

Robotersteuerung mit genetischer Programmierung *

Udo Stenzel Heiko Stamer

17. Oktober 2000

Zusammenfassung

In diesem technischen Report beschreiben wir einige Experimente mit genetischer Programmierung. Hauptziel war es, Steuerprogramme für Roboter (in unserem Fall für das Modell Khepera) mittels Evolution zu erzeugen und zu verbessern.

Nach anfänglichen Erfolgen mussten wir leider feststellen, daß selbst einfache Aufgaben, wie z.B. die Suche nach Lichtquellen, für unsere imaginäre Roboterpopulation kaum zu bewältigen waren. Die Ursachen sind wohl in unausgereiften Algorithmen, Schwierigkeiten bei der Bestimmung der Fitness-Funktion und im Wesen der genetischen Programmierung selbst zu suchen. Letztendlich konnten wir mit einem reduzierten Termsatz doch noch zufriedenstellende Ergebnisse erzielen.

Vielleicht kann unsere Arbeit trotzdem als Ansatz für künftige Projekte und Forschungen auf diesem Gebiet dienlich sein.

1 Einführung

In diesem Abschnitt wollen wir kurz auf die Theorie der genetischen Programmierung eingehen, Voraussetzungen klären und einen Überblick zur Benutzung des Khepera-Simulators geben.

1.1 Genetische Programmierung

Im Gegensatz zu genetischen Algorithmen, deren Population jeweils aus genetischen Strings besteht, versucht die **genetische Programmierung**, evolutionäre Operatoren auf eine Population von Programmen/Steuersequenzen anzuwenden. Ein wesentlicher Unterschied besteht also in der Interpretation der Individuen einer Population. Bei der genetischen Programmierung werden diese als Ableitungsbäume einer Grammatik aufgefasst. Dieses Vorgehen bedingt natürlich auch die Änderung der genetischen Operatoren, so daß die Anwendbarkeit auf Baumstrukturen gewährleistet wird. Kritisch sind hier besonders die **Initialisierung der Population**, die **Kreuzung** und die **Mutation**. Die Fitness-Bewertung eines Individuums erfolgt dann durch Auswertung/Interpretation seines Programms.

1.2 Voraussetzungen zur Steuerung des Khepera-Roboters

Der Khepera-Roboter verfügt als Fortbewegungsmittel über zwei von Schrittmotoren betriebene Räder. Diese können getrennt mittels Angabe einer Geschwindigkeit gesteuert werden.

*udo@arthur.sax.sub.de, stamer@informatik.uni-leipzig.de

Zur Wahrnehmung seiner Umgebung besitzt der Roboter 8 Infrarot-Sensoren, von denen 6 an der Front und 2 am Heck angebracht sind. Diese Sensoren arbeiten auf Reflexionsbasis, erlauben jedoch auch Messung von ambienten Lichtverhältnissen.

1.3 Arbeiten mit der Simulationssoftware

Eine detaillierte Beschreibung des Khepera-Simulators findet man in [1]. Der Autor *Olivier Michel* möchte in jeder wissenschaftlichen Arbeit (die den Simulator verwendet) gerne zitiert werden, was wir hiermit tun:

Olivier Michel. *Khepera Simulator* Package version 2.0: Freeware mobile robot simulator written at the University of Nice Sophia - Antipolis by Oliver Michel. Downloadable from World Wide Web at <http://wwi3s.unice.fr/~om/khep-sim.html>

Grundsätzlich gestaltet sich die Arbeit mit dem Simulator als schwieriges Unterfangen. Das zugrundegelegte Sensormodell ist von der X11-Oberfläche abhängig, was unsere Bemühungen in 2.5 (auf Seite 6) nicht gerade erleichterte. Das Konzept des *Khepera Simulators* und dessen Realisierung in C sind unausgereift.

2 Implementierung

Die Umsetzung der genetischen Programmierung erfolgte in C++, das durch seine Objektorientierung beste Voraussetzungen zur Schaffung eines erweiterbaren Konzeptes bot.

2.1 Grammatik und Sprache

Im folgenden gehen wir auf die Elemente unserer Termbäume ein, die, als Ganzes betrachtet, Steuerprogramme für den Roboter darstellen. Im Anhang A auf Seite 10 sind solche Programme aufgelistet.

Grundsätzlich wird bei der Auswertung von Ausdrücken mit Fließkommazahlen vom Typ `double` gearbeitet. Deren Genauigkeit ist für unsere Zwecke ausreichend.

