Fachbereich Informatik

Hausarbeit

im Studiengang Informatik Master of Science

Thema:	**Evolution sensomotorischer Kopplungen**
Fach:	Künstliche Intelligenz
eingereicht von:	Stephan Dreyer (20062042)
eingereicht am:	06. Juni 2010
Betreuer:	Dipl.-Inform. Ingo Boersch Prof. Dr.-Ing. Jochen Heinsohn

Inhaltsverzeichnis

1 Einführung

1.1 Überblick

Die Informatik beschäftigt sich unter anderem mit der Steuerung und Regelung von sensomotorischen Systemen. Der Bereich der künstlichen Intelligenz ist ein "Teilgebiet der Informatik, welches versucht, menschliche Vorgehensweisen der Problemlösung auf Computern nachzubilden, um auf diesem Wege neue oder effizientere Aufgabenlösungen zu erreichen."[LäCl01] Eine Analyse des Verhaltens von Tieren und Pflanzen bringt Informatiker jedoch ebenfalls ständig auf neue Lösungen für altbekannte Probleme, insbesondere im Bereich der Robotik. Als Folge solcher Beobachtungen entwickelte sich das Teilgebiet der evolutionären Algorithmen, dessen Ziel darin besteht, die Lösung von Problemen nicht zu programmieren, sondern den Computer die Problemlösung selbst finden zu lassen.

1.2 Aufgabenstellung

Die praktische Aufgabenstellung lautete "Konstruieren Sie ein mechatronisches System, das ständig mit einem evolutionären Algorithmus lernt". Dabei war der Aufbau des Systems in keiner Weise vorgeschrieben. Die Anforderung war lediglich, dass das System mit Hilfe eines evolutionären Algorithmus einen Fitnesswert optimiert und dabei in der Lage ist, sich an eine Veränderung der Umwelt anzupassen.

2 Grundlagen

2.1 Evolutionäre Algorithmen

Ein in der Natur bei allen Lebewesen anzutreffendes Prinzip zur Optimierung von Parametern ist die Evolution. Sie optimiert verschiedene Parameter zur Erreichung einer möglichst hohen Überlebensfähigkeit und zur Erzeugung von Nachkommen. Dies bezeichnet man in diesem Zusammenhang als Fitness. Lebewesen, die sehr gut an ihren Lebensraum angepasst sind und sich so gegenüber ihren Artgenossen durchsetzen können, erreichen eine hohe Fitness. Daraus folgt, dass sie ebenfalls eine hohe Chance haben, sich zu reproduzieren. Im Gegensatz dazu ist es bei Lebewesen mit einer geringen Fitness unwahrscheinlicher, dass sie sich vermehren und sich ihr Genom über Generationen hinweg durchsetzen kann. Nach Charles Darwin[Dar99] sind vier Bedingungen für das Auftreten von Evolution erforderlich:

- Vermehrung von Individuen einer Population
- Veränderungen, die die Vermehrungsrate beeinflussen

- Vererbbarkeit der Veränderungen
- Begrenzte Ressourcen

Die Aufgabe evolutionärer Algorithmen besteht nun darin, diese Bedingungen auf technische Systeme zu übertragen. Im Laufe der zweiten Hälfte des zwanzigsten Jahrhunderts haben sich vier Unterarten der evolutionären Algorithmen durchgesetzt:

- Evolutionäre Programmierung (EP)
- Evolutionsstrategien (ES)
- Genetische Algorithmen (GA)
- Genetisches Programmieren (GP)

Alle vier Verfahren funktionieren nach dem selben Schema: Lösungen für technische Probleme werden auf einzelne Individuen abgebildet. Ein Individuum kann beispielsweise ein Programm, ein Vektor von Parametern oder gar eine Baumstruktur sein. Für jedes Individuum muss es möglich sein, einen Fitnesswert zu ermitteln. Dieser kann berechenbar, durch eine subjektive Beurteilung oder implizit durch eine Überlebenswahrscheinlichkeit zu bestimmen sein. Außerdem müssen auf das Individuum genetische Operatoren (z.B. Mutation oder Crossover) anwendbar sein.
Die Arbeitsweise genetischer Algorithmen gliedert sich in zwei verschiedene Verfahren. Bei der generationenbasierten Arbeitsweise wird die gesamte alte Generation durch die neuere ersetzt. Im Gegensatz dazu werden bei der steady-state-Arbeitsweise nur einzelne Individuen der Population durch neue ersetzt. Dies erfordert eine Ersetzungsselektion. Ein wichtiger Parameter dafür ist das Alter eines Individuums. In den meisten Algorithmen werden die ältesten Individuen als erste ersetzt. Es kann jedoch auch von Vorteil sein, das Individuum mit der höchsten Fitness unabhängig vom Alter vor einer Ersetzung zu schützen. [BHS07]

2.2 Genetische Algorithmen

Genetische Algorithmen wurden 1975 vom US-amerikanischen Informatiker John Holland entwickelt. Diese benötigen als Grundlage eine geeignete Codierung des zu optimierenden Sachverhaltes, zumindest der Werte, die die Situation charakterisieren. Die zu optimierenden Größen werden häufig als ein einziger Bitstring dargestellt.[LäCl01]

Zunächst wird eine Population von Individuen zufällig erzeugt. Für jedes Individuum wird anschließend die Fitness bestimmt. Ausgehend von der Fitness werden zwei Individuen ausgewählt, daher eine Selektion getroffen. Zwischen diesen Individuen wird mit einer festgelegten Wahrscheinlichkeit P_c eine Kreuzung (Crossover) durchgeführt und mit einer ebenfalls definierten Wahrscheinlichkeit P_m eine Mutation durchgeführt. Eine Mutation bedeutet eine zufällige Abweichung von den Elternindividuen. Diese kann auf der Bitebene oder auf der Parameterebene stattfinden. Ein Crossover sieht meist so aus, dass zwei Elternteile an einem zufälligen Index i zerschnitten und die Endstücke getauscht werden (1-Punkt-Crossover). Beim 2-Punkt-Crossover werden beide Individuen in zwei Punkten zerschnitten und nur die Mittelstücke getauscht.[BHS07]

3 Konzept

3.1 Grundidee

Um eine Evolution sensomotorischer Kopplungen umsetzen zu können, musste zunächst eine Plattform entwickelt werden. Da im Bereich der künstlichen Intelligenz gerne die Assoziation zur Natur demonstriert wird ist es nahe liegend, natürliche Lebewesen als Vorbild zu betrachten. So entstand die Idee, eine Pflanze und ihr Streben nach dem Licht zu simulieren. Die zu optimierenden Parameter sind Motorstellungen, die die Ausrichtung der Blüte beeinflussen. Die Fitness ist in diesem Fall die Lichtintensität an der Blüte, also eine Größe, die mittels eines Photowiderstands gemessen werden kann.

