern – nicht erreicht wurde. Die Gründe hierfür müssten genauer untersucht werden. An dieser Stelle wählten wir das Vorgehen, mit einigen Individuen in der nächsten Stufe weiterzuarbeiten, die zwar „von Hand“ erstellt wurden, aber nur unter Benutzung von Syntax und Sprachumfang des Interpreters und der Operationenmenge aus den GP-Läufen. So spricht nichts dagegen, dass diese „künstlichen“ Individuen nicht auch aus der Evolution hätten hervorgehen können. Zum Anderen können die Individuen der Stufen a und b in Stufe c wieder verändert werden. Das ergibt neue Möglichkeiten für eine Evolution.

7.5 Analyse der Stufe c

Das Ziel der Projektgruppe 348 ist die Evolution von „guten“ schachspielenden Programmen¹²⁵. Der erste Abschnitt präzisiert dieses Vorhaben, und liefert die Klärung, was ein „gutes“ Individuum auszeichnet. Die folgenden Abschnitte widmen sich der Analyse, inwiefern dieses Ziel erreicht wurde.

Aufgrund der knappen Zeit konnten nur wenige Analysen durchgeführt werden. Dieses Kapitel enthält einige Hinweise auf weitere Analysen, welche nicht durchgeführt worden sind.

Im Abschnitt 7.5.2 wird das Spielverhalten, sowie die Spielstärke untersucht. Können die Programme etwa gegen einen Menschen oder gegen andere Schachprogramme gewinnen?

Abschnitt 7.5.3 beschäftigt sich mit dem Aufbau von erfolgreich spielenden Individuen. Unterscheidet sich die vom Individuum realisierte Baumsuche von den Standardverfahren?

Der letzte Abschnitt zeigt die Entstehung spielstarker Individuen im Evolutionsverlauf auf.

7.5.1 „Gute“ Individuen – das Evolutionsziel

Hauptaufgabe der Projektgruppe 348 ist nicht die Erstellung eines spielstarken Schachsystems, sondern vielmehr das Auffinden von akzeptablen Schachprogrammen unter Zuhilfenahme der Genetischen Programmierung. Akzeptabel bedeutet dabei ein möglichst günstiges Verhältnis zwischen Ausdruckskraft und Spielstärke der Individuen. Unter Ausdruckskraft wird nicht nur die Vielfalt der Berechnungen aufgefasst, zu welchen ein Programm theoretisch in der Lage ist, sondern auch, wie einfach diese realisiert werden können. Im Falle eines Individuums, welches die Informationen des Schachbrettes in Form von Konstanten

¹²⁵ Hier auch als Individuen bezeichnet.

einsehen kann, und ausschließlich arithmetische Operationen zur Wahl eines Zuges heranziehen darf, ist es bereits ein Erfolg, gegen einen Spieler zu gewinnen, der zufällig¹²⁶ einen Zug spielt. Wie in Kapitel 6.1 beschrieben, vollziehen alle Individuen eine Baumsuche. Dies begünstigt sicherlich spielstärkere Individuen, hebt aber auch gleichzeitig die Ansprüche hinsichtlich der Spielstärke.

Insgesamt ist erwünscht, dass Individuen mit folgenden Eigenschaften aus dem Evolutionsprozess hervorgehen:

1. Die Individuen sollen so spielstark sein, dass einem Schachlaien interessante Partien geboten werden.
2. Es soll eine Suche realisiert werden, welche vom reinen Minimaxverfahren abweicht.
3. Das Individuum soll ressourcenschonend arbeiten. Dies bedeutet, dass es nicht allzuviele Knoten im Suchbaum expandieren darf.

Die letzten beiden Punkte bedürfen näherer Erläuterung. Da die von den Individuen berechneten Bewertungen an den Bättern des Suchbaumes gemäß dem Minimax-Verfahren an die Wurzel zurückgegeben werden (vgl. Kapitel 6.1.1), könnte ein Individuum auf einfache Weise Spielstärke erreichen, indem es ein reines Minimaxverfahren implementiert. Dazu müssten folgende Voraussetzungen gegeben sein: unter den a-Programmen existiert eine Materialbewertung, die von den Blattmodulen ohne anschließende Modifikation aufgerufen wird. Weiterhin dürften die Operationen der Entscheidungsmodule die Expansion des Baumes an einem Knoten nur dann abbrechen, wenn eine konstante Tiefe erreicht ist. Außerdem müsste das ausgewählte b-Individuum eine vollständige Zugliste zur Aufspannung des Baumes zurückliefern.

Es ist selbstverständlich erwünscht, dass eine Suche durchgeführt wird, welche ebenbürtige oder bessere Züge liefert, aber dafür weniger Knoten betrachten muss. Dies wird dadurch gefördert, dass der Anzahl der Knoten im expandierten Suchbaum eine Grenze gesetzt wird. „Bessere“ Individuen nutzen die beschränkten Ressourcen hoffentlich effizienter aus, als das reine Minimaxverfahren.

7.5.2 Spielverhalten und Spielstärke der Individuen

Um das Spielverhalten zu testen, wurden zahlreiche Partien Mensch gegen Individuum gespielt. Damit sollten Fehler in der Realisierung gefunden und die Spielstärke der Individuen abgeschätzt werden. Ab und zu verhielten sich die Individuen unerwünscht:

- Viele Individuen spielten immer den ersten Zug aus der Liste der in dieser Stellung überhaupt möglichen Züge. Das war auf Programmierfehler zurückzuführen, und wurde behoben. Trotzdem ist es prinzipiell weiterhin möglich, dass Individuen immer den ersten möglichen Zug liefern.

¹²⁶ Gemäß uniformer Verteilung.

- Fast alle Individuen vernachlässigten die Figurenentwicklung zu Beginn eines Spiels. Meist wurde mit weniger als drei Figuren gespielt. Deswegen wurden selbstentworfenen Schachprogramme in den Pool aufgenommen, die solche Individuen in wenigen Zügen Matt setzen. Durch die erhöhte Anzahl an verlorengegangenen Spielen verringert sich die Fitness der Individuen.
- Die Individuen zeigten eine sehr aggressive Spielweise. Oft wurden eigene Figuren zum Schlagen angeboten, ohne dass sich langfristig ein Vorteil geboten hätte. Die Wahrscheinlichkeit, dass aggressive Züge zur Expansion im Suchbaum benutzt wurden, war zu groß.¹²⁷ Als voreingestelltes Modul wird nun Beste-Züge benutzt, das GP-Individuum kann aber auch weiterhin eine andere Wahl treffen.

Es kam aber auch zu erfreulichen Resultaten. So wurde ein Individuum evolviert, welches eine hinreichend starke Spielstärke aufweist, um einem Schachlaien eine interessante Partie zu bieten. Dabei ist es sogar vorgekommen, dass das Individuum als Sieger aus dem Spiel hervorgeht.

Die Spielstärke soll nicht nur aus den Spielen gegen Menschen bestimmt werden, sondern auch in Spielen gegen verschiedene (selbstgeschriebene) Schachprogramme:

- Das Schachprogramm „Zufall“ liefert einen zufälligen Zug, jeden einzelnen mit der gleichen Wahrscheinlichkeit. Jedes Programm und jede Person mit geringen Schachfertigkeiten wird erwartungsgemäß mehr als die Hälfte der Partien gegen Zufall gewinnen. Zufall ist natürlich ein sehr schlechter Gegner. Allerdings ist jeder Zug möglich. Und gegen ein Individuum, welches nur bestimmte Figuren zieht, hat Zufall gute Chancen.
- Zufällige Individuen, also etwa die Individuen der Startgeneration, werden ebenfalls als Gegner ausprobiert. Zufällige Individuen sollten besiegt werden, falls sich das zu testende Individuum in einem längeren Evolutionslauf als Bestes ergeben hat.
- Desweiteren wird gegen eine Art Minimax-Algorithmus gespielt. Dieser benutzt eine clevere Materialbewertung, die einige positionelle Kriterien berücksichtigt. Der Algorithmus berechnet den Minimaxwert und somit den besten Zug, ist aber um einiges effizienter als das Minimax-Verfahren. Je nach Suchtiefe erhält man unterschiedlich starke Gegner. Programme der Tiefe 4 sind für den Schachlaien kaum noch zu bezwingen.

Der Tabelle 11 ist zu entnehmen, wie gut eines der besten Individuen gegen die einzelnen Programme besteht. Die auffallend hohe Anzahl an Remis in Spielen gegen zufällig initialisierte GP-Individuen, ist auf die Tatsache zurückzuführen, dass diese Spiele deterministisch ablaufen und somit hin und wieder in Endlosschleifen enden. Informationen über die Evolution in der dieses Individuum entstanden ist, finden sich in Abschnitt 7.5.4.

¹²⁷ Es wurde fast immer das voreingestellte Aggressive-Züge b-Programm zur Expansion benutzt.

Gegner	Zufall	zufälliges Indivi- duum	Minimax (1)	Minimax (2)	Minimax (3)
Sieg (für das Individuum)	100	87	95	38	0
Überlegenheit	0	0	3	18	0
Remis	0	13	0	13	5
Unterlegenheit	0	0	2	13	8
Niederlage	0	0	0	18	87

Tabelle 11: Spielausgänge der Spiele eines der besten Individuen gegen diverse Gegner

7.5.3 Aufbau spielstarker Individuen

Ein Individuum besteht aus vier Modulen der Stufe c und je acht Programmen der Stufen a und b. Die Analyse der a- und b-Programme wurde in den vorhergehenden Kapiteln durchgeführt.

Die zunächst sehr ernüchternden Ergebnisse in der Stufe b haben dazu geführt, dass zwei b-Programme von Hand entworfen wurden. Diese hätten im Prinzip durch Evolution entstehen können und wurden bei der Evolution in Stufe c eingesetzt. Im folgenden werden diese Programme beschrieben.