2.1.1 Konstanten

Es kommen ganzzahlige Konstanten aus dem Intervall $[-2, 2]$ zum Einsatz, die in einen `double`-Wert umgewandelt werden. Das Intervall wird in `NonaryOp.hh` bei der Methodendefinition `ConstOp::create(int)` festgelegt. Bei unseren Experimenten standen also die Konstanten $-2, -1, 0, 1, 2$ zur Verfügung.

2.1.2 Variablen

Wir führten folgende nullstellige Operatoren ein:

<code>x</code>	x-Position des Roboters im Simulator
<code>y</code>	y-Position des Roboters im Simulator
<code>a</code>	α -Ausrichtung des Roboters im Simulator
<code>l1 - l8</code>	IR-Lichtsensoren (8 Stück)
<code>d1 - d8</code>	IR-Entfernungssensoren (8 Stück)

Die ersten drei Operatoren setzen wir später allerdings nicht ein, um eine Abhängigkeit vom Simulator zu vermeiden. (`x`, `y`, `a` sind Parameter die der Simulator in der Struktur `Robot`¹ liefert.) Im Gegensatz dazu sind die Entfernungs- und Lichtsensoren auch beim realen Khepera-Roboter verfügbar.

Alle Klassendefinitionen für **Konstanten** und **Variablen** sind in `NonaryOp.hh` und `NonaryOp.cc` zusammengefasst und können dort geändert werden.

2.1.3 Funktionen

Die ein- und zweistelligen Operatoren kann man in Klassen aufgliedern:

arithmetische Operationen:

- + Addition zweier Ausdrücke
- Subtraktion zweier Ausdrücke
- * Multiplikation zweier Ausdrücke
- / Division zweier Ausdrücke²

Die Dateien `BinaryOp.hh` und `BinaryOp.cc` stellen diese Operationen bereit.

Winkelfunktionen:

- `sin` Sinus eines Ausdrucks
- `cos` Cosinus eines Ausdrucks

transzendente Funktionen:

- `exp` Exponentialfunktion
- `log` Logarithmus

Diese Klassen findet man in `UnaryOp.hh` und `UnaryOp.cc`.

2.1.4 Kontrollfluß

Als Kontrollflußanweisung implementierten wir nur die **Alternative**.

- ? Alternative

Bei dieser dreistelligen Operation wird verglichen, ob der erste Operant größer Null ist. Wenn dieser Vergleich zutrifft, wird der zweite, ansonsten der dritte Ausdruck bei der Auswertung zurückgegeben.

In ANSI-C könnte man dieses Verhalten mit `(op1 > 0) ? op2 : op3` beschreiben.

`TertiaryOp.hh` und `TertiaryOp.cc` enthalten diese Funktionalität.

¹siehe `SRC/robot.h` beim Khepera-Simulator von *Olivier Michel*

²Um Stetigkeit zu gewährleisten, wird bei Division durch Zahlen nahe Null ($< 10^{-12}$) die Konstante `1e+12` zurückgegeben.

2.2 Genetische Operatoren

Folgende Abbildung zeigt das genetische Ablaufschema, das bei unserer Implementierung zum Einsatz kommt. Bei der Erstellung einer neuen Population wird jedes Individuum durch **Zufallsinitialisierung** erzeugt. Das Individuum wird danach durch **Simulation** auf Fitness getestet, anhand der dann die **Selektion** innerhalb der Population erfolgt. Die verbliebenen, starken Eltern werden der **Kreuzung** unterworfen, wodurch zwei Kinder entstehen. Beide durchschreiten die **Mutation**, um lokale Extrema zu vermeiden. Alle Kinder und die fitere Hälfte der Elternpopulation bilden eine neue Generation und gelangen wieder zur Simulations-Stufe.

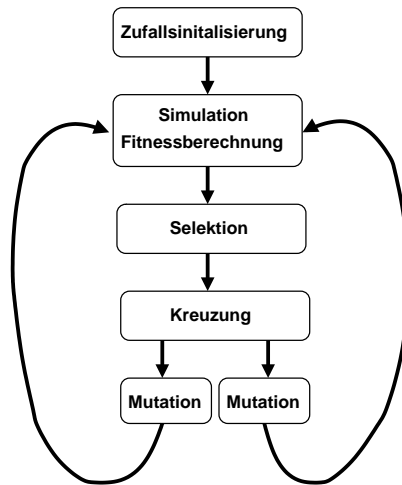


Abbildung 1: genetisches Ablaufschema

2.2.1 Zufallsinitialisierung

Bei der Zufallsinitialisierung wird die Startpopulation erzeugt. Hierzu ist dem Konstruktor der Klasse `Population` die Größe der zu erstellenden Population zu übergeben. Bei uns passiert dieser Schritt in der Datei `user.c` des Khepera-Simulators.