3.2 Mechanischer Aufbau

Von Anfang an war klar, dass der Roboter mit Lego® konstruiert werden sollte, da dies einen schnellen und flexiblen Aufbau ermöglicht. Das erste Konzept beinhaltete einen Lichtsensor, welcher auf einer Konstruktion von drei um verschiedene Achsen rotierende Motoren installiert werden sollte. Dieses Konzept hätte jedoch hohe Anforderungen an die mechanische Konstruktion gestellt und würde sehr künstlich wirken. Das zweite Konzept beinhaltete einen flexiblen Stängel, dessen Ausrichtung durch drei von Motoren angetriebene Seilwinden beeinflusst werden sollte. Abbildung 1 zeigt, wie die Motoren dazu angeordnet sein sollten. Die gestrichelt dargestellten Fäden sollten die Blüte in eine bestimmte Position ziehen. Da Servomotoren, welche man in eine definierte Stellung hätte bringen können, aufgrund der physischen Belastung schnell verschleißen würden, wurden einfache Motoren verwendet, deren Zugkraft durch deren PWM-Wert[1] verändert werden könnte, wenn ein gewisser Widerstand dem Zug entgegen wirkt. Dieses Konzept erwies sich als leicht umsetzbar und hatte außerdem den Vorteil, dass das System einer natürlichen Blume sehr ähnlich sehen würde.

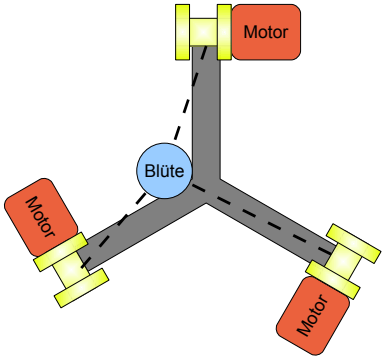


Abbildung 1: Anordnung der Motoren

[1] Pulsweitenmodulation: Steuerung der Motordrehzahl durch Veränderung des Tastverhältnisses eines Impulses bei konstanter Frequenz

3.3 Theoretische Vorbetrachtung

Da für den Anwendungszweck lediglich eine Parameteroptimierung und keine Erzeugung von Einzelprogrammen sinnvoll war, entschied sich der Autor für einen genetischen Algorithmus. Die Rahmenbedingungen waren, dass die Blume es mittels des Algorithmus schaffen muss, ihre Ausrichtung so zu verändern, dass eine möglichst optimale Lichtausbeute erreicht werden kann. Wenn diese Stellung einmal erreicht werden würde, sollte der Algorithmus sich jedoch nicht darauf beschränken, sondern stets weitere Stellungen testen, so dass die Blume in der Lage sein sollte, sich an eine veränderte Lichtsituation schnell anzupassen.

Die Fitness der Individuen soll die Lichtintensität sein. Bevor diese gemessen werden kann, müssen mechanische Teile bewegt werden. Aufgrund dessen ist der zeitaufwändigste Teil des entworfenen Algorithmus die Fitnessbestimmung. Daher entschied sich der Autor für eine steady-state-Abarbeitung, in der nur die Fitness neuer Individuen bestimmt wird. Während der Konzeption des Algorithmus wurde entschieden, ein Individuum durch ein Unsigned-Byte-Array der Form $\{A, B, C, F\}$ zu repräsentieren. Die Parameter A, B und C werden hier als ganze Zahlen kodiert und stellen die PWM-Werte der jeweiligen Motoren dar. Die Fitness F des Individuums ist ebenfalls eine ganze Zahl. Zunächst wurde es in Erwägung gezogen, das Alter des Individuums ebenfalls in dem Array zu speichern. Dies würde jedoch erfordern, dass das Alter aller Individuen regelmäßig aktualisiert werden muss, was zusätzlichen Rechenaufwand bedeutet hätte. Eine entsprechende Abarbeitung vorausgesetzt, kann das Alter eines Individuums auch durch seinen Index in der Population repräsentiert werden. Um dies zu ermöglichen, ohne die Elemente des Arrays verschieben zu müssen, wurde das Prinzip des Ringpuffers aufgegriffen, bei dem die Indizes des ersten und des letzten Elements zyklisch verschoben werden. In diesem Fall musste der Index des letzten Elements nicht gespeichert werden, da das Array sich in seiner Größe nicht verändert (konstante Populationsgröße). Somit steht der Index nach seiner Inkrementierung modulo der Populationsgröße immer auf dem ältesten Element, welches rigoros überschrieben wird.

Als genetische Operatoren wurden Kombination (Crossover) und Variation (Mutation) gewählt. Die Kombination erfolgt mittels einer geringfügig abgewandelten Form des 1-Punkt-Crossovers, bei dem jedoch nur ein Kind erzeugt wird. Die Variation findet in Form einer Integer-Mutation statt, welche die Parameter des Individuums um einen zufälligen Wert verändert. Abbildung 2 zeigt die Anwendung von 1-Punkt-Crossover und Mutation.

Der entworfene Algorithmus hat als Parameter die Wahrscheinlichkeiten P_c für Crossover und P_m für Mutation. Bei der Entscheidung, ob eine Mutation durchgeführt werden soll, spielt es keine Rolle, ob bereits ein Crossover stattgefunden hat. Die Entscheidung, ob ein einzelner Parameter mutiert werden soll, wird ebenfalls unabhängig von den anderen Parametern gefällt. Weitere Parameter des Algorithmus sind der Intervall, in dem mutiert werden kann sowie die Größe der Population, über die die Größe des Suchraums und indirekt auch der Selektionsdruck gesteuert werden können.

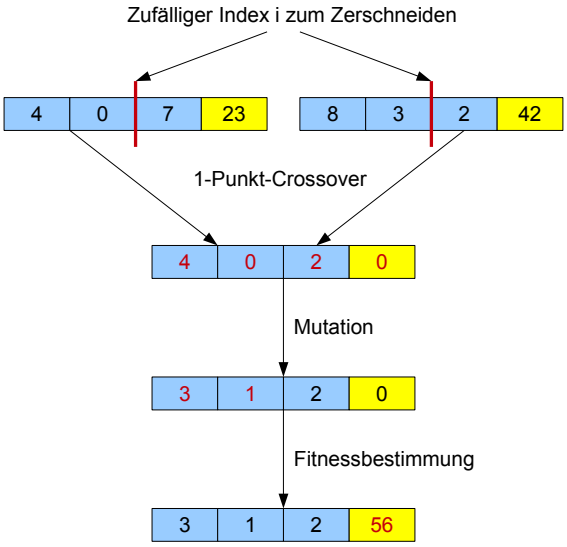
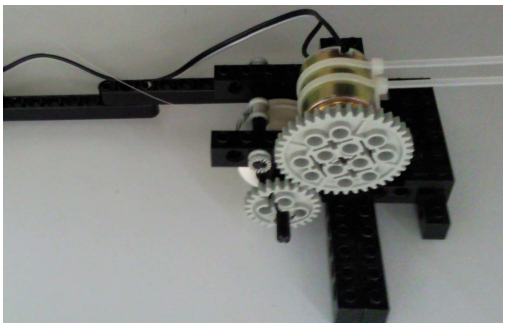


