## Genetische Algorithmen in Denkspielen

**Wissenschaftlich ausgedrückt sind genetische Algorithmen nur eine Methode numerischer Optimierung. Praktisch gesprochen eignen sie sich, um in einem System mit extrem vielen Parametern, die analytisch nicht mehr zu bändigen sind, die optimalen Wichtungen zu finden. Die Faszination liegt darin, dass das quasi automatisch geschieht, per Evolution: der Programmierer erzeugt in seinem Computer zahllose simulierte Lebewesen, die auf Partnersuche sind und deren Chancen auf Fortpflanzung umso besser stehen, je besser sie das gestellte Problem lösen. Dann geht es mit den Kinderchen dieser Ur-Wesen in die nächste Runde ... nach wenien Generationen entstehen so Wesen, die ausgezeichnete Problemlösefähigkeiten haben. Wer das für esoterisch hält, möge sich durch die praktischen Erfolge eines Besseren belehren lassen: Genetische Algorithmen fanden ein besseres Sortierverfahren als Informatik-Guru Donald Knuth, optimierten Düsenformen und erstellen robuste FPGA-Schaltungen! Wäre diese Art Optimierung nicht auch etwas für die Bewertungsfunktion von Denkspielen, zum Beispiel Schachprogrammen, in denen es genau darum geht, viele hundert Parameter im Zusammenhang zu gewichten?**

Schach-Engines bieten zahllose Parameter, ihnen ins Hirn zu greifen, angefangen bei simplen Materialwerten über subtilere Freibauern-Gewichtung bis hin zu magischen Sammelparametern wie der Selektivität. Hardcore-Anwender experimentieren gern damit herum, um noch ein paar Elo-Pünktchen herauszukitzeln, legendär sind hier die vielen Spielstile der King-Engine. Programmierer haben dasselbe Problem in Potenz, denn die meisten Wichtungen innerhalb einer Bewertungsfunktion sieht der Anwender gar nicht, weil bereits vom Entwickler festgelegt wurde, welches positionelle Kriterium wie viele Hunderstel Bauern wert sein soll. Typischerweise probiert der Programmierer irgendwas, das sich für ihn subjektiv gut anfühlt, und dann lässt er es testen in vielen schnellen Partien. Schneidet die Engine besser ab, ist es gut, fertig, aus. Es mag sein, dass vielversprechende Ideen nur deshalb nicht funktionieren, weil sie sich mit einer anderen verwendeten Technik beißen, oder weil sie geringfügig falsch gewichtet sind (es geht um Hundertstel Bauern!). Die Optimierung per Hand und nach Gefühl mag funktionieren, vielleicht aber tut sie es nur, weil noch keiner andere Methoden probiert hat? Wie spannend wäre eine Technik, die all diese vielen Parameter einmal im Zusammenhang optimal einstellt!

In anderen Spielen finden Methoden der numerischen Optimierung durchaus erfolgreiche Anwendung. So beruht das im Go momentan so erfolgreiche UCT auf der Monte-Carlo-Methode, die direkt aus der Mathematik stammt! Auch die neuronalen Netze, die so stark Backgammon spielen, stellen letztlich nichts anderes als eine hochoptimierte Bewertungsfunktion dar. Und werden die Netze etwa von Hand optimiert? Mitnichten, sondern mit einem automatischen Verfahren (Backpropagation), das ebenfalls unter numerische Optimierung fällt!

Sogar genetische Algorithmen wurden schon ausprobiert, allerdings erst ein Mal, da allerdings mit Erfolg. Und mit einer Art Erfolg, die nahelegt, diese Methode könnte auch komplexe Bewertungsfunktionen optimieren. Um zu verstehen, warum das so ist, sollte man ein bisschen tiefer einsteigen in die Welt der digitalen Evolution, in der Programmierer nicht nur Gott spielen, sondern wirklich Götter sind!