Für jedes Individuum werden dabei die Termbäume (siehe 2.4.1) angelegt. Wichtige Parameter bei diesem Prozess sind:

- verwendete **Termbausteine** (siehe `Term.cc`)
- **Anzahl der Knoten** des Termbaums

Als Termbausteine verwendeten wir von vornherein alle Funktionen aus 2.1.3, was sich später als Problem herausstellen sollte. Die Anzahl der Knoten bei der Initialisierung legten wir auf 25 fest. Dies schien uns, als Mittel zwischen Übersichtlichkeit / Geschwindigkeit und Komplexität, ein geeignetes Maß zu sein. Die Knotenzahl kann in der Datei `Individual.hh` geändert werden, um ggf. komplexere Strukturen/Bäume zu erzeugen. Möchte man nur mit eingeschränkten Termkonstrukten arbeiten, sollte man `Term.cc` an seine Bedürfnisse anpassen.

2.2.2 Selektion und Kreuzung

Unsere Selektionsstrategie ähnelt der *linearen Rangselektion*. Zuerst werden die Individuen nach Fitness absteigend sortiert. Nach Berechnung der Gesamtfitness, des Mittelwerts und der Standardabweichung wird die Hälfte der Population gekreuzt.

Die Auswahl von „Mutter“ und „Vater“ geschieht zufällig. Die dabei entstehenden zwei Kinder werden der neuen Generation zugeordnet. Danach erfolgt der eigentliche Selektionsschritt, wobei die obere Hälfte der ursprünglichen Individuen auch noch in die neue Generation übernommen wird. Die Populationsgröße bleibt hierbei konstant.

Diese Schritte können in unserer Methode `Population::cross_and_select()` aus der Datei `Population.cc` nachvollzogen werden. Besonders die Implementierung des Kreuzungs-Operators erforderte erheblichen Aufwand, da die Kreuzung nur an Termbaumknoten erfolgen kann, die über gleiche Stelligkeit und Gewicht verfügen. Der Kreuzungspunkt wird nach diesen Bedingungen zufällig gewählt.

2.2.3 Mutation

Nach Erzeugung der Kinder werden diese mit einer bestimmten Wahrscheinlichkeit der Mutation unterworfen. Diese Mutationsrate kann in `Population.cc` mit der Variable `radioactivity` festgelegt werden. Bei unseren Versuchen verwendeten wir einen Wert von 0.07, der einer Veränderung pro 700 Individuen entspricht.

Ebenso wie bei der Kreuzung nicht beliebige Teile der Ableitungsbäume ausgetauscht werden können, kann auch bei der Mutation nicht ein beliebiger Teil völlig zufällig verändert werden. Vielmehr wird ein Knoten im Termbaum als Mutationspunkt gewählt. An dieser Stelle wird dann ein neuer zufälliger Teiltermbaum gleichen Gewichts eingesetzt. Dabei kommen Methoden des Kreuzungs-Operators zum Einsatz.

2.3 Fitnessfunktion

Wesentlicher Bestandteil und Ausgangspunkt bei genetischer Evolution ist die Berechnung einer **Rohfitness** für jedes Individuum der Population. Diese Rohfitness muß etwas über die Güte des Individuums in Bezug auf die Aufgabenstellung aussagen.

Unsere experimentellen Fitnessfunktionen stellen wir später noch vor.

2.4 Simulation

Zur Simulation der Population nutzten wir zuerst den unveränderten Simulator von *Olivier Michel*. Dazu implementieren wir eine entsprechende Simulationssequenz im zugehörigen Teilstück `user.c`. Später benutzen wir einen eigenen „Brutkasten“ (siehe Abschnitt 2.5 auf Seite 6).

2.4.1 Erster Ansatz

Der erste naive Ansatz war, die beiden unabhängigen Motoren (für das rechte und linke Rad des Khepera-Roboters) separat durch eigene Termbäume zu steuern. Nur Experiment 3.1.1 wurde damit durchgeführt.

2.4.2 Richtung und Geschwindigkeit

Um unseren ersten Ansatz zu verbessern, führten wir eine Modifikation durch. Wir implementierten nun jeweils einen genetischen Termbaum für Richtung und Geschwindigkeit. Da die Motoren bei vielen Bewegungen voneinander abhängig arbeiten, schien dieses Modell eine höhere Erfolgswahrscheinlichkeit zu haben.