Abbildung 2: Genetische Operatoren auf ein Individuum angewendet

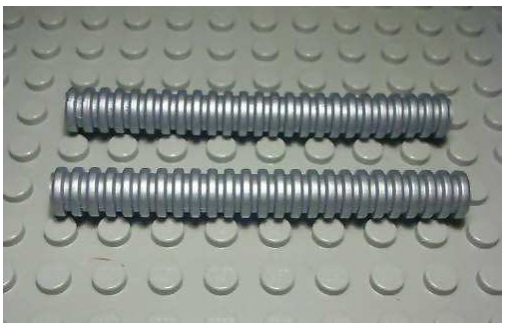
4 Realisierung

4.1 Konstruktion des Systems

Das System wurde, wie bereits erwähnt, mit Lego® Technic Bausteinen konstruiert. Lego® bietet einen sehr flexiblen Aufbau, bewegliche Teile, Antriebswellen, Zahnräder, und noch viel mehr. Als Motorik wurden zunächst drei Seilwinden wie in Abbildung 3(a) konstruiert. Für die Getriebe wurde eine Übersetzung von 5:3 gewählt, da es erforderlich ist, die gewünschte Endstellung so schnell wie möglich zu erreichen und nicht viel Kraft auszuüben. Es wurden Zahnräder mit 40 und 24 Zähnen verbaut. Anschließend wurden die Seilwinden in einer sternförmigen Konstruktion miteinander verbunden. Der Stängel der Blume wurde aus zwei Lego® Flex Rohren wie in Abbildung 3(b) gebaut.



(a) Motorgetriebene Seilwinde



(b) Lego® Flex Rohr

Abbildung 3: Konstruktion mit Lego®

Die „Blüte“ wurde optisch einer realen Blüte nachempfunden. Da sich der Photowiderstand in den ersten Versuchen als sehr lichtempfindlich erwies, wurde er mit einem gelben Klebstreifen und Fotofilm überklebt. Abbildung 4 zeigt die fertig montierte Blume.

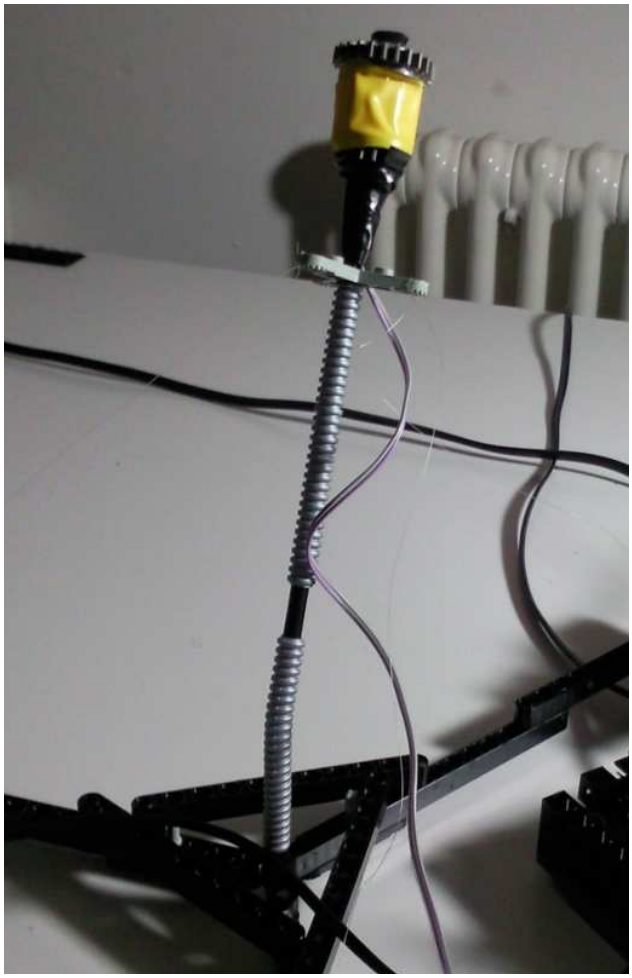


Abbildung 4: Fertig konstruierte Blume

Als Mikrocontroller-Plattform wurde das von der Fachhochschule Brandenburg entwickelte AKSEN-Board[2] verwendet. Ausgangsseitig wurden die drei Motoren angeschlossen sowie eine LED zur Anzeige der Fitnessbestimmung. Eingangsseitig wurde zunächst die Photodiode angeschlossen. Später kamen noch vier Taster hinzu, über die die Parameter des Algorithmus zur Laufzeit eingestellt werden können. Dies wird im nächsten Kapitel noch einmal genauer erläutert. Zur Stabilisierung wurde das AKSEN-Board zusammen mit den Tastern in ein Gehäuse aus Lego®-Steinen fest integriert. Dies soll auch das Abrutschen der angeschlossenen Sensoren- und Aktorenkabel verhindern.

[2] `http://ots.fh-brandenburg.de/aksen`

4.2 Umsetzung des Algorithmus

Der Algorithmus wurde mittels imperativer Programmierung in der Programmiersprache C umgesetzt. Die Population wird von einem zweidimensionalen Array repräsentiert, eine Dimension sind die Individuen, die zweite Dimension stellen die Parameter und der Fitnesswert jedes Individuums dar. Abbildung 5 zeigt den Algorithmus in Form eines Programmablaufplans.

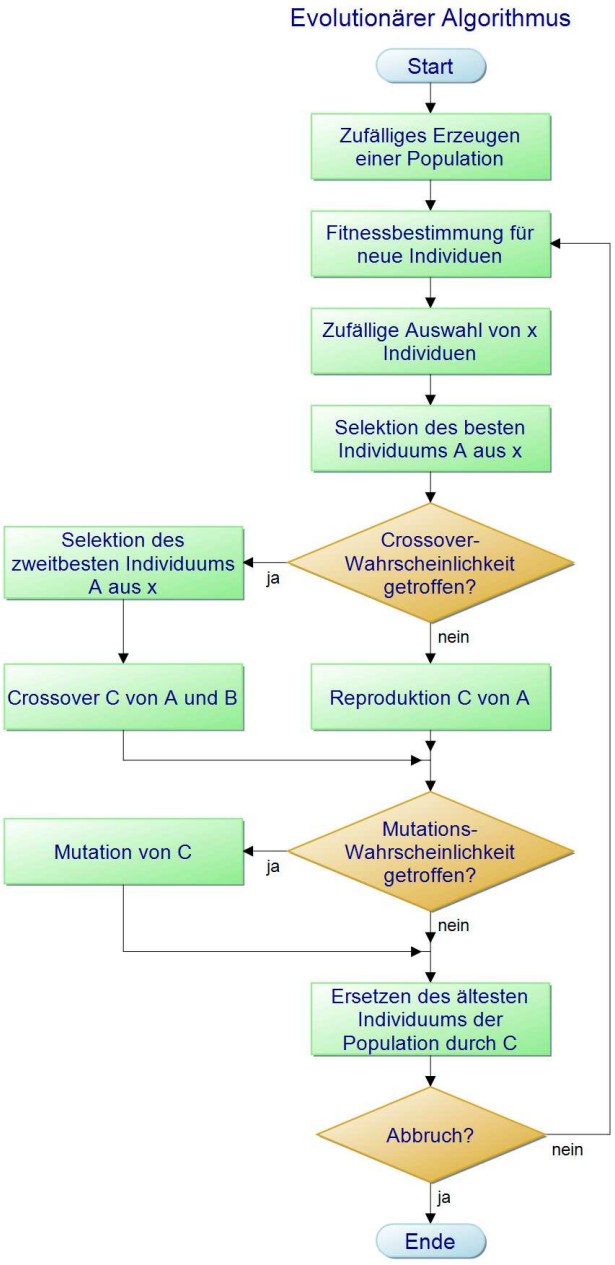


Abbildung 5: Vereinfachter Programmablaufplan des Algorithmus

Nach dem Start des Programms wird die Population initialisiert, indem sie mit Individuen mit zufälligen Parametern befüllt wird. Anschließend erfolgt eine Fitnessbestimmung für alle Individuen. Die Motoren werden für einige Millisekunden entsprechend den Parametern des jeweiligen Individuums angesteuert, so dass sich die Blume nach dem Licht ausrichtet. An dieser Stelle wird eine Messung der Lichtintensität vorgenommen. In ersten Versionen wurde dieser Wert bereits unver-