7.5.3.1 Selbstentworfenen b-Individuen

Da die Analyse der evolvierten Individuen der Stufe B nicht die gewünschten Ergebnisse zeigte, wurden zu den GP-Individuen schnittstellengleiche Module implementiert. Damit sollte zumindest die Tragfähigkeit des verfolgten Gesamtkonzepts mit einem guten Resultat in der Stufe c unter Beweis gestellt werden. Beim Entwurf dieser wurden drei Zielvorgaben im Auge behalten:

1. Die Module sollten mit den Sprachmitteln der Operationenmenge der Stufe b darstellbar sein.
2. Die Module sollten auf die formale Spezifikation der jeweiligen Bewertung ausgerichtet sein und daher „unter Evolutionsbedingungen“ einen guten Fitnesserfolg verzeichnen können.
3. Ein Zusammenhang der zurückgelieferten Züge zum schachlichen Konzept der jeweiligen Bewertung (wie z. B. „Angriffszüge“) sollte intuitiv erkennbar sein.

Die Ausdruckskraft der in der Stufe b verwandten Sprache ist sehr beschränkt. Es kann intern kein Zug probenhalber ausgeführt werden. Dieses scheint jedoch bei Bewertungen wie „Tauschzüge“ oder „Materialzüge“ unerlässlich für die Erfüllung obiger Postulate zu sein. Daher wurde darauf verzichtet, für alle acht Bewertungsfunktionen ein eigenes Musterindividuum zusammenzustellen.

Dies bedeutet nicht, dass aus den der Stufe b zur Verfügung gestellten Sprach-elementen keine geeigneten Lösungen gebildet werden können, sondern lediglich, dass sich die Teilnehmer der Projektgruppe nicht dazu in der Lage sahen, selbst eine solche Lösung zu formen, die allen obigen Forderungen und insbesondere der zweiten genügt. Es wurden also nur zwei solcher Module implementiert:

- Beste Züge

Das Ersatzmodul für die Teilaufgabe „Beste Züge“ liefert Züge in dieser Reihenfolge: Schlagzüge, die kurze Rochade, Entwicklungszüge, Züge in die gegnerische Bretthälfte hinein und Züge auf die Zentrallinien.

Innerhalb der Menge der Schlagzüge wird die Reihung bestimmt durch die Wertigkeit des geschlagenen Materials. Ob der schlagende Stein im Fortlauf einer plausiblen Spielsequenz ebenfalls verloren ginge, wird nicht berücksichtigt. Es schließt sich an die kurze Rochade, da es meist eine gute Idee ist, den eigenen König am Brettrand und hinter Bauern zu verstecken. Unter *Entwicklungszügen* werden Züge verstanden, deren Startfeld auf der eigenen Grundreihe liegt. Man beachte, dass beispielsweise auch unsinnige Königszüge oft dieser Bedingung genügen. Danach folgen in der Rückgabezugliste Züge, die (in dieser Reihenfolge) auf die 8., 7., 6. oder 5. Reihe zielen. Zuletzt folgen Züge auf Felder der e- oder d-Linie. Als Standardsortierkriterium innerhalb der beschriebenen Teilmengen findet die lexikographische Ordnung der Start- und dann präzisierend Zielfelder Verwendung.

Dieses künstliche B-Modul entspricht in etwa einem auf Baumsuche basierenden Schachprogramm mit gängiger Stellungsbewertung und einer Suchtiefe von *einem* Halbzug. Im Unterschied zu vielen evolvierten Individuen liefert dieses eventuell „zu viele“ Züge an die Stufe C. Die dort vorhandenen Operationen „cut“ und „exit“ ermöglichen es aber, vorgeschlagenen Züge schnell zu verwerfen.

- Gefährliche Züge

Dieses Modul berechnet eine echte Teilmenge der durch das zuvor beschriebene „Beste Züge“-Individuum gefundenen Züge. Nur Schlagzüge und Züge auf die 8. oder 7. Reihe werden hier berücksichtigt.

Die hier zugrundeliegende Bewertungsfunktion verwendet eine zwar vorausschauende, aber doch naive, Suche. Und zwar werden (den Regeln natürlich nicht entsprechende) Doppelzüge, also zwei Züge einer Partie direkt hintereinander, zugelassen. Der erfolgreichste Doppelzug ist dann ein „gefährlicher Zug“.

Solche Züge lassen sich, wenn auch nicht erschöpfend, auch ohne eine Expansion des Spielbaums feststellen. So sind z. B. alle Schlagzüge „gefährlich“, weil die schlagende Figur im zweiten Zug einfach wieder auf ihr Ausgangsfeld zurückkehren könnte (dies gilt natürlich nicht für Bauern, aber jeder mögliche geschlagene Stein ist mindestens so viel wert wie ein

Tiefe	2	3	4	5	6	7
Minimax	1201	38303	1489509	49123919	–	–
$\alpha\beta$	529	11105	99823	4192966	21122367	–
NegaScout	610	11171	87692	3658054	15331810	–
F-NegaScout	529	10119	84586	3569096	14981259	–
Iterat. F-NegaScout	148	2454	28966	241712	2595195	41520237
F-NegaScout (a)	1201	21807	186498	5980860	34405479	–
Iterat. F-NegaScout (a)	1201	19801	107096	668836	10791164	–

Tabelle 12: Anzahl besuchter Knoten verschiedener Baumsuchverfahren

Bauer). Die Züge auf die letzten Reihen der Gegenpartei werden zugelassen in der Vermutung des sich dort aufhaltenden gegnerischen Königs, der durch Doppelzüge sicherlich leicht in schwere Mattgefahr geriete.

Bei der Beurteilung des Ergebnisses der Stufe c gilt es, die Verwendung dieser beiden, den reinen GP-Ansatz infragestellenden, b-Module zu beachten.

7.5.3.2 Rückschlussmöglichkeiten von der Anzahl expandierter Knoten im Baum auf die Cleverness der Suche

Der Aufbau eines ganzen Individuums konnte aus Zeitgründen nicht untersucht werden. Die Analyse der 50–500 Kilobytes großen Individuen wird daher aufgrund bestimmter Kennzahlen versucht.

Eine wichtige Kennzahl einer Baumsuche ist die Anzahl der besuchten Knoten im Suchbaum. Besucht ein spielstarkes Individuum nur wenige Knoten, ist die Suche cleverer als Minimax. In der Tabelle 12 sind verschiedene Baumsuchverfahren gegenübergestellt. Die Baumsuchverfahren sind in Kapitel 3.2 vorgestellt worden. Diese berechnen den Minimaxwert und damit auch den besten Zug. Abzulesen sind jeweils die Anzahl der Knoten, die die Verfahren bei der Suche im Suchbaum expandieren. Die Anzahl der Blätter im Baum, die besucht werden, ist von der gleichen Größenordnung. Man beachte, dass an jedem Blatt eine Stellungsbewertung erfolgt.

Die beiden Negascoutverfahren, die in der Tabelle mit (a) markiert sind, berechnen darüberhinaus den Minimaxwert jedes Zuges; es sind also keine Schnitte an der Wurzel erlaubt. Die Verfahren benutzen eine einfache Materialbewertung an den Blättern. Die Negascoutprogramme, die in Stufe c als Spieler auftauchen, unterscheiden sich in zwei Punkten vom Iterativen F-NegaScout-Verfahren (5. Zeile). Zum einen liefern sie, wenn es mehrere beste Züge gibt, einen zufälligen von diesen. Zum anderen benutzen sie eine clevere Stellungsbewertung, wodurch sie spielstärker und schneller¹²⁸ werden.

¹²⁸ Aufgrund des ersten Punktes reicht es nicht aus, dass ein Zug maximal genauso gut wie eine gefundene Alternative ist, um diesen nicht weiter zu betrachten. Er muss sich schon als echt schlechter erweisen. Eine feinere Bewertung ermöglicht hier mehr Diversität und somit mehr Schnitte im Suchbaum.

Die Tabelle hat eine stark beschränkte Aussagekraft, da nur eine einzige Schachstellung untersucht wurde. Die Individuen können maximal 20 000 Knoten expandieren, danach wird nach aktuellem Stand zurückbewertet. Mit nur 20 000 Knoten kann ein reines Minimaxverfahren nicht mehr bis zur Tiefe 3 vordringen. Individuen, welche mit Minimaxverfahren der Tiefe 3 konkurrieren können, müssen eine effizientere Suche vollziehen. Die besten bisher beobachteten Individuen, scheinen eine partielle Suche zu betreiben, welche mindestens die Tiefe 2 erreicht. Dabei nutzten sie die maximale Anzahl (20 000) der Knoten bei weitem nicht aus. In einem Testspiel gegen eines der besten Individuen expandierte dieses durchschnittlich 531 Knoten pro Zug, die Standardabweichung ist dabei 431.

7.5.4 Entstehung spielstarker Individuen

Dieser Abschnitt behandelt die Entstehung spielstarker Individuen im Laufe der Evolution. Es liegt ein mehrere Wochen andauernder Evolutionslauf zugrunde, dessen wichtigste Parameter nun erläutert werden. Näheres zum Ablauf der Evolution ist in Kapitel 6.1 zu finden.

Die Population enthält 100 Individuen, 10 Schachprogramme von Typ „Zufall“ und je 5 Programme von Typ Minimax mit den Suchtiefen 1, 2, 3, und 4 (näheres zu den Programmen siehe Abschnitt 7.5.2). Pro Generation werden 100 Spiele gespielt¹²⁹, im Schnitt wird also jedes Programm jede Generation an zwei Spielen beteiligt sein. Ein Spiel wird spätestens nach 80 Halbzügen unterbrochen. Jedes Individuum darf für den Zug im Suchbaum nur bis zur Tiefe 6 vordringen und maximal 20 000 Knoten expandieren.