2.5 „Brutkasten“ (Veränderter Khepera-Simulator)

Um die Evolution auch im Hintergrund und über einen längeren Zeitraum zu ermöglichen, mussten wir Ausgabe und Interaktivität des Simulators einschränken. Da es uns beim „Brutkasten“ im wesentlichen aber auf die Berechnung der Evolution ankam, konnten wir durch die Vermeidung von Bildschirmausgaben (X11) die Geschwindigkeit des Simulationsprozesses auch noch deutlich steigern.

Konkret nahmen wir folgende Änderungen vor:

- Vermeidung jeglicher Interaktivität mit dem Benutzer
- Zufallsinitialisierung bzw. Laden einer Simulationspopulation
- Ausführung der genetischen Evolution
- regelmäßige Sicherung der Population
- Änderung der Kollisionsberechnung

Die regelmäßige Sicherung war vor allem notwendig geworden, da sich aufgrund der Dualboot-Option des Rechnerpools unsaubere Computerabschaltungen und damit der Verlust der gesamten Simulationspopulation häuften.

Die Veränderungen am Khepera-Simulator können den `patch`-Quellen im Anhang C entnommen werden.

3 Ergebnisse

3.1 Experimente

Wir führten verschiedene Experimente mit unserer Implementierung durch, um deren Tauglichkeit zu testen. Dabei gemachte Beobachtungen werden im Abschnitt 3.2 diskutiert.

3.1.1 Ansteuern eines Punktes

Hier hatte der Roboter die Aufgabe, einen bestimmten Punkt P anzufahren und dort zu halten. Der Start erfolgte von allen vier Ecken der quadratischen „Welt“. Als Fitnessfunktion wählten wir den Abstand des Roboters von P . Dieser Wert wurde über die einzelnen Simulationsschritte aufsummiert.

Dieses Experiment funktionierte recht gut - die Roboter „lernten“ den Weg von den Startpunkten zu P „auswendig“. Hier waren leider auch noch die Operatoren `x`, `y` und `a` im Einsatz.

3.1.2 Suche nach Lichtquellen

Die nächste Aufgabe war schon etwas komplexer. Der Roboter sollte eine Lichtquelle finden und sie ansteuern. Hier fielen die Ergebnisse auch deutlich schlechter aus, was wohl in der geringen Reichweite der Lichtsensoren begründet lag. Bei diesem Versuch erhielten wir meist „tote“ Roboter (siehe Anhang A.1).

3.1.3 Folgen einer „Lichtspur“

Aufgabenstellung bei diesem Experiment war das Folgen einer gelegten Lichtspur. Der Roboter sollte eine erste Lichtquelle finden, diese ausschalten und dann zur nächsten fahren. Der Schritt des Ausschaltens wiederholt sich an jeder Lichtquelle, so daß dieser Prozess eine Ähnlichkeit mit dem „Dominoprinzip“ hat. Da die Enden der Lichtspur verbunden sind, ist der Startpunkt irrelevant.

Unsere ersten Versuche zeigten erschreckende Ergebnisse. Die Roboter folgten entweder der Spur nicht bzw. fanden sie gar nicht erst. Es entstanden etliche „tote“ Roboter und „Zitterer“. Veränderungen an den Parametern der genetischen Evolution brachten auch keinen Erfolg.

3.1.4 Reduzierter Termsatz - Ein neuer Versuch

Erst eine radikale Einschränkung des Termsatzes stellte eine Lösung dar. Wir verwendeten nun nur noch die Operatoren l1-l8, d1-d8, +, -, * und ?. Unter diesen Beschränkungen entwickelten sich die Roboter hervorragend, so daß in der 328. Generation ein fast perfekter „Lichtsammler“ entstand.

Die Fitnessfunktion (siehe `Individual.hh`) berücksichtigt neben der Entfernung zur Lichtquelle auch noch die Termbaumgröße. Kleinere Programme bekommen so einen gewissen Vorzug.

3.2 Beobachtungen

3.2.1 Konvergenz der genetischen Programmierung

Nur beim Experiment 3.1.4 konnten wir ein Konvergenzverhalten beobachten. Folgende Abbildung zeigt diese Entwicklung:

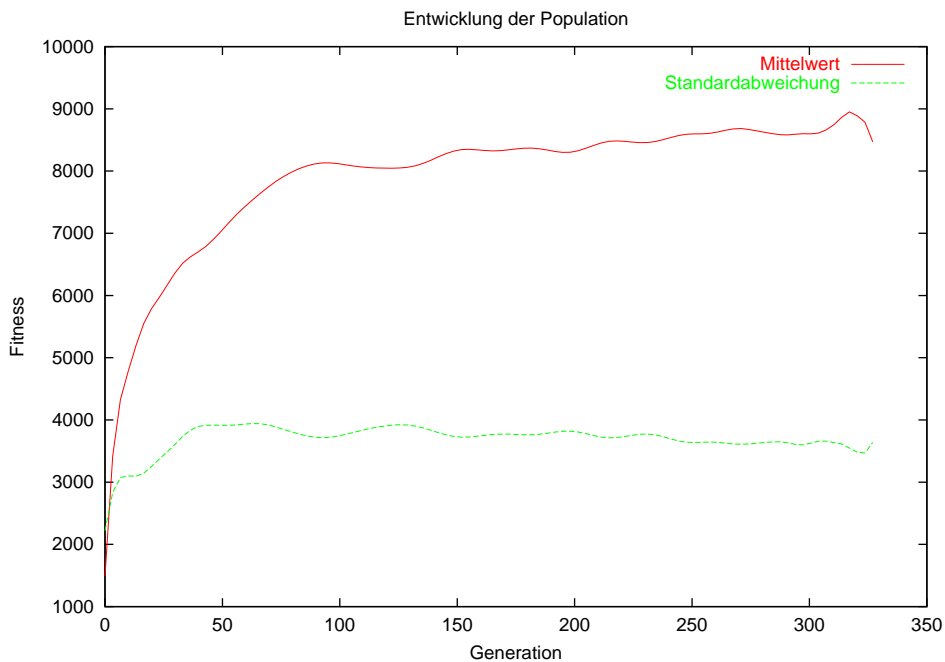


Abbildung 2: Entwicklung bei Experiment 3.1.4

Die Steigerung der durchschnittlichen Fitness ist deutlich zu erkennen. Interessant ist die leichte Abwärtstendenz der Standardabweichung ab der 50. Generation. Dieser Trend ist auf **genetische Verarmung** und zunehmende **Konvergenz** zurückzuführen.

Die seltsame Kurvenform ab Generation 325 ist mit Glättung der Rohwerte durch den Bezieralgorithmus zu erklären. Die Kardinalität der Punktmenge am Ende des Entwicklungsintervalls war für eine Glättung unzureichend und hätte bei der Visualisierung abgeschnitten werden müssen.

3.2.2 Vergrößerung der Termbäume

Trotz unserer Anstrengung mittels Fitnessfunktion die Termbaumgröße zu verkleinern, trat ein umgekehrter Effekt ein. Die Termbäume wurden im Laufe der Entwicklung immer größer; die Zahl der Knoten stieg von anfangs 40 auf 748 am Ende an. Diese Beobachtung konnten wir aber nur beim reduzierten Termsatz in Experiment 3.1.4 machen. Bei allen anderen Versuchen blieb die Termbaumgröße konstant oder sank sogar leicht. Die Ursachen für diesen Effekt sind noch nicht geklärt.

3.2.3 Genetische Fluktuation

Die nächste Abbildung zeigt die abnormalen Ergebnisse bei vollem Termsatz:

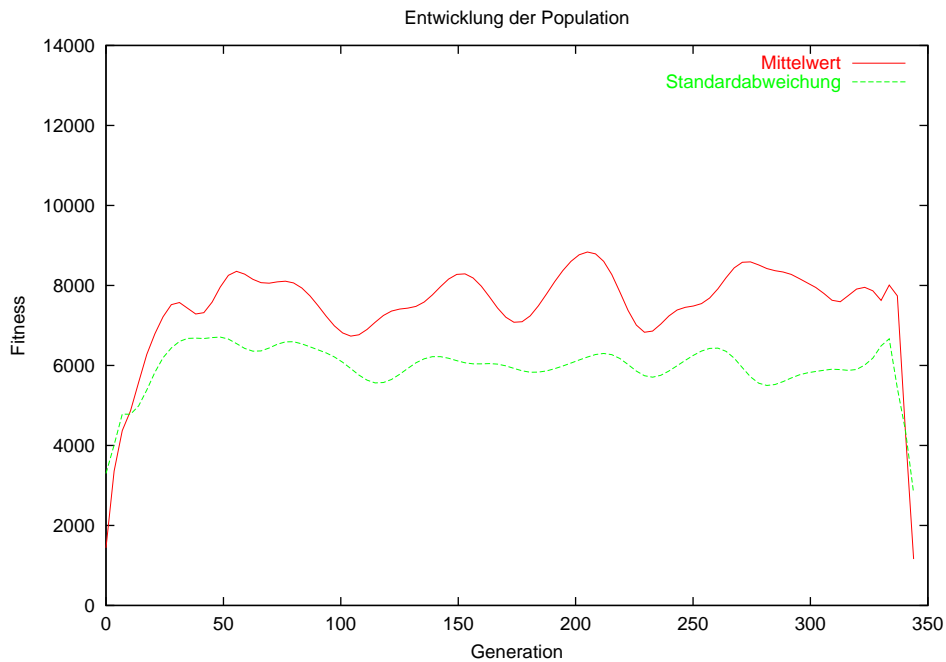


Abbildung 3: Entwicklung bei Experiment 3.1.3

Es ist unklar, ob es sich bei dem beobachteten Verhalten um **genetische Fluktuation** handelt oder ob hier Implementierungsfehler zum Tragen kommen.

Verfolgt man die Entwicklung über einen noch längeren Zeitraum, wird die durchschnittliche Fitness immer schlechter und fällt irgendwann unter die Standardabweichung. Auffällig ist auch die starke Volatilität der beiden Werte.

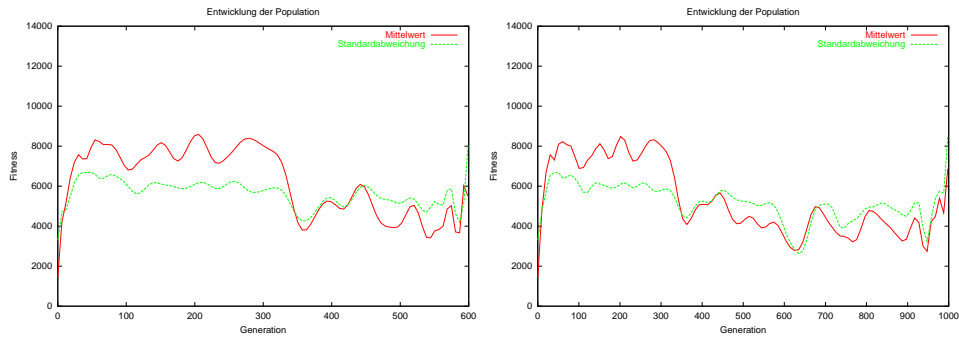


Abbildung 4: weitere Entwicklung bei Experiment 3.1.3

3.2.4 Genetische Verarmung

Ein generelles Phänomen, welches wir auch beobachten konnten, ist die sogenannte **genetische Verarmung**. In älteren Generationen war ein nicht geringer Teil der Steuerprogramme sich ähnlich, wenn nicht sogar identisch. Mit fortschreitender Evolution wurde dieses Verhalten immer deutlicher sichtbar.

Durch Erhöhung der Mutationsrate auf 0.07 konnte dem Effekt etwas vorgebeugt werden. Genetische Verarmung war aber trotzdem festzustellen.

4 Ausblick

Weitere Möglichkeiten, die wir leider noch nicht testen/implementieren konnten, möchten wir noch kurz vorstellen:

- genaue Beschreibung der Einflußnahme von Termteilmengen auf das Konvergenzverhalten der genetischen Programmierung
- Einbau verschiedener Selektionsmethoden, Fitnessberechnungen³
- Implementierung des Kontrollflußelements: Schleife
- Testläufe mit komplexeren Aufgabenstellungen (Fitnessfunktion !)

Wir möchten uns hiermit auch bei *Dr. R. Der* bedanken, der durch den Rahmen seines Robotik-Praktikums diese Arbeit erst ermöglicht hat.

Wir hoffen, daß unsere Arbeit trotz der offensichtlichen Probleme und Unzulänglichkeiten einige Ideen und Anregungen für weitere Forschung und Entwicklung auf diesem Gebiet liefert. Falls Sie die Implementierung bzw. Teile davon einsetzen, verbessern bzw. weiterentwickeln möchten, scheuen Sie sich bitte nicht, mit uns in Kontakt zu treten. Wir helfen Ihnen gern weiter.