## Evolution im Computer

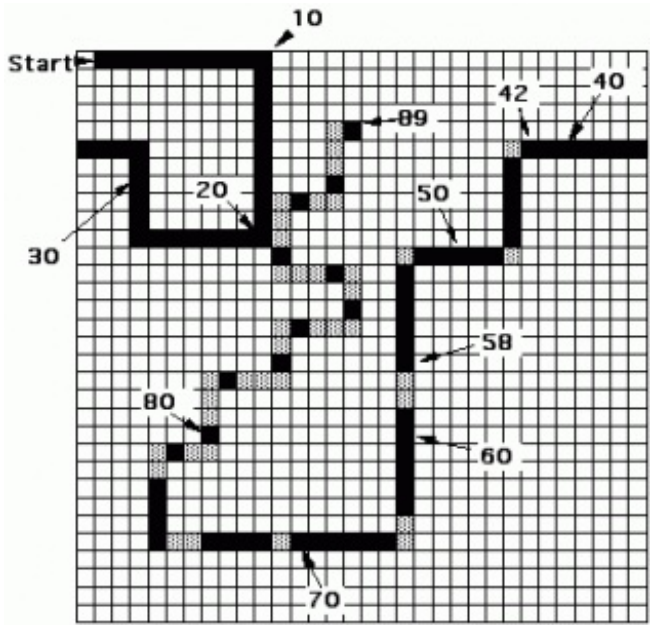
Die Existenz der Evolution in der weiten Welt dort draußen wird von den meisten Leuten anerkannt, sieht man einmal von religiösen Fanatikern und anderen Dummköpfen ab. Und so, wie man analoge Signale digitalisieren kann, um sie im Rechner zu verarbeiten, funktioniert das auch mit der Evolution, auch die kann man im Computer, nein, eben nicht simulieren, sondern *ablaufen lassen* und dabei beobachten. Dafür braucht man eine Umwelt und künstliche Lebewesen. Die Umwelt besteht ganz einfach aus dem zu lösenden Problem. Die Lebewesen brauchen also irgend eine Art Funktion, in die man das Problem einfüllt, es verarbeitet und aus der eine Lösung herausfällt. Die Parameter dieser Funktion bestehen aus den Genen. Im einfachsten Fall könnte das eine simple lineare Funktion sein, y = a * x, wobei x der Eingangswert, a das hier einzige Gen und y die Lösung wäre.

Als nächstes erzeugt ein Programm die Startpopulation, also beispielsweise ein paar tausend Lebewesen mit zufälligen Genen. Jedes dieser Lebewesen wird nun auf das Problem losgelassen, und wegen der unterschiedlichen Gene löst jedes es anders. Dann schlägt die Evolution zu. Die Lebewesen werden nach der Qualität ihrer Lösung sortiert, und die, die dichter an der Lösung waren als andere, dürfen sich fortpflanzen. Das kann, wie in der Natur, auf sexuelle und auf asexuelle Weise geschehen. Asexuelle Fortpflanzung beruht auf Zellteilung; der Organismus wird einfach dupliziert, wobei aber mit einer gewissen Wahrscheinlichkeit eine Mutation eintritt, also in einem zufällig bestimmten Gen ein Bit umkippt. Bei der sexuellen Fortpflanzung werden zwei Organismen ausgewählt und ihre Gene kombiniert. Dafür bestimmt wieder der Zufall in jedem Gen ein Bit, an dem die Gene der Eltern aufgesplittet werden. Das betreffende Gen des Kind-Organismus‘ entsteht durch Rekombination; der untere Teil stammt vom Vater-Organismus, der obere vom Mutter-Organismus. Das Verfahren heisst Crossing Over.

Nachdem auf diese Weise eine neue Generation von Organismen entstanden ist, wird wieder von jedem Organismus die Fitness bestimmt und das Spiel wiederholt sich. Allmählich nimmt die Fitness der Population zu, solange, bis sie allmählich an einem bestimmten Maximum konvergiert. Das muss nicht die beste erreichbare Lösung sein; üblicherweise, bei Beachtung bestimmter Regeln, wird die Lösung aber dicht am Optimum liegen.