ändert als Fitness übernommen. Später ergab sich jedoch das Bedürfnis, nicht nur nach Helligkeit, sondern auch nach geringem Stromverbrauch zu optimieren. Aus diesem Grund wurde die Verbesserung eingeführt, die Summe der PWM-Werte der Motoren von der Fitness zu subtrahieren. Folglich wird die Fitness bestimmt durch die Formel:

$$F = Lichtintensität - (A + B + C)$$

Ein hoher Stromverbrauch, der mit hohen PWM-Werten einher geht, vermindert also die Fitness. Durch dieses Prinzip konnte die Effizienz des Systems nochmals gesteigert werden. Fällt das Licht senkrecht von oben auf die Blume, muss diese die Motoren nun nicht mehr ansteuern, denn dies würde die Fitness vermindern, da die Helligkeit nicht weiter ansteigt.

In der darauf folgenden Selektionsphase wird eine vordefinierte Anzahl von Individuen aus der Population zufällig ausgewählt, in die engere Auswahl zu kommen. Anschließend werden auf das beste Individuum (bei Mutation) oder die besten zwei Individuen (bei Crossover) die genetischen Operatoren angewendet. Dieses Selektionsverfahren wird als Turnierselektion bezeichnet. [BHS07] Es wird eine Turniergruppe gebildet und die besten Individuen in diesem Turnier „gewinnen“ immer, werden also immer selektiert. Nach der Selektion erfolgt die Anwendung der genetischen Operatoren, wenn die entsprechende Wahrscheinlichkeit getroffen wird. Wenn keine der Wahrscheinlichkeiten getroffen wird, ist das neue Individuum eine genaue Kopie des selektierten Individuums. Am Ende des Schleifendurchlaufs ersetzt das neue Individuum das älteste in der Population.

Zur Aufzeichnung der Fitnesskurven in den Versuchen musste eine Möglichkeit geschaffen werden, Werte vom AKSEN-Board an den PC zu übertragen. Zu diesem Zweck wurde Funktion `serielle_putchar()` der Bibliothek `serielle.h` verwendet. Mit ihr wird die maximale Fitness der Population bei der Erzeugung eines neuen Individuums über die serielle Schnittstelle ausgegeben. Diese kann am PC mittels einer Software wie HyperTerminal ausgelesen und gespeichert werden. Da die AKSEN-Bibliothek über keine Zufallsfunktion verfügt, wurde eine Pseudo-Zufallszahlen-Implementierung aus den Linux man pages[3] übernommen.

Der Quelltext des gesamten Programms befindet sich in Anhang A.

4.3 Konfiguration des Algorithmus

Der entwickelte Algorithmus verfügt über eine Reihe von Parametern, die die Flexibilität und die Geschwindigkeit, mit der ein Optimum erreicht werden kann, deutlich beeinflusst. Tabelle 1 fasst diese zusammen. Die Auswirkung dieser Parameter in Abhängigkeit der Parameter ist in Anhang B genauer erkennbar.

Um die Parameter des Algorithmus verändern zu können, ohne das Programm neu kompilieren zu müssen, wurde ein Konfigurationsmenü geschrieben. Dieses ist durch die vier Tasten auf Abbildung 6 zu bedienen. Taste A ruft den nächsten Parameter auf, Taste B den vorhergehenden. Mit den Tasten C und D kann der Wert des ausgewählten Parameters erhöht oder verringert werden.

[3] `http://linux.die.net/man/3/rand`

Parameter	Beschreibung	Wertebereich	Standardwert
p_crossover	Wahrscheinlichkeit für Crossover in %	[0,100]	20
p_mutate	Wahrscheinlichkeit für Mutation in %	[0,100]	15
max_mutate	Maximaler Mutationsintervall.Bei max_mutate=5 kann der neue Wert um ± 2.5 vom Ausgangswert abweichen	[0,10]	5
pop_size	Populationsgröße	[5,200]	15
best_of_x	Turniergröße bei Selektion	[3,200]	3
wait_time	Länge der Wartezeit bei Fitnessbestimmung in ms	[100,5000]	200
optimize_power	Energieverbrauch in die Fitnessbestimmung mit einfließen lassen	{0,1}	1
move_back	Nach Fitnessbestimmung zurück in Mittelposition gehen	{0,1}	0

Tabelle 1: Auflistung der Parameter des Algorithmus

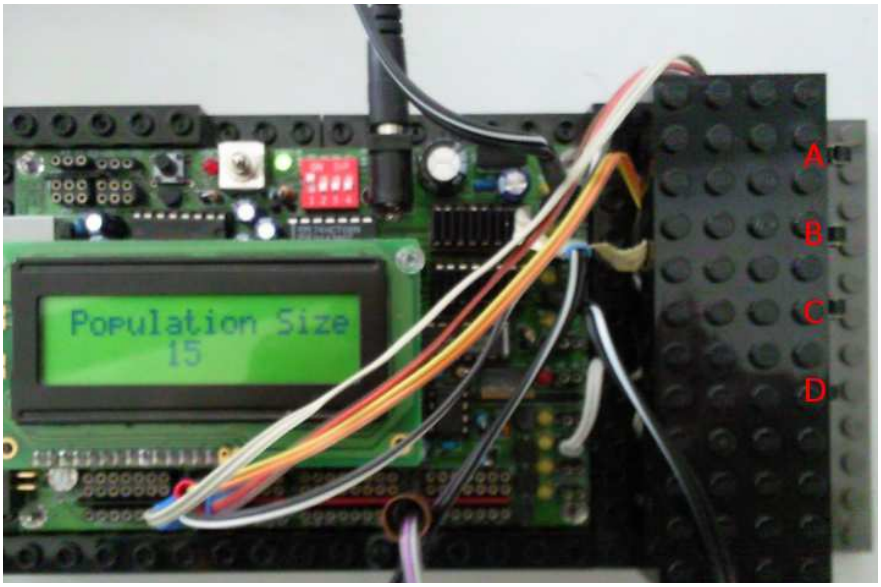


Abbildung 6: Taster zur Konfiguration

5 Funktionsnachweis

5.1 Versuche

Zum Nachweis der Funktion des Systems wurden Lernkurven mit verschiedenen Einstellungen ermittelt. Diese sind in Anhang B zusammengefasst. Die Funktion des genetischen Algorithmus wird dadurch nachgewiesen, dass die Fitnesskurve in jedem Versuch ansteigt. Dies bedeutet, dass das System lernt, also seine maximale Fitness über die Zeit verbessert. Werden die Parameter ungünstig gewählt oder verändert sich die Umwelt, kann die Fitness jedoch auch nachlassen, wie beispielsweise in Abbildung 11 zu sehen ist. Ein Video, das während der Versuche aufgenommen wurde, kann auf Youtube[4] angesehen werden.