Jedes fünfte Spiel wird versucht, ein schlechtes Individuum durch ein neues Individuum zu ersetzen. Jedes Individuum hat eine Schonzeit von 7 Spielen. Erst nachdem es sieben Mal gespielt hat, darf es an dem Turnier zur Auswahl des zu ersetzenden Individuums teilnehmen; die Turniergröße beträgt dabei 10 Individuen. Das neue Individuum wird mit je gleicher Wahrscheinlichkeit durch Rekombination oder Mutation erzeugt. Als Eltern kommen nur Individuen in Frage, die bereits 5 Spiele absolviert haben.

Die Abbildungen 99 bis 103 zeigen die Elo-Bewertung und die Veränderung der Spielstärke des besten Individuum jeder Generation.

Jede Generation wird das hinsichtlich der Elo-Bewertung beste Individuum betrachtet. Dieses spielt insgesamt 25 Spiele (nur für die Analyse). Jeweils 5 Spiele erfolgen gegen die Minimaxvarianten der Tiefe 1, 2 und 3. Diese benutzen, wie in Abschnitt 7.5.2 erläutert, eine clevere Bewertung und sind somit der herkömmlichen Minimaxvariante überlegen. Zusätzlich erfolgen noch jeweils 5 Spiele gegen das Programm „Zufall“ und gegen zufällig initialisierte¹³⁰ Individuen.

Jedes Spiel resultiert in einer Zahl aus $[0, 1]$. 1 bedeutet, dass das GP-Individuum gewonnen hat, bei 0 hat es verloren. Ein Spielergebnis in der Höhe 0,5

¹²⁹ Dabei werden immer 30 Spiele parallel gespielt.

¹³⁰ Für jedes Spiel ein neues Individuum.

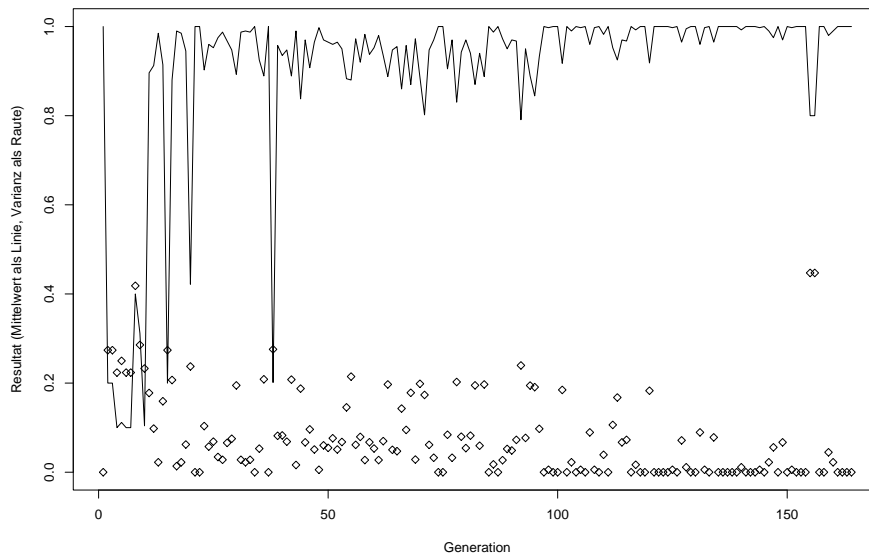


Abbildung 99: Spielergesultnisse gegen das Programm Zufall

deutet auf einen ausgeglichenes Spiel hin. Für jeden Gegnertyp sind in der Abbildung nun der Mittelwert und die Varianz der 5 jeweiligen Werte dargestellt. Darüberhinaus ist der Elo-Wert des besten Individuums und die durchschnittliche Fitness in der Population (samt der Computerprogramme, deren Elo-Wert diesen kräftig anheben) in Abbildungen 104 eingezeichnet.

Aufgrund eines Fehlers in der Realisierung wurde in den Generationen 7 bis 65 weder Rekombination, noch Mutation ausgeführt. Da ein Spiel, im Gegensatz zu der Mutation oder der Rekombination, den Durchschnitt der Elo-Werte nicht ändert, ist er in diesem Zeitraum konstant.

In den Abbildungen 99 bis 103 ist der Fortschritt deutlich erkennbar. Die besten Individuen setzen sich im Laufe der Evolution klar gegen das Programm „Zufall“, gegen zufällig initialisierte GP-Individuen und gegen das verbesserte Minimaxverfahren der Tiefe 1 durch. Einige Individuen sind darüberhinaus spielstärker als das verbesserte Minimaxverfahren der Tiefe 2. Die Evolutionszeit von wenigen Wochen hat allerdings nicht dazu gereicht, dass ein GP-Individuum mit einem Minimaxverfahren der Tiefe 3 konkurrieren kann.

Die Fitnesswerte schwanken stark, obwohl die Menge der Individuum sich nicht ändert. Allerdings sind die Elo-Werte in verschiedenen Generationen, ebenso wie die Elo-Werte von unterschiedlichen Evolutionsläufen nur schlecht vergleichbar, da es sich um eine relative Bewertung (unter den Individuen in der Population zu einem Zeitpunkt) handelt. Es kann jedoch die qualitative Entwicklung des besten Individuums relativ zum Durchschnitt abgelesen werden.

Für eine fundierte Beurteilung der Spielstärke des besten Individuums, sind die Ergebnisse aus den Spielen gegen die selbstgeschriebenen Schachprogramme