## Evolutionsgeschichte

Genetische Algorithmen wurden 1962 von John Henry Holland zuerst beschrieben. Holland, der zu jener Zeit am MIT lehrte, ließ sich dabei von einem Buch des Evolutionsbiologen R. A. Fisher inspirieren. Die erste praktische Umsetzung dieser Ideen erfolgte durch David Jefferson an der University of California in Los Angeles. Jefferson legte auf einem torusförmigen Netz aus 32x32 Feldern einen Pfad von 89 Feldern Länge an, der teilweise unterbrochen war und verschiedenste Richtungsänderungen aufwies. Er ließ „künstliche Ameisen“ diese „Pheromonspur“ verfolgen, wobei diese Ameisen aus je 450 Bits bestanden, die im Zusammenhang mit der Umwelt das Verhalten des Organismus‘ für die nächste Zeiteinheit festlegten. Die Ameise erhielt als Umwelt-Information dabei nur den Zustand des Feldes direkt vor dem eigenen Feld. Nachdem die Ameise diesen Zustand und ihren eigenen überprüft hatte, konnte sie zwischen verschiedenen Reaktionen wählen: sich nach links oder rechts drehen, weitergehen oder überhaupt nichts tun.

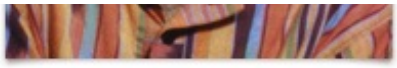


Der Pfad, dem die genetischen Ameisen zu folgen hatten ...

Dann erzeugte Jefferson 65536 dieser Ameisen mit rein zufälliger Bitfolge und ließ sie auf dem Pfad loslaufen. Die meisten taten natürlich nichts, drehten Pirouetten oder liefen irgendwohin. Manche waren aber in der Lage, dem Pfad einige wenige Schritte zu folgen, und diese wurden zur Fortpflanzung ausgewählt. Dabei wurden die Bitketten zweier Organismen an zufällig bestimmter Stelle aufgetrennt und die Teile miteinander zu einer neuen Ameise kombiniert. Nach nur 20 Generationen konnte eine durchschnittliche Ameise aus dem Mittelfeld der Population bereits 30 Felder weit dem Pfad folgen, während die besten Ameisen über 60 Felder schafften. Nach 70 Generationen konnten die meisten Ameisen dem Pfad bis zum Ende folgen, der König der Pfad-Ameisen absolvierte den Weg, als habe ein guter Programmierer ihn entworfen. Die Ergebnisse dieser Arbeit sind hier nachzulesen.

Nach dieser ersten Demonstration der Leistungsfähigkeit genetischer Algorithmen folgte eine Phase, in der viele Experimente mit genetischen Algorithmen und evolutionärer Programmierung durchgeführt wurden. 1972 setzten zwei Ingenieure, Hans-Paul Schwefel und Ingo Rechenberg, evolutionäre Programmierung ein, um Düsenformen zu optimieren. Zu jener Zeit erreichte das beste mathematische Modell einen Wirkungsgrad von 55 Prozent. Schwefel und Rechenberg unterteilten eine Düsenform in viele Ringe unterschiedlichen Durchmessers. Durch Austausch der Ringe änderte sich die Düsenform; die Gesamtanzahl der möglichen Kombinationen lag bei 1060. Sie konnten den Wirkungsgrad auf beinahe 80 Prozent steigern.

1986 setzte Danny Hillis genetische Algorithmen ein, um einen Sortieralgorithmus zu finden. Eine Kette aus zwölf Zahlen sollte der Größe nach sortiert werden, mit möglichst wenig Austausch-Operationen, die zudem vorher feststehen müssen. Das Problem hatte eine Reihe von Mathematikern und Informatikern schon länger beschäftigt, weil für die Sortierung in Hardware keine Software-Sortier-Algorithmen wie Quicksort in Frage kommen, weil die Austausch-Operationen da vom Ergebnis vorheriger Operationen abhängen. 1962 erschien ein Algorithmus, der mit 65 Austausch-Operationen auskam und dessen Autoren in Anspruch nahmen, die beste Lösung gefunden zu haben. 1964 verbesserte der Guru aller Informatiker, Donald Knuth das Verfahren um zwei Schritte auf 63, 1969 erschien ein Algorithmus, der mit 62 Austausch-Operationen auskam, und 1970 veröffentlichte Milton Green eine Lösung mit nur 60 Schritten. Hillis‘ genetische Sortierprogramme fanden ein Verfahren mit 62 Schritten, waren also immerhin erfolgreicher als Donald Knuth – ein Ergebnis, das die meisten Programmierer auf diesem Planeten zu einem Freudenbesäufnis veranlassen dürfte.