[4] `http://www.youtube.com/watch?v=JQeTEmwHZ2k`

6 Zusammenfassung und Ausblick

6.1 Fazit

Im Rahmen des Projektes konnte ein Roboter in Form einer künstlichen Blume konstruiert werden und dafür ein funktionierender genetischer Algorithmus implementiert werden. Gegenüber einer herkömmlichen Regelung ist dieser vermutlich auch in der Lage, sich an Veränderungen wie den Verschleiß der technischen Komponenten anzupassen, beispielsweise eine nachlassende Motorleistung. Das verwendete Lego® Flex Rohr hat die Eigenschaft, die Biegung beizubehalten wenn es permanent in eine Richtung gebogen wird. Dies wäre für eine Regelung eine unkontrollierbare Störung, der genetische Algorithmus kann dies jedoch kompensieren. Darüber hinaus hat die Verformung sogar Vorteile, da die Motoren dann weniger, oder in der Endstellung überhaupt nicht mehr angesteuert werden müssen. Der Algorithmus sollte ebenfalls in der Lage sein, das Vertauschen von zwei Motoranschlüssen auszugleichen, denn es sollte vollkommen unerheblich sein, wo welcher Motor angeschlossen ist. Fällt ein Motor komplett aus, wird der Algorithmus vermutlich versuchen, mit den noch verbliebenen Motoren das bestmöglichste Ergebnis zu erzielen. Diese Anpassungen an die Umwelt, sowie an Verschleiß der Komponenten konnte aufgrund von Zeitmangel leider nicht mehr durch Versuche und entsprechende Messungen belegt werden. Daher sind die Aussagen darüber als hypothetisch anzusehen.

Einen großen Vorteil brachte die Veränderung, dass die PWM-Werte der Motoren mit in die Fitness einfließen. Eventuell könnte dies durch einen Faktor noch optimiert werden.
Das Konfigurationsmenü, welches implementiert wurde, erwies sich als sehr nützlich, da die Parameter nun schnell und einfach konfiguriert werden können, um schnell ein gutes Setup zu finden. Letztendlich gibt es keine Parameter, die für alle Umgebungen und Anforderungen gleich gut geeignet sind, daher müssen sie angepasst werden. In einer weiterführenden Arbeit könnte untersucht werden, wie eine dynamische Parameteranpassung implementiert und der Algorithmus so noch verbessert werden könnte. Ändert sich beispielsweise die Lichtsitation und fällt die Fitness dadurch rapide ab, wäre es sinnvoll, die Mutationsrate zu erhöhen, um den Suchraum zu vergrößern.
Als Crossover-Operator sollte besser das herkömmliche 1-Punkt-Crossover verwendet werden, bei dem zwei Nachkommen erzeugt werden. Die hier verwendete Variante mit nur einem Nachkommen könnte zu einem Ungleichgewicht führen, da die ersten Parameter des Nachkommens immer vom Individuum mit der höheren Fitness, und die letzten vom Individuum mit der geringeren Fitness übernommen werden. Des Weiteren könnten noch weitere Selektionsvarianten implementiert und verglichen werden.

Tabellenverzeichnis

Abbildungsverzeichnis

Literatur

[BHS07] Boersch, Ingo; Heinsohn, Jochen; Socher, Rolf: *Wissensverarbeitung*, 2. Auflage, Spektrum Akademischer Verlag, 2007. ISBN 978-3-8274-1844-9

[Dar99] Darwin, Charles: *The Origin of Species by means of Natural Selection*, 6. Auflage, Project Gutenberg, 1999. `http://www.gutenberg.org/ebooks/2009`

[LäCl01] Lämmel, Uwe; Cleve, Jürgen: *Lehr- und Übungsbuch Künstliche Intelligenz*, Fachbuchverlag Leipzig, 2001. ISBN 3-446-21421-6