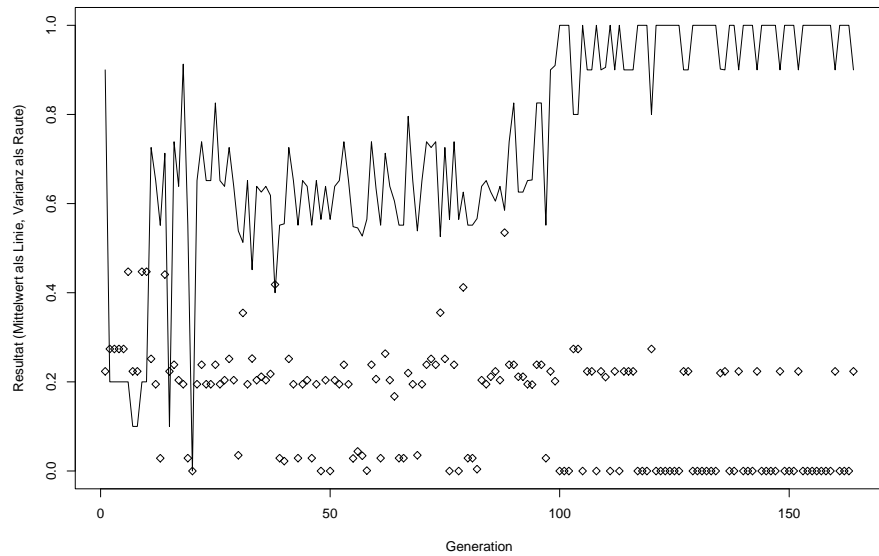


Abbildung 100: Spielergebnisse gegen zufällige Individuen

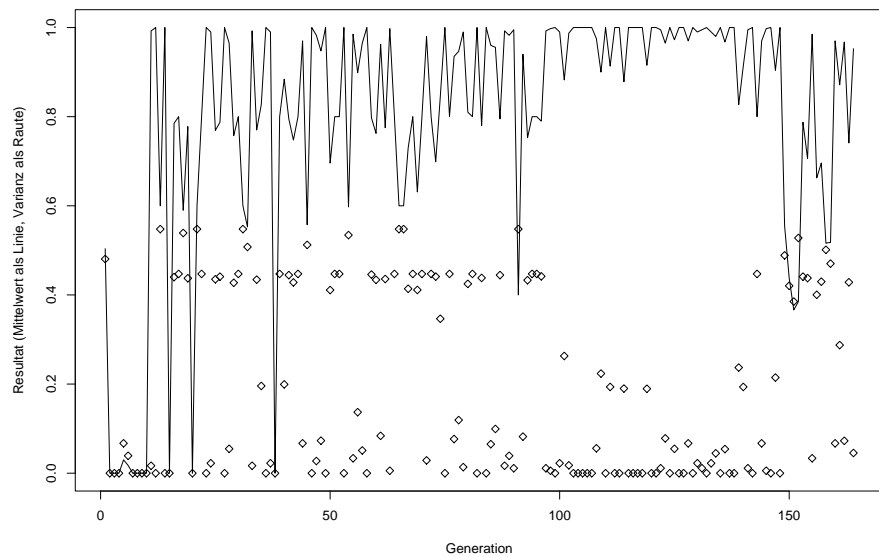


Abbildung 101: Spielergebnisse gegen das Minimaxverfahren bis Tiefe 1

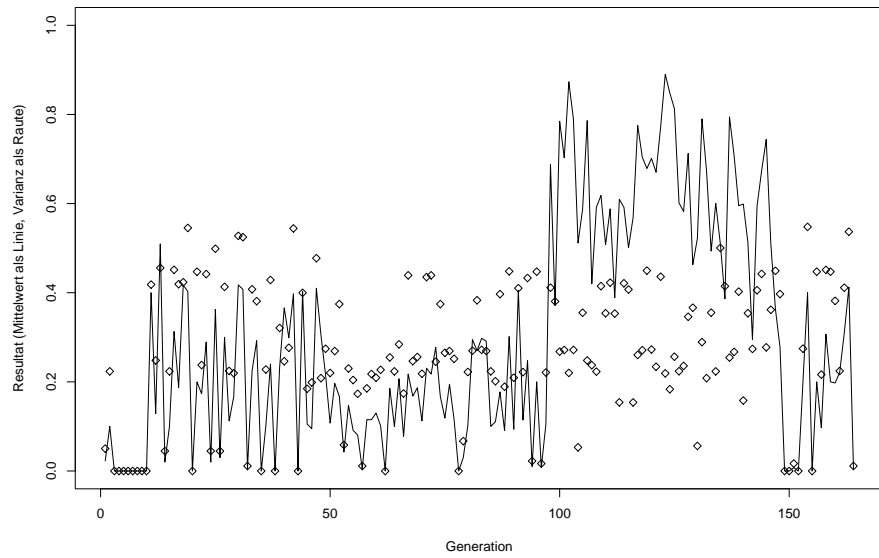


Abbildung 102: Spielergebnisse gegen das Minimaxverfahren bis Tiefe 2

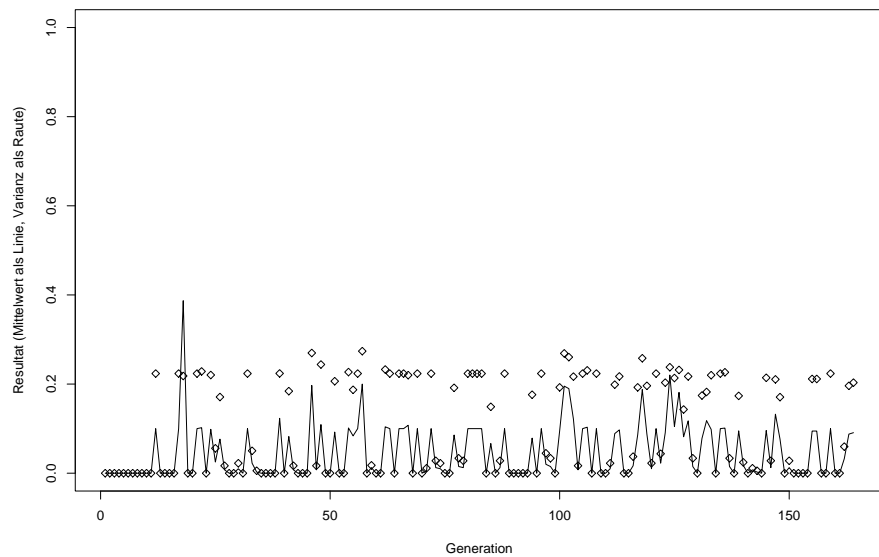


Abbildung 103: Spielergebnisse gegen das Minimaxverfahren bis Tiefe 3

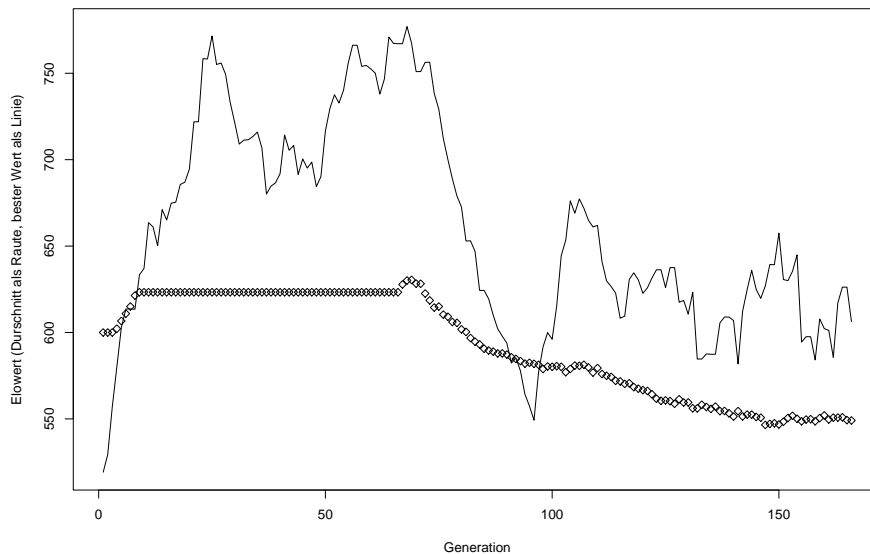


Abbildung 104: Die Entwicklung der Elo-Werte über die Zeit

heranzuziehen. Allerdings sind auch hier kleine Schwankungen nicht allzu schwer zu gewichten. Aufgrund der geringen Anzahl der Spielwiederholungen (fünf mal jeweils), ist es möglich, dass ein und das selbe Individuum leicht unter- oder überbewertet wird. Für eines der besten Individuen wurde die Anzahl der Spiele deswegen in einer weiteren Untersuchung auf 100 hochgesetzt. Die Ergebnisse sind der Tabelle 11 zu entnehmen.

Leider war es nicht möglich, weitere Läufe dieser Größenordnung durchzuführen, da selbst bei der parallelen Berechnung auf 28 Rechnern dazu jeweils mehrere Wochen nötig sind.

7.5.5 Analyse gespielter Partien

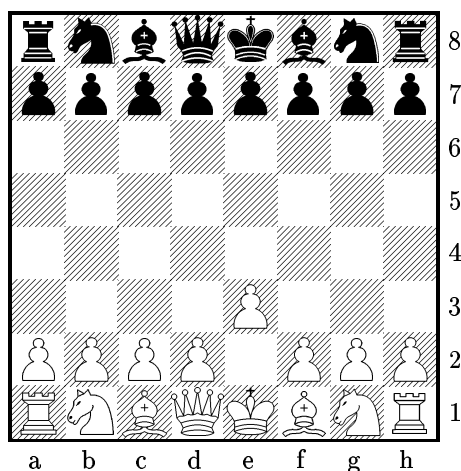
Am Beispiel einiger Partien soll nun das Spielverhalten der evolvierten Individuen beleuchtet werden, wobei auch unerwartete Probleme, die während der Projektarbeit auftraten, besprochen werden sollen.

7.5.5.1 ChessGP-Individuen unter sich

Den Auftakt bildet eine Partie des besten Individuums der 130. Generation unseres einzigen Laufes gegen den Champion der 50. Generation.

Weiß: laufP_R05_130 – Schwarz: laufP_R05_50

1. e2-e3



Es wird ohne Eröffnungsbibliothek gespielt. Im Nachhinein scheint es angemessen, eine Eröffnungsbibliothek mit einem Halbzug (1. e2–e4 für Weiss / 1. ... e7–e5 bzw. 1. ... d7–d5 für Schwarz) anzulegen.

Die gespielte Eröffnung ist nicht wirklich schlecht, aber etwas zaghaft. Im Schach ist es mit der Eröffnung etwa so, wie es sich im Tennis mit dem Aufschlag verhält: Der Aufschlag kann kommen, oder er kann nicht kommen. Ein Zug wie das hier gewählte 1. e2–e3 hat jedenfalls einen Anflug von einem „second service“. Schachexperten bevorzugen fast ausschließlich Eröffnungen, die mit dem Doppelschritt eines Bauern beginnen, was aggressiver ist. Die hier gespielte Eröffnung hat jedoch den Vorteil, dass sie mit Weiß wie mit Schwarz (dann als 1. ... e7–e6) in jeder Partie gespielt werden kann. Dies ist mit hoher Wahrscheinlichkeit während der Evolution von Bedeutung gewesen, denn die Individuen sind „farbblind“ und wissen nicht, ob sie Weiß oder Schwarz haben, und können so ohne zusätzliche if-Zeilen eine „vernünftige“ Eröffnung spielen. Spielte nämlich beispielsweise ein Individuum, das immer mit dem Doppelschritt des d-Bauern eröffnete gegen eines, das gern den e-Bauern doppelt vorzieht, dann könnte nach 1. d2–d4 e7–e5 gleich mit 2. d4×e5 der Bauer des Schwarzen geschlagen werden. Im universellen Sinne ist aus den gezeigten Gründen der Eröffnungszwilling 1. e2–e3 und 1. ... e7–e6 wohl sogar einer der stärksten!

Ein weiterer Vorteil des Zugs 1. e2–e3 ist der, dass er die Möglichkeit des sogenannten „Schäferzugs“ (♠f1–c4, ♔d1–f3, ♚f3×f7, Schachmatt!), denen die Vertreter früherer Evolutionsstufen noch zum Opfer fallen konnten, ausschließt. Es mag sein, dass diese Tatsache im Evolutionsprozess eine Rolle gespielt hat.

1. ... ♞b8–a6

„Springer am Rand bringt Kummer und Schand“, lautet ein bekanntes Schach-Sprichwort.

2. ♕f1×a6

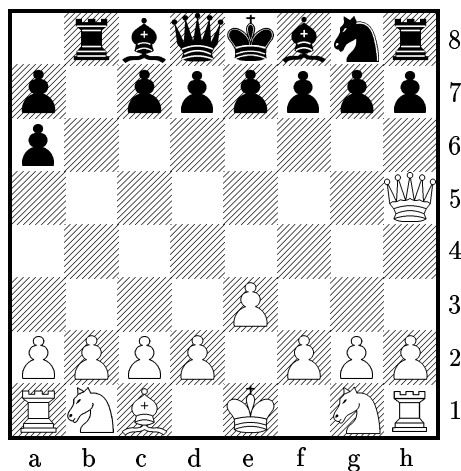
Das weiße Individuum zeichnet sich dadurch aus, dass es alle Tauschmöglichkeiten wahrnimmt, die wenigstens werterhaltend sind. Im Wettkampf mit sehr schlechten Individuen ist dies von Vorteil, denn diese schlagen nicht unbedingt zurück.

2. ... ♞b7×a6

3. ♔d1–h5

Weiß führt die Dame zu früh in die Schlacht. Dies ist ein bekannter Anfängerfehler. Das Problem des vorzeitigen Damenausflugs ist, dass Schwarz die Dame mit weniger wertvollen Steinen angreifen kann, und diese dann fortlaufend zurückweichen muss, wodurch Zeit für andere Operationen verloren geht. Aber auch dieses Verhalten war in der Evolution sicherlich von Vorteil: Die Dame hat durch ihre vielfältigen Zugmöglichkeiten auch viele Schlagmöglichkeiten (siehe auch die Anmerkung zu 2. ♕f1xa6). Ein weitere Motivation für diesen Zug ist, dass Weiß eventuell seinerseits den oben besprochenen Schäferzug einleitet.