Informatik-Guru Donald Knuth (The Art of Computer Programming, LaTeX)

Die im Kontext der Elektrotechnik bedeutsamste Anwendung fanden genetische Algorithmen beim Entwurf von Schaltungen. 1994 begann Adrian Thompson, FPGAs genetisch programmieren zu lassen. Das erste Resultat von Thompsons Arbeit war eine Schaltung, welche die Wörter „stop“ und „go“ unterscheiden konnte und mit nur 37 Gattern eines 10x10-Arrays auskam. Neben der hohen Effizienz zeichnet genetisch entwickelte Schaltungen eine weitgehende Unempfindlichkeit gegen Hardware-Fehler aus, denn nach Thompsons Ergebnissen verhält sich ein über tausende Generationen entwickeltes Konfigurationsprogramm recht stoisch gegenüber Mutationen – Versuche zeigten, daß die Zerstörung eines Gatters im laufenden Betrieb unter günstigen Umständen der Schaltung wie eine Mutation erscheinen, an die sie angepaßt ist und die sie kompensieren kann. Die Störungsunanfälligkeit ist insbesondere bezüglich der Temperaturtoleranz bedeutsam, denn Thompson trainiert seine Schaltungen in einem weiten Temperaturbereich, indem er Kühl- und Heizelemente unter den FPGAs anbringt.

## Genetische Algorithmen in Denkspielen

Angesichts dieser Resultate verwundert es fast, dass noch niemand versucht hat, die Bewertungsfunktion komplexer Spielprogramme mit genetischen Algorithmen zu optimieren. Das einzige Beispiel betrifft das simple TicTacToe, es ist allerdings interessant, weil es die Unterschiede zwischen genetischer Optimierung mit vollständiger und unvollständiger Information zeig, also den Vergleich zwischen Verfahren, die versuchen, zu einer Datenbank aller Stellungen die zugehörigen Stellungsbewertungen zu lernen und Verfahren, die TicTacToe dadurch erlernen, daß sie es einfach spielen.

Die Bewertungsfunktion bestand einfach aus drei zufällig zusammengewürfelten goniometrischen Funktionen, Sinus, Cosinus und Sigmoid. Die Parameter dieser Funktionen wurden durch die Spielfelder sowie die Gene der Organismen befüllt, wobei weitere Gene entscheiden konnten, welche Funktionen überhaupt verwendet wurden. Dieser primitive Ansatz wurde auf das TicTacToe-Spiel losgelassen .

Beim Antizipieren einer Datenbank scheiterte der genetische Algorithmus grandios, trotz des getriebenen ziemlich hohen Aufwandes. Zwar konnte letztendlich durch eine Kombination zweier Organismen ein perfekt spielendes Programm geschaffen werden, doch muß es immer anfangen, und immer nur mit einem vorgegebenen Zug. Zudem benötigt es eine Taktik-Kontrolle auf sofortigen Gewinn oder Verlust.

Besteht man aber nicht darauf, daß *jede* Stellung korrekt bewertet werden möge (ein Ziel, das zu erreichen bei komplexeren Problemen als TicTacToe ohnehin kaum realisierbar ist), sondern gibt dem genetischen Algorithmus das Ziel des Projektes – zu spielen, nie zu verlieren und so oft wie möglich zu gewinnen – direkt mit auf den Weg, ohne den Umweg über obskure Datenbanken, entsteht mit größter Leichtigkeit eine perfekte Bewertung. Zwar nicht jeder Stellung, aber jeder *relevanten* Stellung – jeder Stellung, die dem geschaffenen Organismus jemals unterkommen kann, wenn er eine Partie spielt. Solch ein Organismus, der als Anziehender niemals verliert, entsteht bereits nach wenigen Generationen einer Population, deren Fitness-Parameter nur darin besteht, fünftausend Partien gegen einen Zufallsgenerator zu spielen. Wenn die Fitness-Funktion nicht nur auf das Vermeiden von Niederlagen optimiert (was natürlich am wichtigsten ist!), sondern auch Siege höher als Remisen wichtet, entsteht ein Organismus, der gegen einen Zufallszüge machenden Gegner prozentual besser abschneidet als eine Datenbank oder ein Baumsuch-Programm, denn die Organismen werden gewissermaßen darauf trainiert oder besser: dafür gezüchtet, Stellungen aufzubauen, in denen der Pfad zum Unentschieden für den Gegner so schmal wie möglich ist.