Literatur

- [1] Olivier Michel: *Khepera Simulator version 2.0 - User Manual*
University of Nice - Sophia Antipolis, Laboratoire I3S, CNRS
bât. 4, 250, av. A. Einstein 06560 Valbonne, France, March 1996
- [2] Helmut Hörner: *Ein Kern für genetisches Programmieren in C++*
Diplomarbeit an der Wirtschaftsuniversität Wien, April 1996

³z.B. normalisierte, adjustierte, standardisierte Fitness

A Beispiele für genetische Steuerprogramme

Bei unseren Experimenten konnten wir hauptsächlich abnormale Roboterprogramme beobachten. Einige Beispiele aus einer 170er Population mit Termbaumgröße (Initialisierungspopulation) von maximal 25 Knoten wollen wir angeben:

A.1 „Toter“ Roboter

Das folgende Programm stellt einen sogenannten „Toten“ dar, d.h. einen Roboter, der sich nicht von der Stelle bewegt. Er entstand als Bester in der 4. Generation mit einer Fitness von $1.80e+04$. Der Fitness-Mittelwert der gesamten Generation betrug $3.95e+03$, die Standardabweichung war $3.81e+03$.

```
( 12 cos d1 d5 - / d1 d3 * cos l1 * exp exp sin exp + exp ,  
11 -1.000000 2.000000 d3 ? log - d0 -1.000000 d5 ? * cos )
```

Die erste Komponente (vor dem Komma) entspricht dem Termbaum für die Geschwindigkeit, die zweite Komponente (nach dem Komma) dem Termbaum für die Richtung. Alle Termbäume sind in umgekehrter polnischer Notation angegeben, da dies Parsing und Interpretation erleichtert.

A.2 „Kreisel“

Oft entstanden auch „Kreisel“-Roboter, die sich an der Stelle bzw. um eine imaginäre Achse drehen. Solche Modelle waren meist wertlos.

```
( 13 17 17 / + exp 13 12 - 14 d1 -1.000000 14 ? sin / + sin - ,  
15 1.000000 exp -2.000000 -1.000000 d7 * sin ? d2 exp - exp exp  
cos log exp * )
```

A.3 „Zitterer“

Dieser Kandidat bewegt sich kurz vor und dann gleich wieder zurück. Er „zittert“ an der Stelle. Dieses Verhalten zeigt er auch in vollkommener „Dunkelheit“.

```
( 14 d2 2.000000 * * 13 d2 ? cos sin sin d6 d3 0.000000 / sin exp  
? , 10 12 exp -2.000000 2.000000 * + 17 d3 d5 * ? exp sin / )
```

A.4 Ein „Sonnenanbeter“

Hierbei handelt es sich schon um einen recht interessanten Roboter. Er richtet seine Front (vorne rechts) nach der stärksten Lichtquelle, die er durch die Sensoren wahrnimmt, aus. Leider sucht er nicht nach Licht, sondern dreht sich bloß und bewegt sich zu wenig.

```
( 1.000000 d2 d6 - -1.000000 2.000000 - - log log * log exp  
0.000000 14 16 / + * , d1 14 / -2.000000 14 -1.000000 16 - exp ?  
cos sin log exp log d2 * / )
```

Dieses Exemplar konnte sich leider nicht durchsetzen. Sind zuviele Lichtquellen in seiner Nähe, tritt ein „Blendeffekt“ auf.