3. ... ♖a8–b8



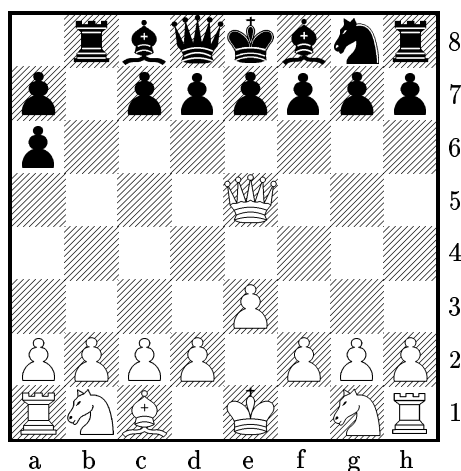
Schwarz stellt seinen Turm auf die offene Linie. Ein vernünftiger Zug.

4. ♔h5–e5 ♜b8–a8

Schwarz ging es mehr darum, den Turm zu ziehen, als um den Besitz der b-Linie. Auch solch ein Zug überrascht den Eingeweihten aber nicht: Wenn einem Individuum kein Zug „einfällt“ (weil das aufgerufene b-Modul nämlich eine leere Zugliste zurückliefert), so wird der lexikographisch erste Zug ausgeführt. Dabei ist zu beachten, dass dem Individuum das Brett immer „aus weißer Sicht“ mit evtl. vertauschten Farben präsentiert wird. Der lexikographischen Ordnung zuliebe hätte 4. ... a6–a5 geschehen müssen. Das hier vorliegende Phänomen ist verwandt mit dem beschriebenen: Die Operation „boardLoop“ in der Stufe b durchläuft das Brett „reihenweise“, und 4. ... ♜b8–a8 wird zuerst betrachtet.

5. ♔e5–d5 ♜a8–b8

6. ♔d5–e5

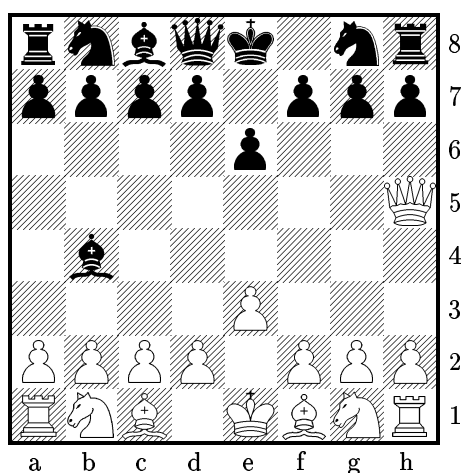


Hier könnte die Partie als unentschieden abgebrochen werden. In einem realen Schachspiel kann die Seite am Zug Remis reklamieren, wenn sie zum dritten Mal in der Partie die gleiche Stellung mit dem gleichen Spieler am Zug herbeiführen kann. Im CHESSGP-Kontext liegt die Sache etwas anders. Dass die Individuen Remis reklamieren, ist nicht vorgesehen. Vielmehr wird die Partie bis zum 70. Zug fortgesetzt und dann mit dem Materialverhältnis als Ergebnis abgebrochen. Da Läufer und Springer als gleichwertig betrachtet werden, befindet sich das Material im Gleichgewicht. Die Partie wird also nach dem 70. Zug mit dem Ergebnis 0.5 : 0.5 abgebrochen.

Die nächste Partie zeigt das 130er-Individuum gegen das beste der 100. Generation. Besonders im Anfangstadium wird eine gewisse Verwandtschaft sichtbar.

Weiß: laufP_R05_130 – Schwarz: laufP_R05_100

1. e2–e3 e7–e6
2. ♖d1–h5 ♠f8–b4

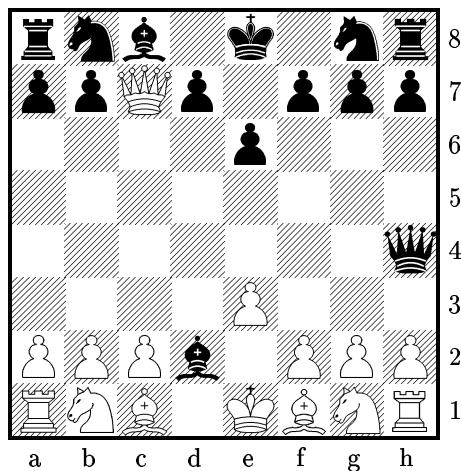


Das Schwarz spielende Individuum hat vermutlich erkannt, dass der „konsequente“ Zug 2. ... ♖d8–h4 am einfachen Schlagen 3. ♖h5×h4 gescheitert wäre.

3. ♖h5–e5 ♜d8–h4

Jetzt, da die weiße Dame das Feld h4 nicht mehr kontrolliert, kann dieser Zug ausgeführt werden.

4. ♜e5×c7 ♙b4×d2



Schwarz verschenkt einfach seinen Läufer und bekommt nur einen Bauern dafür. Außerdem kommt jetzt eine weiße Figur ohne Zeitverlust ins Spiel.

5. ♘b1×d2 ♜h4–g4

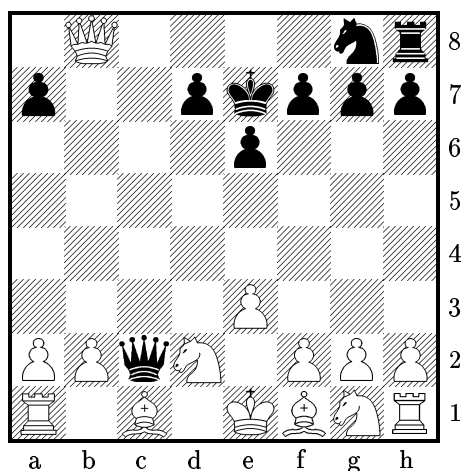
Ein weiterer schwerer Fehler. Jetzt ist der Läufer auf c8 ohne Deckung. Mit 5. ... ♜b8–c6 hätte der Turm auf a8 für diese Aufgabe herangezogen werden können. Vielleicht hat Schwarz aber auch den „Röntgenblick“: Stünden keine Bauern auf e6 und d7, würde die schwarze Dame den Läufer jetzt verteidigen. Vom Operationenset der Stufe a (!) wäre dies theoretisch möglich, aber sehr unwahrscheinlich, weil die Darstellung äußerst komplex wäre.

6. ♜c7×c8 ♔e8–e7

7. ♜c8×b7 ♜g4–a4

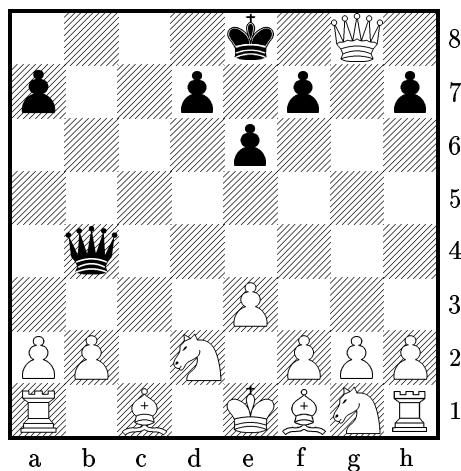
8. ♜b7×a8 ♜a4×c2

9. ♜a8×b8



Die weiße Dame hat im Alleingang den gesamten schwarzen Damenflügel verspeist, und unterdessen hat Schwarz nur ummotiviert mit seiner Dame umhergezogen. Zumindest an der schwarzen „Strategie“ wird sich auch im folgenden Spielverlauf nicht ändern:

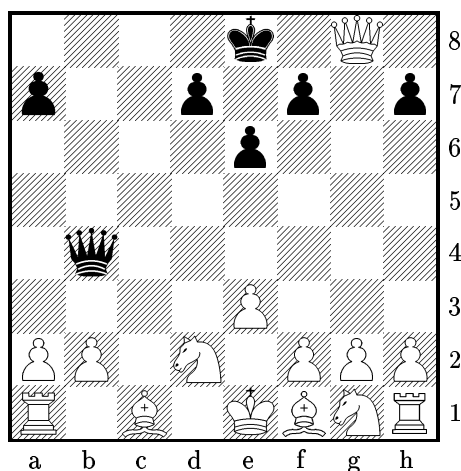
- | | | |
|-----|--------|--------|
| 9. | ... | ♔c2–a4 |
| 10. | ♚b8–e5 | ♚a4–b4 |
| 11. | ♚e5–g5 | ♔e7–e8 |
| 12. | ♚g5×g7 | ♚b4–a4 |
| 13. | ♚g7×h8 | ♚a4–b4 |
| 14. | ♚h8×g8 | |



Weiß hat jetzt auch noch den Königsflügel geplündert und sollte die Partie nun mit einem Matt krönen. Das Mattsetzen in materiell sehr überlegener Stellung ist jedoch eine der Hauptschwächen von CHESSGP-Individuen. Auch in einfachen Endspielen des Typs „mehrere Schwerfiguren gegen ungefähr nackten König“ machen sie einen hilflosen Eindruck, setzen aber bemerkenswert oft durch „Zufall“ matt.

- | | | |
|-----|--------|--------|
| 14. | ... | ♔e8–e7 |
| 15. | ♚g8–g5 | ♔e7–e8 |
| 16. | ♚g5–g8 | |

Weiß bleibt in einem Zyklus hängen, aus dem auch Schwarz nicht ausbrechen kann.