Auch die Einschränkungen des an der Datenbank angelernten Organismus‘ verschwinden, wenn man die Evolution durch Spiele gegen ein Zufallsprogramm fortschreiten läßt. So muß kein Anfangszug vorgegeben werden, und es ist auch möglich, einen Organismus erschaffen zu lassen, der als Nachziehender perfekt spielt. Mehr noch, man kann das Spiel auf bestimmte Gegner optimieren und sogar völlig obskure Ziele realisieren: So entstand wegen eines Programmierfehlers ein Organismus, der als Anziehender niemals verlor, aber als Nachziehender praktisch *niemals gewann*. Gegen einen Zufallszüge spielenden Gegner von 5000 Partien nur zwei oder drei zu gewinnen ist ganz gewiß schwieriger als niemals zu verlieren.

Es hat sich gezeigt, daß zumindest für so ein relativ einfaches Problem wie TicTacToe eine Baumsuche von neun Halbzügen komplett durch eine von einem genetischen Algorithmus optimierte Bewertungsfunktion ersetzt werden kann! Und selbst wenn dies nicht allgemein gilt – denn natürlich kann man nicht jede beliebige Baumsuche ersetzen – zeigt sich doch, wie geeignet ein genetischer Algorithmus zum Entwerfen einer Bewertungsfunktion ist. Eine Bewertungsfunktion an den Blättern einer Suche muß keineswegs perfekt sein – aber je genauer sie ist, desto effizienter arbeitet die Suche.

## Maschinen-Sex ohne Inzucht

Die Fitness einer sich entwickelnden Population steigt am Anfang schnell an; die Geschwindigkeit der Fitness-Steigerung sinkt aber mit der Zeit immer mehr; die Fitness der besten Organismen nähert sich asymptotisch einem Maximalwert, von dem man hofft, er möge dem theoretisch erreichbaren Maximum so nahe wie möglich kommen. Irgendwann konvergiert die Population; die erfolgreichsten Gene verbreiten sich und fast alle Organismen erreichen dieselbe Fitness. Auf die maximale Fitness einer Population und die zum Erreichen derselben benötigte Zeit haben etliche Parameter großen Einfluß, insbesondere die Populationsgröße, die Anzahl möglicher Nachkommen, die Auswahl der Partner sowie die Mutationsrate.

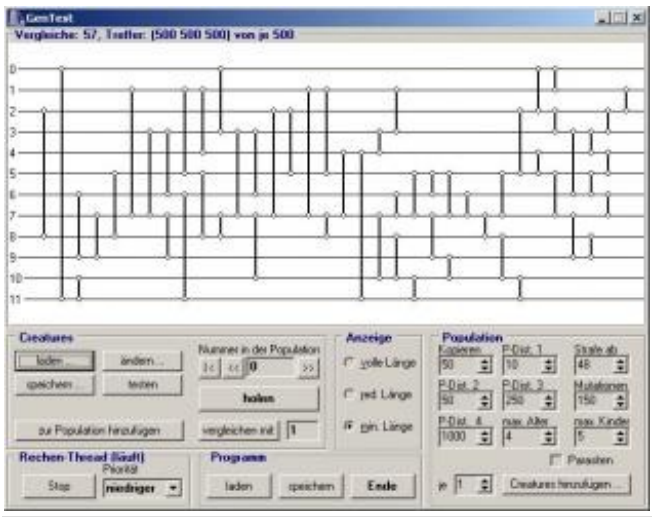
Der genetische Algorithmus ist keineswegs perfekt, sondern leidet unter einigen Problemen, die auch aus der natürlichen Evolution bekannt sind. So neigen zum Beispiel endemische Arten, von denen es nur eine relativ kleine Population auf begrenzten Raum gibt, eher zum Aussterben, weil schon kleine Änderungen der Umwelt für die gesamte Art lebensbedrohlich werden. Die Viecher im Computer können nicht aussterben, aber sie können aufs Zufalls-Niveau zurückgeworfen werden.