A Quelltext des Programms

```
1  /**
2   * GeneFlower
3   *
4   * Author: Stephan Dreyer
5   * Last modified: 2010-06-04 18:00
6   *
7   * This is an AkSen board program for a flower to face the light.
8   * It uses an genetic algorithm that controls 3 electric motors
        to
9   * maximize the fitness. More light means more fitness.
10  * In this algorithm, every individual of the population consists
11  * of 3 pwm values for the motors and the fitness of the
        individual.
12  *
13  * The parameters for the algorithm can be configured over a menu
14  * when the program is running (no recompilation necessary).
15  *
16  *   ACTORS
17  *     ATT_MOT1_PWM     motor on port 0
18  *     ATT_MOT2_PWM     motor on port 1
19  *     ATT_MOT3_PWM     motor on port 2
20  *
21  *     LED          led on led 1
22  *
23  *   SENSORS
24  *     PHOTO_SENSOR         photo sensor on analog 0
25  *
26  *     BUTTON_UP      button on digital 0
27  *     BUTTON_DOWN      button on digital 1
28  *     BUTTON_PLUS      button on digital 2
29  *     BUTTON_MINUS     button on digital 3
30  *
31  *   DIP SWITCH
32  *     CONFIGURATION MODE   switch on digital 4
33  *     RESET POPULATION   switch on digital 5
34  *     PHOTODIODE TEST    switch on digital 6
35  *     REST AT BEST     switch on digital 7
36  */
37
38 #include <stdio.h>
39 #include <stdlib.h>
40 #include <time.h>
41
42 //#define AKSEN
43
44 #ifdef AKSEN
```

```
45 #define DEBUG
46
47 // aksen library
48 #include <regc515c.h>
49 #include <stub.h>
50 #include "serielle.h"
51 #endif
52
53 // SOME MACROS FOR EASIER HANDLING
54 #define FALSE 0
55 #define TRUE 1
56 #define PHOTO_SENSOR analog(0)
57 #define LED(value) led(1, value)
58 #define BUTTON_UP !digital_in(0)
59 #define BUTTON_DOWN !digital_in(1)
60 #define BUTTON_PLUS !digital_in(2)
61 #define BUTTON_MINUS !digital_in(3)
62 #define SWITCH_0 !dip_pin(0)
63 #define SWITCH_1 !dip_pin(1)
64 #define SWITCH_2 !dip_pin(2)
65 #define SWITCH_3 !dip_pin(3)
66 #define FORWARD 0
67 #define BACKWARD 1
68 #define UNINITIALIZED 255
69
70 // FOR THE SETTINGS MENU
71 #define SETTING_P_CROSSOVER 0
72 #define SETTING_P_MUTATE 1
73 #define SETTING_MAX_MUTATE 2
74 #define SETTING_POP_SIZE 3
75 #define SETTING_BEST_OF_X 4
76 #define SETTING_WAIT_TIME 5
77 #define SETTING_OPTIMIZE_POWER 6
78 #define SETTING_MOVE_BACK 7
79 #define SETTINGS_COUNT 8
80
81 // ATTRIBUTES OF AN INDIVIDUAL
82 #define ATT_MOT1_PWM 0
83 #define ATT_MOT2_PWM 1
84 #define ATT_MOT3_PWM 2
85 #define FITNESS 3
86 #define ATT_SIZE 4
87
88 // FOR TESTING
89 // number of generations to calculate
90 #define TEST_GENERATIONS 10
91
92 // maximum population size
```

```
93 #define MAX_POP_SIZE 200
94
95 // GENETIC ALGORITHM PARAMETERS
96 // probability for crossover in percent
97 unsigned char p_crossover=20;
98 // probability for mutation in percent
99 unsigned char p_mutate=15;
100 // maximum mutation impact
101 unsigned char max_mutate=5;
102 // total size of population
103 unsigned char pop_size=15;
104 // value for best-of-x selection
105 unsigned char best_of_x=3;
106
107 // FITNESS FUNCTION PARAMETERS
108 // wait time for photo sensor
109 unsigned int wait_time=200;
110 // power consumtion dependent fitness
111 unsigned char optimize_power=1;
112 // move back between steps
113 unsigned char move_back=0;
114
115 // WHERE THE NEXT INDIVIDUAL SHOULD BE WRITTEN
116 unsigned char writeIndex = 0;
117
118 // THE POPULATION CONTAINS ALL INDIVIDUALS
119 unsigned char population[MAX_POP_SIZE][ATT_SIZE];
120
121 // THE SELECTION ALGORITHM REQUIRES THAT SELECTED INDIVIDUALS
122 // WILL BE STORED SO THEY CAN NOT BE SELECTED AGAIN
123 unsigned char selected_individuals[MAX_POP_SIZE];
124
125 // FUNCTIONS THAT ARE MISSING IN THE AKSEN LIB
126 #ifdef AKSEN
127 #define min(a, b) a<b?a:b
128 #define max(a, b) a>b?a:b
129
130 /*
131  * the following random - function has been taken from
132  * the rand(3) Linux man page located at http://linux.die.net/man
       /3/rand
133  */
134 unsigned long int next = 1;
135
136 int rand(void) {
137   next=next * 1103515245 + 12345;
138   return (unsigned int)(next/65536) % 32768;
139 }
```

```
140
141 void srand(unsigned int a) {
142   next = a;
143 }
144 /*
145  * end of random - function
146  */
147
148 void printIndividual(unsigned char idx){
149   // print information for the individual
150   lcd_cls();
151   lcd_setxy(0,1);
152   lcd_puts("[");
153   lcd_ubyte(population[idx][ATT_MOT1_PWM]);
154   lcd_puts("␣");
155   lcd_ubyte(population[idx][ATT_MOT2_PWM]);
156   lcd_puts("␣");
157   lcd_ubyte(population[idx][ATT_MOT3_PWM]);
158   lcd_puts("]");
159   lcd_setxy(1,1);
160   lcd_puts("FITNESS:␣");
161   lcd_ubyte(population[idx][FITNESS]);
162 }
163
164 void faceTo(unsigned char idx) {
165   motor_richtung(ATT_MOT1_PWM,FORWARD);
166   motor_richtung(ATT_MOT2_PWM,FORWARD);
167   motor_richtung(ATT_MOT3_PWM,FORWARD);
168
169   motor_pwm(ATT_MOT1_PWM,population[idx][ATT_MOT1_PWM]);
170   motor_pwm(ATT_MOT2_PWM,population[idx][ATT_MOT2_PWM]);
171   motor_pwm(ATT_MOT3_PWM,population[idx][ATT_MOT3_PWM]);
172 }
173
174 void stop(){
175   motor_pwm(ATT_MOT1_PWM,FALSE);
176   motor_pwm(ATT_MOT2_PWM,FALSE);
177   motor_pwm(ATT_MOT3_PWM,FALSE);
178 }
179
180 void faceBack(unsigned char idx) {
181   // go back with half speed
182   motor_pwm(ATT_MOT1_PWM,population[idx][ATT_MOT1_PWM]/2);
183   motor_pwm(ATT_MOT2_PWM,population[idx][ATT_MOT2_PWM]/2);
184   motor_pwm(ATT_MOT3_PWM,population[idx][ATT_MOT3_PWM]/2);
185
186   // rotate back to center the flower
187   motor_richtung(ATT_MOT1_PWM,BACKWARD);
```

```
188   motor_richtung(ATT_MOT2_PWM,BACKWARD);
189   motor_richtung(ATT_MOT3_PWM,BACKWARD);
190
191   sleep(10);
192
193   // go back with quarter speed
194   motor_pwm(ATT_MOT1_PWM,population[idx][ATT_MOT1_PWM]/4);
195   motor_pwm(ATT_MOT2_PWM,population[idx][ATT_MOT2_PWM]/4);
196   motor_pwm(ATT_MOT3_PWM,population[idx][ATT_MOT3_PWM]/4);
197
198   sleep(10);
199
200   stop();
201 }
202
203 void faceToBest(){
204   unsigned int i, max = 0;
205
206   // FIND MAXIMUM FITNESS
207   for (i = 0; i < pop_size; i++){
208     if (population[max][FITNESS] < population[i][FITNESS] &&
          population[i][FITNESS] != UNINITIALIZED) {
209       max = i;
210     }
211   }
212
213   printIndividual(max);
214
215   faceTo(max);
216
217   while (SWITCH_3);
218
219   faceBack(max);
220 }
221 #endif
222
223 void select(unsigned char index) {
224   selected_individuals[index] = TRUE;
225 }
226
227 char wasSelected(unsigned char index) {
228   return selected_individuals[index];
229 }
230
231 void resetSelection(){
232   unsigned char i;
233   for (i = 0; i < pop_size; i++){
234     selected_individuals[i] = FALSE;
```

```
235   }
236 }
237
238
239 // fitness calculation
240 void calcFitness(){
241   unsigned char pop;
242   #ifdef DEBUG
243   unsigned char max = 0;
244   char str[11];
245   unsigned char i;
246
247   #endif
248
249   for (pop = 0; pop < pop_size; pop++) {
250     int fitness = population[pop][FITNESS];
251
252     #ifdef DEBUG
253     if (population[max][FITNESS] < population[pop][FITNESS] &&
          population[pop][FITNESS] != UNINITIALIZED) {
254       max = pop;
255     }
256     #endif
257
258     // fitness=UNINITIALIZED means it hasn't been evaluated
259     if (fitness == UNINITIALIZED){
260       #ifdef AKSEN
261
262       // turn light on to indicate that a new individual has now
             its turn
263       LED(TRUE);
264
265       faceTo(pop);
266
267       sleep(wait_time);
268
269       // turn light off
270       LED(FALSE);
271
272       // get fitness from sensor
273       fitness = 255 - PHOTO_SENSOR;
274
275       // fitness=255 means it hasn't been evaluated, maximum
             fitness can be 254
276       fitness = min(fitness, 254);
277
278       if (optimize_power){
279         unsigned char power = population[pop][ATT_MOT1_PWM] +
```

```
            population[pop][ATT_MOT2_PWM] + population[pop][
            ATT_MOT3_PWM];
280         fitness = max( (int)fitness - (int)power,0 );
281       }
282
283       if (move_back){
284         faceBack(pop);
285       }
286
287       #else
288
289       // mathmatical fitness function for testing
290       fitness = max(population[pop][0] + population[pop][1] -
             population[pop][2],0);
291
292       #endif
293
294       population[pop][FITNESS] = fitness;
295
296       #ifdef AKSEN
297       printIndividual(pop);
298       #endif
299     }
300   }
301
302   #ifdef DEBUG
303   // debug output over the serial bus
304   sprintf(str, "%3d;", population[max][FITNESS]);
305
306   for (i = 0; i < 4; i++){
307     serielle_putchar(str[i]);
308   }
309
310   #endif
311 }
312
313 void printPopulation(){
314   #ifndef AKSEN
315   unsigned char pop, att;
316
317   printf("\nPopulation:\n");
318
319   for (pop = 0; pop < pop_size; pop++) {
320     printf ("␣[");
321     for (att = 0; att < ATT_SIZE; att++) {
322       printf ("%2d␣", population[pop][att]);
323     }
324     printf ("]\n");
```

```
325   }
326   #endif
327 }
328
329 void initPopulation(){
330   unsigned char pop, att;
331
332   srand((unsigned char)time(NULL));
333
334   #ifndef AKSEN
335   printf("Settings\n\n");
336   printf("␣Crossover␣Poss\n");
337   printf("␣␣␣␣␣%3d%%\n", p_crossover);
338   printf("␣Mutation␣Poss\n");
339   printf("␣␣␣␣␣%3d%%\n", p_mutate);
340   printf("␣Mutation␣Impact\n");
341   printf("␣␣␣␣␣␣%2d\n", max_mutate);
342   printf("␣Population␣Size\n");