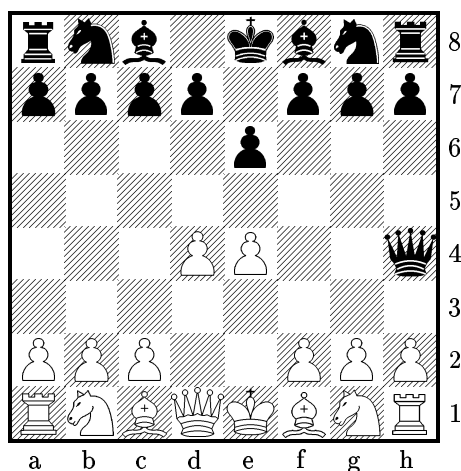
In einer „richtigen“ Schachpartie könnte (und würde!) Schwarz bei einer solchen *Schaukel* das Remis reklamieren. Nicht so jedoch in der CHESSEGP-Welt. Die Partie wird fortgesetzt bis zum 70. Zug und dann mit dem Stand des Materialverhältnisses abgebrochen, was hier dem Resultat 0.73 : 0.27 entspricht. Eine ressourcenschonende Verbesserungsmöglichkeit für die Fitness-Auswertung der Stufe c (siehe Abschnitt 6.1.7) bestünde darin, das Auftreten solcher Zyklen als weiteres Terminierungskriterium zu verwenden.

7.5.5.2 Partien gegen Menschen

Die Spielstärke der gezeigten Individuen reicht aus, um dem Schachlaien interessante Partien zu bieten. Spieler mit profunden Schachkenntnissen bleiben jedoch unbeeindruckt. Im folgenden sei ein Spiel dargestellt, in welchem der mit Weiß spielende menschliche Schachexperte die Schwächen des GP-Individuums konsequent in einen Sieg ummünzt.

Weiß: Martin Villwock (Experte, Elo 2122) – Schwarz: laufP_R05_130

1. e2-e4 e7-e6
2. d2-d4 ♔d8-h4



Der Zug greift einen ungedeckten Bauern an und erhöht die Aktivität der Dame. Trotzdem überwiegen aber die Nachteile des frühen Damenzugs. Nicht umsonst lautet eine bekannte Anfängerregel: „Ziehe nicht zu früh mit der Dame!“. Eine gute Möglichkeit wäre 2. ... d7–d5 gewesen. Die dann entstehende Stellung ist bekannt als Ausgangsstellung der *Französischen Verteidigung*, die auch von vielen Großmeistern angewendet wird.

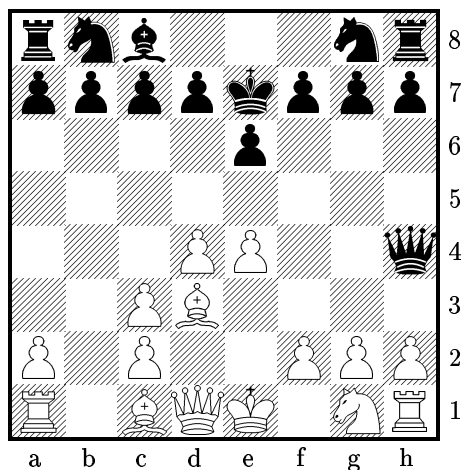
3. ♖b1–c3 ♙f8–b4

Erneuert die Drohung gegen den Bauern e4, weil der Springer an den König gefesselt wird.

4. ♙f1–d3 ♙b4×c3

Schlag- und Schachzüge werden bevorzugt. Schwache Gegner im Evolutionsprozess hätten jetzt vielleicht 5. ♖e1–e2 entgegnet.

5. b2×c3 ♔e8–e7



Dieser Königszug ist eine Frucht der schlechten b-Module. Schon jetzt sollte die schwarze Partie objektiv verloren sein! Der schwarze König darf nach diesem Zug nicht mehr rochieren, und wird in der Brettmitte ein dankbares, statisches Ziel abgeben.

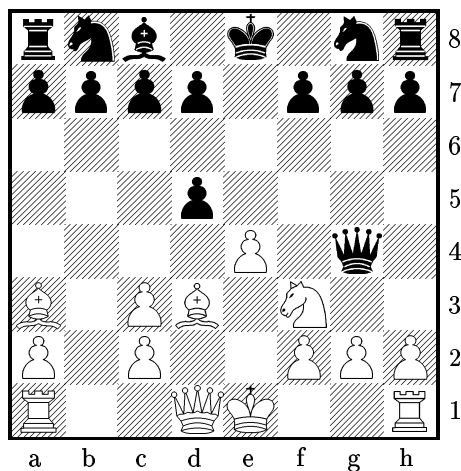
6. ♙c1–a3 ♔e7–e8

7. ♖g1–f3

Schon muß die Dame erneut ziehen.

7. ... ♙h4–g4

8. d4–d5 e6×d5



Schlagzüge werden bevorzugt berücksichtigt. Der Zug ist schlecht, weil die Deckung des schwarzen Königs gelockert wird. Insbesondere kann nach dem Zurückschlagen des weißen Bauern auf e4 Weiß mit der Dame oder einem der Türme entlang der e-Linie „Schach“ ansagen.

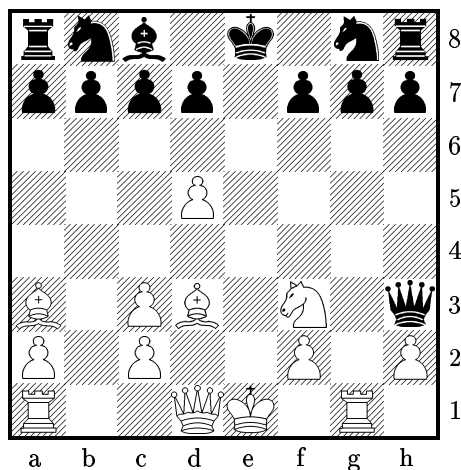
9. e4×d5 ♛g4×g2

Schwarz gewinnt einen Bauern, aber dieser wird ihm im Halse stecken bleiben. Durch das Anbieten von solcherart Bauerngeschenken kann man sämtliche CHESSGP-Individuen leicht auf falsche Fährten locken.

10. ♖h1–g1

Der angegriffene Turm rettet sich und dreht den Spieß um: Jetzt muss die schwarze Dame zum vierten Mal ziehen.

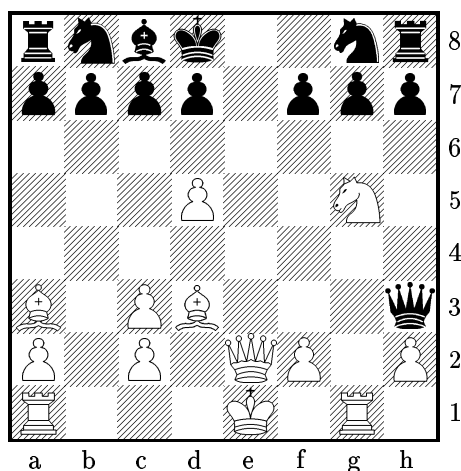
10. ... ♛g2–h3



Zieht man in dieser Position einmal Zwischenbilanz, so fällt ins Auge, dass Weiß schon fast alle seine Figuren *entwickelt* hat, während Schwarz nur seine Dame nach vorne gebracht hat. Weiß steht, obwohl er einen Bauern weniger hat, klar auf Gewinn. Er kann sogar bereits zum finalen Schlag ansetzen:

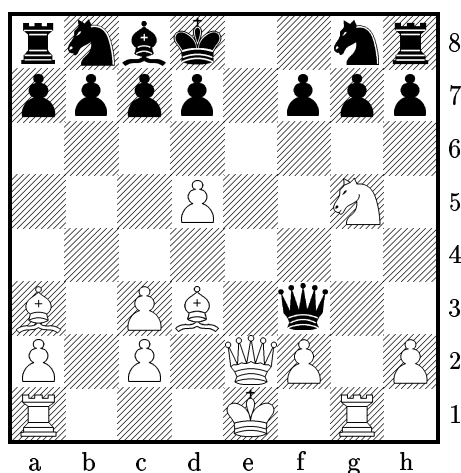
11. ♛d1–e2 ♔e8–d8

12. ♘f3–g5



Schwarz steht vor einer unangenehmen Wahl: Seine Dame ist angegriffen, aber der Springer droht gleichzeitig Matt auf f7.

12. ... ♖h3-f3



Schwarz versucht, zwei Fliegen mit einer Klappe zu schlagen: Zieht die Dame weg und deckt f7. Natürlich kann Weiß die schwarze Dame einfach schlagen. Auch 12. ... ♖h3-h5 oder 12. ... ♖h3-f5 mit der gleichen Idee hätten die Dame nicht gerettet. Der gespielte Zug 12. ... ♖h3-f3 ist ein allgemein für Computerprogramme typischer Zug: Durch den *Horizonteffekt* sieht Schwarz nicht, dass er sowieso mattgesetzt wird. Menschliche Spieler würden in einer solchen, hoffnungslosen Situation eher die direkte Aufgabe oder ein „Racheschach“ vorziehen.

13. ♖e2×f3 ♔d8-e8

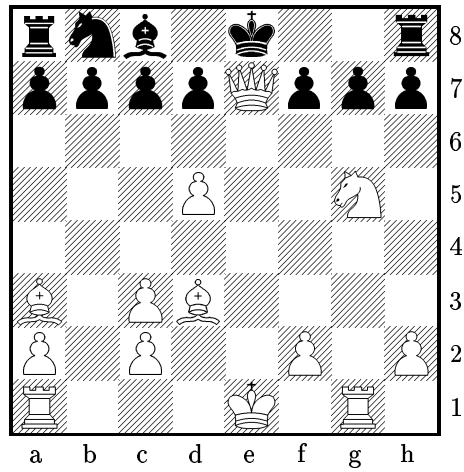
Ein etwas langsames Siechtum hätte 13. ... ♘g8-h6 14. ♘g5×f7+ ♘h6×f7 15. ♚g1×g7 ♚h8-e8+ 16. ♔e1-d2 d7-d6 17. ♖f3-f6+ (17. ♚g7×f7 ♘b8-d7) 17. ... ♚e8-e7 18. ♚g7-g8+ ♔d8-d7 19. ♔d3-f5+ ♚e7-e6 20. ♖f6×f7# bedeutet.