Auch der Sechs-Finger-Effekt, bekannt aus kleinen, abgeschiedenen Dörfern, spielt eine große Rolle: erfolgreiche Gene werden weitergegeben und verbreiten sich in der Population. Gerade in Organismen mit relativ wenig Genen, die in kleinen Populationen leben, neigen die künstlichen Organismen zur Inzucht und haben irgendwann alle identische Gene. Da kann man kreuzen, soviel man will, es gibt keinen Fortschritt mehr, außer durch seltene Mutationen. Die Population hat eine gewisse Anpassung erreicht, aber meist nicht die bestmögliche.

Der genetische Algorithmus tut ja streng genommen etwas Ungewolltes. Der Anwender ist an einer möglichst guten Problemlösung interessiert, also ausschließlich am erfolgreichsten, fittesten Organismus. Der genetische Algorithmus aber optimiert gerade nicht den Einzelorganismus, sondern die gesamte Population. Der Einstein der Organismen ist in der künstlichen Welt nur eine Eintagsfliege, nach Ablauf seiner Lebensdauer vergessen. Ignoriert man diesen Sachverhalt, führt das dazu, daß die Population schnell einem lokalen Maximum entgegenstrebt und dort verharrt. Den Lösungsraum vieler Probleme kann man sich vorstellen wie eine Landschaft mit Bergen, Hügeln und Tälern, wobei die Höhe das Maß für die Qualität der Problemlösung darstellt. Eine Population „wandert“ durch diese Landschaft und wird durch die Selektionsmechanismen gedrängt, möglichst weit bergauf zu „siedeln“. Dabei landet die Population nahezu zwangsläufig auf einer Erhebung, von der sie normalerweise nicht herunterkäme, weil dafür eine temporäre Reduzierung der Fitness erforderlich wäre. Diese Erhebung stellt jedoch nur in seltenen Ausnahmefällen die höchste Erhebung in der Lösungsraum-Landschaft dar – die Population verharrt also bei einer suboptimalen Lösung.

Um Inzucht zu vermeiden, muss der Programmierer geeignete Mechanismen zur Fortpflanzung implementieren: die Anzahl der möglichen Nachkommen ebenso wie das Lebensalter eines erfolgreichen Organismus' einschränken, dafür sorgen, dass zu ähnliche Partner keine Nachkommen zeugen, sich kümmern, dass auch schlappere Organismen gelegentlich noch ein paar Nachkommen hinterlassen dürfen ... vor allem jedoch hängt es an der Populationsgröße: je mehr Lebewesen zu einer Population gehören, desto größer ist der Genpool, desto geringer also die Chance, einen zu ähnlichen Partner zu finden.

Als weitere wirkungsvolle Maßnahme gegen Inzucht hilft eine feindliche Umwelt, zum Beispiel Parasiten. Es wird einfach eine zusätzliche Population von Lebewesen angelegt, deren Selektionsmechanismen genau umgekehrt funktionieren: sie werden belohnt, wenn sie möglichst schwierige Testfälle für die eigentlichen Organismen schaffen können. Je fieser sie die Organismen schädigen, indem sie ihnen schwierige Aufgaben stellen und damit ihren evolutionären Erfolg mindern, desto stärker dürfen sie sich vermehren. Die Organismen werden dadurch gezwungen, sich an wechelnde Bedingungen anzupassen, was das Festsetzen der Population auf einem lokalen Maximum unmöglich macht – kaum haben es sich die künstlichen Lebewesen auf einem kleinen Hügel gemütlich gemacht, schon schwappt eine Welle aggressiver Parasiten über sie hinweg und zwingt die Population, sich eine größere Erhebung in der Lösungsraum-Landschaft zu suchen, denn nur möglichst perfekte Anpassung verspricht sicheren Schutz vor den Parasiten.