A.5 Ein fast perfekter „Lichtsammler“

Dieses Modell entstand bei unseren Experimenten mit dem eingeschränkten Term-satz in der 328. Generation. Seine Fitness betrug $1.13e+04$, der Fitness-Mittelwert der gesamten Population war $8.47e+03$. Der Roboter fährt die komplette „Lichtspur“⁴ ab. Der Startpunkt kann beliebig in der „Simulationswelt“ gewählt werden.

```
( d5 1.000000 + d6 + d3 -1.000000 * * d5 d3 -1.000000 * d5 10
-1.000000 d2 1.000000 + ? 10 -1.000000 d0 ? 1.000000 l1 - * d2
1.000000 + ? + d2 d2 - d2 l3 d0 + ? + -1.000000 l4 l1 ? -1.000000
* d6 l4 * * l2 1.000000 l1 - * 1.000000 1.000000 + ? d5 d2 ? + d5
1.000000 + ? d3 + ? -1.000000 l4 l1 ? -1.000000 * d6 l4 * * l2
1.000000 l1 - * 1.000000 1.000000 + ? d2 d3 + d5 -1.000000 * ? d5
-1.000000 * d5 1.000000 + d3 -1.000000 * d5 10 -1.000000 d0 ? 10
-1.000000 d0 ? 1.000000 l1 - * d2 1.000000 + ? + d2 d2 - l1 l3 d0
+ ? + d5 1.000000 -1.000000 1.000000 ? 10 -1.000000 d0 ? 1.000000
l1 - * d2 1.000000 + ? + + d2 1.000000 + ? d2 1.000000 + 1.000000
+ d2 d2 - d5 -1.000000 * l3 d0 + ? + d5 d5 ? + ? + d5 d3 d5
1.000000 + d5 d5 -1.000000 + + 1.000000 + + -1.000000 * * * *
-1.000000 d2 1.000000 + d5 + ? d3 1.000000 -1.000000 1.000000 ? *
d5 10 -1.000000 d2 1.000000 + ? 10 -1.000000 d0 ? 1.000000 l1 - *
d2 1.000000 + ? + 1.000000 + -1.000000 l4 l1 ? -1.000000 * d6 l4 *
* l2 1.000000 l1 - * 1.000000 1.000000 + ? d5 d2 ? + d5 1.000000 +
? d5 + ? ,
0.000000 d0 d3 ? d2 * l4 l1 d7 + - l3 l7 d1 + - d4 l2 d1 d5 ? * +
? d6 -1.000000 - -1.000000 - 0.000000 d7 d3 ? d2 * * d2 0.000000
d0 d3 ? d2 * * d1 d5 10 2.000000 d4 * - d0 l6 + d4 * ? l4 l1 d7 +
- l3 l7 d1 + - d4 l2 d1 d5 ? * + ? - 0.000000 ? d7 l4 l1 d7 + -
l3 l7 d1 + - d4 l2 d1 d5 ? * + ? - d4 l2 d1 d5 ? * ? - d6
-1.000000 - 0.000000 d7 d3 ? d2 * * -1.000000 d6 -1.000000 -
0.000000 d7 d3 ? d2 * * ? - d7 -1.000000 - 0.000000 d7 d3 ? d2 * *
0.000000 * d7 d6 d3 - 0.000000 d7 d3 ? d7 * * 0.000000 d2 0.000000
d0 d3 ? d2 * * d1 d5 10 2.000000 d4 * - d0 l6 + d4 * ? l4 l1 d7 +
- l3 l7 d1 + - d4 l2 d1 d5 ? * + ? - 0.000000 ? d7 d1 l3 l7 d1 + -
d4 l2 d1 d5 ? * + ? - -1.000000 - 0.000000 d7 d3 ? d2 * * l1 d7 +
* -1.000000 d2 0.000000 d0 d3 ? d2 * * 0.000000 d7 d3 ? d7 * *
0.000000 d6 d5 - 0.000000 d7 d3 ? d2 * * ? ? 0.000000 d7 d3 ? d2 *
* ? ? - 0.000000 d2 * d2 0.000000 d0 d3 ? d2 * * d2 d5 10 2.000000
d4 * - d0 l6 + d4 * ? l4 l1 d7 + - l3 l7 d1 + - d4 l2 d1 d5 ? * +
? - 0.000000 ? d6 l1 d7 + - - d6 ? d1 d7 -1.000000 - 10 2.000000
d4 * - d0 l6 + d4 * ? l4 l1 d7 + - l3 l7 d1 + - d4 l2 d1 d5 ? * +
? - 0.000000 d7 0.000000 -1.000000 - ? d2 * -1.000000 d2 0.000000
d0 d3 ? d2 * * d1 d5 10 2.000000 d4 * - d0 l6 + d4 * ? l4 l1 d7 +
- l3 l7 d1 + - d4 l2 d1 d5 ? * + ? - d7 ? d7 l4 l1 d7 + - l3 l7 d1
+ - d4 l2 d1 d5 ? * + ? - d2 0.000000 d0 d3 ? d2 * * d1 d5 10
2.000000 d4 * - d0 l6 + d4 * ? l4 l1 d7 + - l3 l7 d1 + - d4 l2 d1
d5 ? * + ? - 0.000000 ? d7 d1 l3 l7 d1 + - d4 l2 d1 d5 ? * + ? - ?
- ? - )
```

⁴Dabei handelt es sich um eine Aufreihung simulierter Lichtquellen. Das Gebilde hat die Form einer Acht.

B C++ Quellen

Die C++ Quelltexte sind recht umfangreich und können deshalb hier nicht direkt angegeben werden. Sie können sie jedoch übers WorldWideWeb via

`http://stinfwww.informatik.uni-leipzig.de/~mai97ixb`

beziehen. Sämtliche Quellen unterliegen der GNU GENERAL PUBLIC LICENSE (GPL)⁵ und sind nach deren Maßgabe zu behandeln.

C patch-Quellen für den Simulator

Die von `diff(1)` erzeugten `patch`-Quellen sind ebenfalls vom oben angegebenen URL zu beziehen. Mit `patch(1)` kann aus den ursprünglichen Quelltexten des *Khepera Simulators version 2.0* der veränderte „Brutkasten“ (siehe 2.5 auf Seite 6) erzeugt werden.

⁵siehe <http://www.gnu.org/copyleft/>