14. ♖f3-e2

Ginge der König jetzt nach d8 zurück, folgte das Springerrett auf f7, wie zuvor.

14. ... ♖g8-e7

15. ♔e2×e7



Schachmatt!

1-0

8 Fazit

In diesem Kapitel soll in einem zusammenfassenden Rückblick festgestellt werden, in welchem Ausmaß das PG-Ziel erreicht worden ist und welche weiteren Anknüpfungsmöglichkeiten an die Projektarbeit sich ergeben.

Ziel der PG war es, mittels Genetischer Programmierung Individuen zu erstellen, die in der Lage sind, Schach zu spielen. Diese Individuen wurden im Rahmen eines Hierarchiekonzeptes erzeugt. Dieses Konzept besteht aus drei Stufen, wobei die in einer Stufe evolvierten Programme in den höheren Stufen jeweils als Module benutzt werden können. Auf der dritten Stufe befindet sich das eigentliche Schachprogramm. Zusätzlich entstanden eine Reihe von Hilfsprogrammen, die sowohl den Umgang mit den Schachfunktionalitäten, als auch den Ablauf der Evolutionsläufe unterstützten.

In der Stufe a wurden Schachstellungsbewertungen evolviert, d. h. die Individuen dieser Stufe sollten dazu in der Lage sein, zu einer beliebigen Schachstellung jeweils eine Bewertung in Form einer Zahl zurückzuliefern. Dabei konnte für die Bewertungsfunktion „Materialbewertung“ ein im Sinne unserer Fitnessfunktion optimales Individuum evolviert werden. Für die anderen Bewertungsfunktionen wurden Individuen erzeugt, die zwar einen gewissen Fehler produzierten, sich aber im Mittel auf dem gesamten Datenbestand der Datenbank gleich verhielten. Nur einige wenige Bewertungsfunktionen zeigten noch größere Abweichungen in speziellen Situationen.

Die Evolution der Stufe b bereitete größere Schwierigkeiten. Letztlich war es uns nicht möglich, Individuen zu erzeugen, die Zugvorschläge in Form von Zuglisten in der erwarteten Qualität zurückgeben. Bei den Zuglisten unserer Individuen war kein Lernen der Datenbankvorgaben zu erkennen. Die Gründe hierfür sind unklar und noch näher zu untersuchen. Erklärungen können auch in der von uns gewählten Vorgehensweise liegen, in dieser Stufe im Spielbaum vorausschauende Bewertungsfunktionen erstellen zu wollen, ohne jedoch Operationen mit Vorausschaumöglichkeiten zur Verfügung zu stellen. Motiviert wurde diese Idee durch psychologische Untersuchungen menschlicher Schachexperten: Auch diese ermitteln ihre Züge nicht durch Abarbeitung eines Algorithmus, der etwa mit Minimax oder ähnlichem zu vergleichen wäre. Stattdessen werden ganze Brettstellungen aufgrund der vorhandenen Situation (ohne Vorausschau) eingeschätzt, ohne dass diese Schachspieler selber angeben könnten, wie sie diese Bewertung vorgenommen haben. Dies hatten wir versucht durch einen Ansatz nachzuahmen, bei dem ebenfalls nicht von Anfang an klar war, ob und wie ein GP-Algorithmus die Aufgabe lösen kann.

Da sinnvolle Individuen der Stufe b nicht evolviert werden konnten, fertigten wir einige „künstliche“ Individuen selber an. Durch unser Vorgehen, nur Operationen aus dem Sprachumfang der Stufe b zu erstellen, spricht trotz der misslungenen Evolution in dieser Stufe nichts gegen die prinzipielle Möglichkeit, dass solche Individuen per Evolution erzeugt werden können.

Bei der Stufe c handelt es sich um die eigentliche Spielstufe. Hier wird von einem Individuum, wie in anderen Schachprogrammen auch, ein Spielbaum aufgespannt, um einen Zug auszuwählen. Die Individuen bestimmen einerseits die

Spielbaumstruktur, andererseits legen sie fest, was innerhalb der Knoten des Spielbaumes berechnet werden soll. Die Spiele der Individuen gegeneinander erwiesen sich als sehr rechenzeitintensiv, weswegen eine parallele Architektur realisiert wurde.

Das wichtigste Ergebnis der Projektgruppe sind die GP-Individuen, die Schach spielen. Die entstandenen Individuen wurden im Laufe der Evolution nachweislich spielstärker und konnten einige unserer selbstimplementierten herkömmlichen Schachprogramme bezwingen. Hin und wieder besiegten sie auch Schachläien, welche insgesamt von der Spielstärke mancher Individuen – im Kontext zu deren Erstehung – beeindruckt waren. Experten des Schachspiels bleiben allerdings völlig unbeeindruckt. Ebenso ist die Spielstärke der Individuen sehr weit von der existierender Computerschachprogramme entfernt. Allerdings sollte die Kürze der uns zur Verfügung gestandenen Rechenzeit beachtet werden. Dass mit Genetischer Programmierung im Allgemeinen und unserem Stufenkonzept im Besonderen Schachprogramme gelernt werden können, konnte also gezeigt werden. Aufgrund der Tatsache, dass im Bereich der Logikanwendungen evolutionäre Ansätze noch selten sind, konnte hiermit ein neues Vorgehen für ein altbekanntes – und von anderen Ansätzen bereits recht gut gelöstes – Problem vorgestellt werden.

Desweiteren sollten auch die „Nebenprodukte“ der Projektarbeit nicht vernachlässigt werden: Das hierarchische Stufenkonzept aus wiederverwendbaren GP-Modulen half bei der Komplexitätsreduktion des Problems und führte dadurch auch zu schneller verfügbaren Teilergebnissen. Die Vorgehensweise des Restarts während der Evolutionsläufe automatisierte den Neustart von Läufen bei Stagnation der Fitnesswertentwicklung. Die Parallelisierung machte die Evolutionsläufe überhaupt erst möglich und die Anbindung an eine GUI ermöglicht eine bequeme Benutzung. Einige dieser Vorgehensweisen lassen sich auch auf andere mit GP zu lösende Aufgaben übertragen und nutzbar machen.

Ansätze für weitere Arbeiten an CHESSGP gibt es mehrere: es wäre wichtig, herauszufinden, warum die Evolution der Stufe b nicht funktionierte und wie dies erreicht werden kann. Eine intensivere Analyse der Ergebnisse auf allen Stufen kann zu Erkenntnissen über mögliche Parameterverbesserungen und somit zu besseren Ergebnissen führen. Neue Läufe mit Stufe c sollten mehr Rechenzeit zur Verfügung gestellt bekommen, ebenso wäre der Test verschiedener Einstellungen sinnvoll. Auch die Untersuchung der zur Verfügung gestellten Operationen und deren Problemlösekapazität wäre eine der anstehenden Aufgaben. Weiterhin wäre es interessant, grundsätzlich andere Konzepte zu implementieren und deren Leistungsfähigkeit mit den Ergebnissen unseres Stufenkonzeptes zu vergleichen. Denkbar wären beispielsweise Ansätze, die keine Trennung in Stufen vornehmen oder bei denen die Funktionalität aus der Perspektive der einzelnen Figuren gelernt wird.

Als Abschlussfazit bleibt festzuhalten, dass es der Projektgruppe 348 gelungen ist, mit Genetischer Programmierung Schach spielende Programme zu erzeugen. Die nur unbefriedigend gelösten Teilaufgaben und die Verbesserung der Spielstärke bieten Anlass zu Folgearbeiten.

Literatur

- [BAN94] BANZHAF, W.: Genotype-Phenotype-Mapping and Neutral Variation — A Case Study in Genetic Programming. In: *Lecture Notes in Computer Science* 866 (1994). – ISSN 0302-9743
- [BB99] BRAMEIER, M. ; BANZHAF, W.: A Comparison of Linear Genetic Programming and Neural Networks in Medical Data Mining. In: *IEEE Transactions on Evolutionary Computation* (1999). – (eingereicht)
- [BIS95] BISKUP, Joachim: *Grundlagen von Informationssystemen*. Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, 1995
- [BLE98] DE BLECOURT, Sandra: The Legacy of Arpad Elo - The Development of a Chess-Rating System / Universiteit van Amsterdam, Faculteit der Psychologie, VRT-2. 1998. – Forschungsbericht
- [BNKF98] BANZHAF, W. ; NORDIN, J.P. ; KELLER, R.E. ; FRANCONI, F.D.: *Genetic Programming - An Introduction; On Automatic Evolution of Computer Programs and its Applications*. dpunkt-Verlag, Heidelberg. Morgan Kaufmann, San Francisco, 1998. – ISBN 3-920993-58-6
- [DAR59] DARWIN, C.: *On the Origin of Species by Means of Natural Selection*. Murray, London, 1859
- [DEG65] DEGROOT, A.: *Thought and Choice in Chess*. Den Haag : Mouton, 1965
- [ELO78] ELO, Arpad: *The rating of chess players past and present*. Arco Publishing, New York, 1978
- [FCBN99] FRANCONI, Frank D. ; CONRADS, Markus ; BANZHAF, Wolfgang ; NORDIN, Peter: Homologous Crossover in Genetic Programming. In: BANZHAF, Wolfgang (Hrsg.) ; DAIDA, Jason (Hrsg.) ; EIBEN, Agoston E. (Hrsg.) ; GARZON, Max H. (Hrsg.) ; HONAVAR, Vasant (Hrsg.) ; JAKIELA, Mark (Hrsg.) ; SMITH, Robert E. (Hrsg.): *Proceedings of the Genetic and Evolutionary Computation Conference* Bd. 2. Orlando, Florida, USA : Morgan Kaufmann, 1999
- [FDN59] FRIEDBERG, R. ; DUNHAM, B. ; NORTH, T.: A learning machine: Part II. In: *IBM Journal of Research and Development* 3 (1959), Nr. 3, S. 282-287
- [FRI58] FRIEDBERG, R. M.: A learning machine: Part I. In: *IBM Journal* 2 (1958), S. 2-13
- [FÜR96] FÜRNKRANZ, J.: Machine Learning in Computer Chess: The Next Generation. In: *ICCA Journal* 19 (1996), Nr. 3, S. 147-161. – <ftp://ftp.ai.univie.ac.at/papers/oefai-tr-96-11.ps.gz>