343   printf("␣␣␣␣␣%3d\n", pop_size);
344   printf("Initializing␣population␣with␣%d␣individuals", pop_size)
        ;
345   #endif
346
347   for (pop = 0; pop < pop_size; pop++) {
348     for (att = 0; att < ATT_SIZE; att++) {
349       unsigned char rnd;
350
351       if (att == FITNESS) {
352         rnd = UNINITIALIZED;
353       } else {
354         rnd = (unsigned char) (rand() % 11);
355       }
356       population[pop][att] = rnd;
357     }
358   }
359
360   calcFitness();
361   printPopulation();
362 }
363
364 int isMutation(){
365   return (rand() % 101) <= p_mutate;
366 }
367
368 int isCrossover(){
369   return (rand() % 101) <= p_crossover;
370 }
371
```

```
372 void mutate(int idx){
373   unsigned char att;
374 
375   #ifndef AKSEN
376   printf ("Mutated [");
377   for (att = 0; att < ATT_SIZE; att++) {
378     printf ("%2d ", population[idx][att]);
379   }
380   printf ("] to [");
381   #endif
382 
383   for (att = 0; att < ATT_SIZE-1; att++) {
384     char newAtt = population[idx][att];
385     int rnd = (rand() % (max_mutate*2))-max_mutate;
386 
387     if (isMutation()){
388       newAtt = newAtt + (char)rnd;
389       newAtt = max(newAtt,0);
390       newAtt = min(newAtt,10);
391 
392       population[idx][att] = newAtt;
393     }
394 
395     #ifndef AKSEN
396     printf ("%2d ", population[idx][att]);
397     #endif
398   }
399   population[idx][FITNESS] = UNINITIALIZED;
400 
401   #ifndef AKSEN
402   printf (" %d ]\n", population[idx][FITNESS]);
403   #endif
404 }
405 
406 void crossover(int idx, int source1, int source2) {
407   unsigned char att;
408   unsigned char cut = rand() % ATT_SIZE-1;
409   unsigned char cross = isCrossover();
410 
411   for (att = 0; att < ATT_SIZE-1; att++) {
412     if (att < cut || !cross){
413       population[idx][att] = population[source1][att];
414     } else {
415       population[idx][att] = population[source2][att];
416     }
417   }
418   population[idx][FITNESS] = UNINITIALIZED;
419 
```

```
420   #ifndef AKSEN
421   if (cross){
422     printf ("Crossover␣of␣[");
423     for (att = 0; att < ATT_SIZE; att++) {
424       printf ("%2d␣", population[source1][att]);
425     }
426     printf ("]␣and␣[");
427     for (att = 0; att < ATT_SIZE; att++) {
428       printf ("%2d␣", population[source2][att]);
429     }
430     printf ("]␣is␣[");
431     for (att = 0; att < ATT_SIZE; att++) {
432       printf ("%2d␣", population[idx][att]);
433     }
434     printf ("]\n");
435   }
436   #endif
437 }
438
439 /*
440  * This function is the main part of the genetic algorithm.
441  * Selection, reproduction, crossover and mutation are done
442  * here.
443  */
444 void spawn(){
445   unsigned char rnd, i=0, max=UNINITIALIZED, max2=UNINITIALIZED;
446
447   resetSelection();
448
449   // select, until x random individuals are selected
450   while (i < best_of_x) {
451     unsigned char fitness;
452     rnd = rand() % pop_size;
453
454     if (!wasSelected(rnd)) {
455       select(rnd);
456       fitness = population[rnd][FITNESS];
457
458       if (max==UNINITIALIZED){
459         max = rnd;
460       } else if (max2==UNINITIALIZED) {
461         max2 = rnd;
462       } else if (fitness >= population[max][FITNESS]){
463         if (population[max][FITNESS] > population[max2][FITNESS])
              {
464           max2 = max;
465         }
466         max = rnd;
```

```
467       } else if (fitness >= population[max2][FITNESS] && max !=
              rnd) {
468         max2 = rnd;
469       }
470       #ifndef AKSEN
471       printf ("Selected␣%d␣for␣best␣of␣%d␣challenge.␣Fitness␣is:␣
              %d\n", rnd, best_of_x, fitness);
472       #endif
473
474       i++;
475     }
476   }
477
478   #ifndef AKSEN
479   printf ("Max␣␣fitness␣at␣index␣%d:␣%d\n", max, population[max][
         FITNESS] );
480   printf ("Max2␣fitness␣at␣index␣%d:␣%d\n", max2, population[max2
         ][FITNESS] );
481   #endif
482
483   crossover(writeIndex, max, max2);
484   mutate(writeIndex);
485
486   writeIndex = ++writeIndex % pop_size;
487 }
488
489 void run(){
490   spawn();
491   calcFitness();
492   printPopulation();
493 }
494
495 #ifdef AKSEN
496 void configure(){
497   char setting = SETTING_P_CROSSOVER;
498
499   while (SWITCH_0) {
500     if (BUTTON_DOWN){
501       setting = min ((unsigned char)(setting-1), SETTINGS_COUNT
              -1);
502     } else if (BUTTON_UP){
503       setting = min ((unsigned char)(setting+1), SETTINGS_COUNT
              -1);
504     } else if (BUTTON_PLUS){
505       switch (setting){
506         case SETTING_P_CROSSOVER:
507           p_crossover=++p_crossover % 101;break;
508         case SETTING_P_MUTATE:
```

```
509           p_mutate=++p_mutate % 101;break;
510         case SETTING_MAX_MUTATE:
511           max_mutate=++max_mutate % 11;break;
512         case SETTING_POP_SIZE:
513           pop_size=(pop_size % 200)+5;break;
514         case SETTING_OPTIMIZE_POWER:
515           optimize_power=!optimize_power;break;
516         case SETTING_BEST_OF_X:
517           best_of_x = ++best_of_x % (pop_size+1);break;
518         case SETTING_WAIT_TIME:
519           wait_time =(wait_time + 100) % 5001;break;
520         case SETTING_MOVE_BACK:
521           move_back=!move_back;break;
522       }
523
524
525     } else if (BUTTON_MINUS){
526       switch (setting){
527         case SETTING_P_CROSSOVER:
528           p_crossover = --p_crossover % 101;break;
529         case SETTING_P_MUTATE:
530           p_mutate = --p_mutate % 101;break;
531         case SETTING_MAX_MUTATE:
532           max_mutate = --max_mutate % 11;break;
533         case SETTING_POP_SIZE:
534           if ((int)pop_size-5 <= 0){
535             pop_size = 5;
536           } else {
537             pop_size = pop_size-5;
538           }
539           best_of_x = min(best_of_x, pop_size);break;
540         case SETTING_OPTIMIZE_POWER:
541           optimize_power=!optimize_power;break;
542         case SETTING_BEST_OF_X:
543           best_of_x = max(best_of_x--,3);break;
544         case SETTING_WAIT_TIME:
545           wait_time = max(wait_time-100, 100);break;
546         case SETTING_MOVE_BACK:
547           move_back=!move_back;break;
548       }
549     }
550
551
552     lcd_cls();
553     lcd_setxy(0,1);
554
555     // print the menu
556     switch (setting){
```

```
557         case SETTING_P_CROSSOVER:
558           lcd_puts("Crossover␣Prob");
559           lcd_setxy(1,1);
560           lcd_puts("␣␣␣␣␣");
561           lcd_ubyte(p_crossover);
562           lcd_puts("%");
563           break;
564         case SETTING_P_MUTATE:
565           lcd_puts("Mutation␣Prob");
566           lcd_setxy(1,1);
567           lcd_puts("␣␣␣␣␣");
568           lcd_ubyte(p_mutate);
569           lcd_puts("%");
570           break;
571         case SETTING_MAX_MUTATE:
572           lcd_puts("Mutation␣Impact");
573           lcd_setxy(1,1);
574           lcd_puts("␣␣␣␣␣");
575           lcd_ubyte(max_mutate);
576           break;
577         case SETTING_POP_SIZE:
578           lcd_puts("Population␣Size");
579           lcd_setxy(1,1);
580           lcd_puts("␣␣␣␣␣");
581           lcd_ubyte(pop_size);
582           break;
583         case SETTING_BEST_OF_X:
584           lcd_puts("Best␣of␣x");
585           lcd_setxy(1,1);
586           lcd_puts("␣␣␣␣␣");
587           lcd_ubyte(best_of_x);
588           break;
589         case SETTING_WAIT_TIME:
590           lcd_puts("Motor␣run␣time");
591           lcd_setxy(1,1);
592           lcd_puts("␣␣␣␣␣");
593           lcd_uint(wait_time);
594           break;
595         case SETTING_OPTIMIZE_POWER:
596           lcd_puts("Optimize␣Power");
597           lcd_setxy(1,1);
598           lcd_puts("␣␣␣␣␣");
599           lcd_ubyte(optimize_power);
600           break;
601         case SETTING_MOVE_BACK:
602           lcd_puts("Move␣back");
603           lcd_setxy(1,1);
604           lcd_puts("␣␣␣␣␣");
```

```
605           lcd_ubyte(move_back);
606           break;
607       }
608
609       sleep(200);
610     }
611 }
612
613 void testir(){
614   lcd_cls();
615   lcd_setxy(0,1);
616   lcd_ubyte(PHOTO_SENSOR);
617   sleep(500);
618 }
619
620 void AksenMain(void) {
621   unsigned char reset = TRUE;
622
623   // show welcome message
624   lcd_cls();
625   lcd_setxy(0,1);
626   lcd_puts("␣␣GeneFlower␣␣␣");
627   lcd_setxy(1,1);
628   lcd_puts("will␣start␣soon");
629   sleep(800);
630
631   lcd_cls();
632   lcd_setxy(0,1);
633   lcd_ubyte(SWITCH_0);
634   lcd_puts("|");
635   lcd_ubyte(SWITCH_1);
636   lcd_puts("|");
637   lcd_ubyte(SWITCH_2);
638   lcd_puts("|");
639   lcd_ubyte(SWITCH_3);
640
641   sleep(200);
642   serielle_init();
643
644   while (TRUE) {
645     if (SWITCH_0) {
646       stop();
647       reset = TRUE;
648       configure();
649     } else if (SWITCH_2){
650       stop();
651       testir();
652     } else if (SWITCH_3 || BUTTON_DOWN || BUTTON_UP ||
```

```
          BUTTON_PLUS || BUTTON_MINUS){
653         faceToBest();
654       } else {
655         // population reset
656         if (SWITCH_1 || reset){
657           reset = FALSE;
658           initPopulation();
659         }
660         run();
661       }
662     }
663 }
664 #else
665 int main(int argc, char* argv){
666   int i;
667
668   initPopulation();
669
670   for (i = 0; i < TEST_GENERATIONS; i++){
671     printf("Generation␣%d\n", i+1);
672     run();
673   }
674   getchar();
675
676   return 0;
677 }
678 #endif
```