Ein mit GenSort visualisiertes Sortiernetzwerk

Wer noch nicht wirklich glaubt, dass sowas wirklich funktioniert, kann es selbst ausprobieren. Auf der Webseite des Autors gibt es das Programm GenSort zum Download, das Sortiernetzwerke erzeugt und visualisiert. Es geht wie bei Hillis darum, mit so wenig wie möglich vorher feststehenden Vergleichen eine Zahlenfolge aufsteigend zu sortieren. Die Fortpflanzungs-Parameter sind in weiten Grenzen veränderbar. Es ist spannend, dabei zuzusehen, wie zuerst die Sortierfähigkeit zunimmt und dann die Anzahl der dafür nötigen Operationen reduziert wird. Auch die Fragilität einer Population und ihre Empfindlichkeit gegen Änderungen der Umwelt kann man prima erproben, indem man einfach mal ein paar Meter verdreht. Vielleicht sieht man bedrohte Tierarten auf Madagaskar oder Neuseeland ja auch mit anderen Augen, wenn man mal ein paar künstliche Populationen mit einem Fingerschnippen ruiniert hat?

## Rechenexempel Schach

Genetischen Algorithmen haben einige angenehme Eigenschaften: sie sind einfach zu programmieren, trivial zu parallelisieren, und es macht kaum einen Unterschied in d er Rechenzeit aus, ob fünf Parameter angepasst werden sollen oder fünfhundert! Wäre es nicht einen Versuch wert, sie auf die Bewertungsfunktion eines Schachprogramms wie Fruit oder Toga loszulassen? Was müsste man tun? Ein Schachprogramm ändern und so viele Parameter wie möglich von außen einstellbar machen, sodann eine existierende Implementierung des genetischen Algorithmus' nehmen, zum Beispiel das GenSort-Programm, und die C++-Klasse, aus der die Lebewesen erzeugt werden, mit einem passenden Gensatz ausrüsten, der die Parameter der Engine ändern kann. Und dann müsste man nur noch Zeit haben und zum Beispiel in jeder Generation ein Schweizer-System-Turnier durchführen. Die Anzahl der Runden hinge von der Populationsgröße ab; geht man von ca. 500 Organismen aus, reichten neun Runden. Zwar erlauben Schweizer-System-Turniere keine wirklich gute Reihung nach Spielstärke, aber grob reicht hier, denn jeder Organismus hat eine Chance, sich fortzupflanzen.

Pro Generation wären das also 500 x 9 = 4.500 Partien. Ganz schön viele, und die meisten davon für die Tonne, weil von vermurksten Parametersätzen gespielt. Rechnet man für die Optimierung der Population 50 Generationen, käme man auf 225.000 Partien. Weil es um die Bewertungsfunktion ginge, reichte eine ganz kurze Bedenkzeit, rechnen wir mal mit mit einer Minute pro Partie, dann ergäbe eine Optimierung der Bewertung eine Gesamtrechenzeit von etwa 225.000 Minuten, also 156 Tagen. Das könnte sogar einer allein noch reißen, erst recht aber eine kleine Gruppe von Computerschachfreunden! Der genetische Algorithmus lässt sich prima auf nahezu beliebig viele Rechner verteilen, ohne ernste Performance-Einbußen zu erleiden, und wäre das nicht mal ein spannenderes Experiment als ewig Ranglisten zu erstellen?

Es gab sogar schon einmal den Versuch, ein evolutionäres Schachprogramm zu züchten. Das Evochess getaufte Projekt krankte jedoch an einer fest vorgeschriebenen Suchtiefe, was dumm ist, weil im Endspiel ja tiefer gesucht werden kann, zudem war die Anzahl der pro Zug untersuchten Stellungen auf 100.000 begrenzt, was ebenfalls kontraproduktiv sein dürfte. Und letztlich war es gar kein richtiges Schachprogramm, sondern eine selbstgebaute Frickellösung, die mit einer rudimentären Alpha-Beta-Suche auskommen musste, die auch noch von der Evolution beeinflusst wurde. Der Versuch aber, die Bewertung eines richtigen, existierenden Spitzenprogramms zu optimieren, der wäre neu und aufregend. Und es gäbe in jedem Fall ein spannendes Ergebnis, entweder ein stärkeres Programm, oder aber den deutlichen Hinweis darauf, dass die Programmierer mit ihren handgewichteten Bewertungen eben doch nicht zu verbessern sind! (*Lars Bremer*)