- [GOL89] GOLDBERG, D.E.: *Genetic Algorithms in Search, Optimization & Machine Learning*. Reading : Addison-Wesley Publishing Company, Inc., 1989
- [GÖR95] GÖRZ, G.: *Einführung in die künstliche Intelligenz*. Bonn : Addison-Wesley, 1995
- [HOF85] HOFSTADTER, D. R.: *Gödel, Escher, Bach: ein Endloses Geflochtenes Band*. Stuttgart : Klett-Cotta, 1985
- [HOL75] HOLLAND, J.H.: *Adaptation in natural and artificial Systems*. Univ. of Michigan, 1975
- [JAC97] JACOB, C.: *Principia Evolvica - Simulierte Evolution mit Mathematica*. Heidelberg : dpunkt, 1997
- [JKA83] JUDOWITSCH, Michail ; KOTOW, Alexander A. ; AWERBACH, Juri L.: *Das Schachbuch für Meister von Morgen: Ein Lehr- und Trainingswerk - nicht nur für den Nachwuchs*. 2.Aufl. 1989. Joachim Beyer-Verlag, Hollfeld, 1983. – ISBN 3-88805-050-2
- [JW98] JANSEN, T. ; WEGENER, I.: On The Analysis Of Evolutionary Algorithms - A Proof That Crossover Really Can Help / Department of Computer Science, University of Dortmund. 1998 (CI-51/98). – Forschungsbericht
- [KABK97] KOZA, J.R. ; ANDRE, D. ; BENNET, F.H. ; KEANE, M.A.: *Genetic Programming 3: hardware Automatic Programming and Automatic Circuit Synthesis*. MIT Press, 1997
- [KAI90] KAINDL, H.: Tree Searching Algorithms. In: MARSLAND, T. A. (Hrsg.) ; SCHAEFFER, J. (Hrsg.): *Computers, Chess, and Cognition*. New York : Springer-Verlag, 1990, S. 133–158
- [KAU93] KAUFFMAN, S.: *The origins of order*. Oxford University Press, 1993
- [KOZ92] KOZA, J.: *Genetic Programming: On the Programming of Computers by the Means of Natural Selection*. Cambridge, MA : The MIT Press, 1992
- [KOZ94] KOZA, J.: *Genetic Programming II*. Cambridge, MA : MIT Press, 1994
- [KÜM92] KÜMMERLIN, Dorothee: *Genetische Algorithmen zur Bestimmung heuristischer Bewertungsfunktionen bei Spielen*. ftp://ftp.informatik.uni-freiburg.de/documents/masterThesis/dorotheeKue_mmerlin.ps.gz, Albert-Ludwigs-Universität Freiburg/Br., Diplomarbeit, 1992
- [LEV89] LEVINSON, R.A.: A Self-Learning, Pattern-Oriented Chess Program. In: *ICCA Journal* 12 (1989), Nr. 4, S. 207–215

- [MAT96] MATANOVIC, Aleksandar (Hrsg.): *640 best 64 golden games*. Sahovski Informator, Beograd, 1996
- [MEN97] MENKE, R.: A Revision of the Schema Theorem / Department of Computer Science, University of Dortmund. 1997 (ISSN 1433-332). – Forschungsbericht. Series Computational Intelligence
- [MIT97] MITCHELL, T. M.: *Machine learning*. New York : McGraw-Hill, 1997
- [MON93] MONTANA, David J.: Strongly Typed Genetic Programming / Bolt Beranek and Newman, Inc. 1993 (#7866). – BBN Technical Report
- [MS90] MARSLAND, T.A. (Hrsg.) ; SCHAEFFER, J. (Hrsg.): *Computers, Chess, and Cognition*. Springer-Verlag, New York, 1990. – ISBN 0-387-97415-6
- [MUG90] MUGGLETON, S.: *Inductive Acquisition of Expert Knowledge*. Turing Institute Press, Addison-Wesley, 1990
- [OO94] O'REILLY, Una-May ; OPPACHER, Franz: Using Building Block Functions to Investigate a Building Block Hypothesis for Genetic Programming / Santa Fe Institute. Santa Fe, New Mexico. USA, 1994 (94-02-029). – Working Paper
- [OO95] O'REILLY, Una-May ; OPPACHER, Franz: The Troubling Aspects of a Building Block Hypothesis for Genetic Programming. In: WHITLEY, L. D. (Hrsg.) ; VOSE, Michael D. (Hrsg.): *Foundations of Genetic Algorithms 3*. Estes Park, Colorado, USA : Morgan Kaufmann, 1995, S. 73–88
- [ORA99] ORACLE: Oracle8 Server. In: *Oracle-Docus* (1999)
- [PEA80] PEARL, J.: Scout: A simple game-searching algorithm with proven optimal properties. In: *First Annual National Conference on Artificial Intelligence*, 1980
- [PET90] PETKOVIC, Dusan: *SQL- Die Datenbanksprache*. McGraw-Hill Book Company GmbH, Hamburg, 1990
- [QUI83] QUINLAN, J. R.: Learning efficient classification procedures. In: CARBONELL, J. G. (Hrsg.) ; MITCHELL, T. M. (Hrsg.): *Machine Learning: An Artificial Intelligence Approach*. Palo Alto : Tioga, 1983, S. 463–482
- [QUI90] QUINLAN, J. R.: Induction of Decision Trees. In: SHAVLIK, Jude W. (Hrsg.) ; DIETTERICH, Thomas G. (Hrsg.): *Readings in Machine Learning*. Morgan Kaufmann, 1990. – Originally published in *Machine Learning* 1:81–106, 1986
- [REC73] RECHENBERG, Ingo: *problemata*. Bd. 15: *Evolutionsstrategie*. Stuttgart : Friedrich Frommann Verlag (Günther Holzboog KG), 1973

- [REI83] REINEFELD, A.: An improvement of the Scout tree search algorithm. In: *ICCA Journal* 6 (1983), Nr. 4, S. 4–14
- [REI89] REINEFELD, A.: *Spielbaum-Suchverfahren*. Springer-Verlag, 1989
- [SAM67] SAMUEL, A.L.: Some Studies in Machine Learning Using the Game of Checkers. II-Recent Progress. In: *IBM Journal of Research and Development* Vol. 11, No. 6 (1967), S. 601–617
- [SCH95] SCHWEFEL, Hans-Paul: *Evolution and Optimum Seeking*. New York : John Wiley & Sons, Inc., 1995 (Sixth-Generation Computer Technology Series)
- [SCH97] SCHNEIDER, H.-J.: *Lexikon der Informatik und Datenverarbeitung - Version 4.0*. München, Wien : R. Oldenbourg Verlag, 1997
- [SF95] STEINWENDER, D. ; FRIEDEL, F.A.: *Schach am PC*. Markt & Technik, 1995
- [SKI86] SKIENA, S. S.: An overview of machine learning in computer chess. In: *ICCA Journal* 9 (1986), Nr. 1, S. 20–28
- [STR96] STROUHAL, Ernst: *Acht mal acht: zur Kunst des Schachspiels*. Springer-Verlag, Wien, New York, 1996. – ISBN 3-211-82775-7
- [TAC94] TACKETT, Walter A.: *Recombination, Selection, and the Genetic Construction of Computer Programs*, University of Southern California, Department of Electrical Engineering Systems, Diplomarbeit, 1994
- [TAD89] TADEPALLI, P.: Planning in games using approximately learned macros. In: *Proceedings of the Sixth International Workshop on Machine Learning, Ithaca, NY*, Morgan Kaufmann, 1989, S. 221–223
- [TAN89] TANESE, Reiko: Distributed Genetic Algorithms. In: SCHAFER, J. D. (Hrsg.): *Proceedings of the 3rd International Conference on Genetic Algorithms*. George Mason University : Morgan Kaufmann, 1989
- [TES92] TESAURO, G.J.: Practical issues in temporal difference learning. In: *Machine Learning* 8 (1992)
- [THR95] THRUN, S.: Learning To Play the Game of Chess. In: *Advances in Neural Information Processing Systems* 7 (1995)
- [THU94] THURSTONE, L.L.: A Law of Comparative Judgement. In: *Psychological Review* (1994), Nr. 101, S. 266–270. – Original work published 1927
- [TIG91] TIGGELEN, A. v.: Neural Networks as a Guide to Optimization. In: *ICCA Journal* 14 (1991), Nr. 3, S. 115–118

- [TP91] TUNSTALL-PEDOE, W.: Genetic Algorithms Optimizing Evaluation Functions. In: *ICCA Journal* 14 (1991), Nr. 3, S. 119–128
- [TV95] TELLER, Astro ; VELOSO, Manuela: PADO: Learning Tree Structured Algorithms for Orchestration into an Object Recognition System / Department of Computer Science, Carnegie Mellon University. Pittsburgh, PA, USA, 1995 (CMU-CS-95-101). – Forschungsbericht
- [VOS94] VOSSEN, Gottfried: *Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme*. 2. aktualisierte und erweiterte Auflage. Addison-Wesley (Deutschland) GmbH, 1994
- [WEG00] WEGENER, Ingo: *Branching programs and binary decision diagrams - theory and applications*. SIAM Monographs in Discrete Mathematics and Applications, 2000