B Lernkurven

Die folgenden Abbildungen zeigen die Lernkurven, also den Anstieg der höchsten Fitness einer Population in Abhängigkeit der Parameter des Algorithmus.

Abbildung 7 zeigt die Lernkurve mit den Parametern p_crossover=20, p_mutate=15, max_mutate=5, pop_size=15, optimize_power=1. Mit diesen Werten konnte meistens schnell ein Optimum gefunden werden.

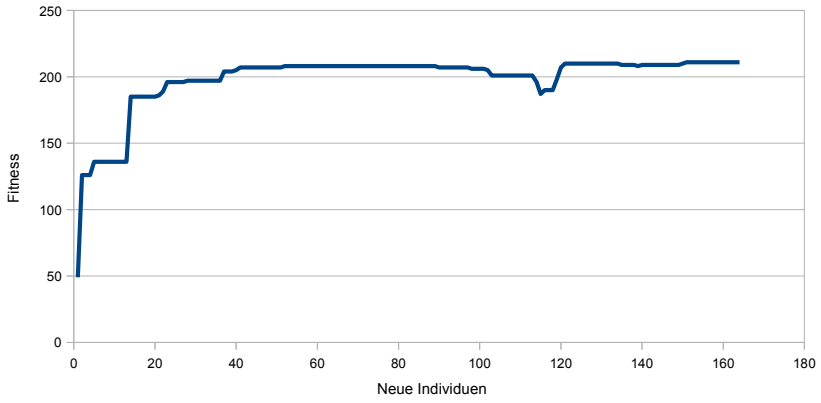


Abbildung 7: Lernkurve mit den Standardwerten

Abbildung 8 zeigt die Lernkurve mit den Parametern p_crossover=15, p_mutate=25, max_mutate=6, pop_size=15, optimize_power=1. Diese Werte führten ebenfalls zu einem guten Ergebnis.

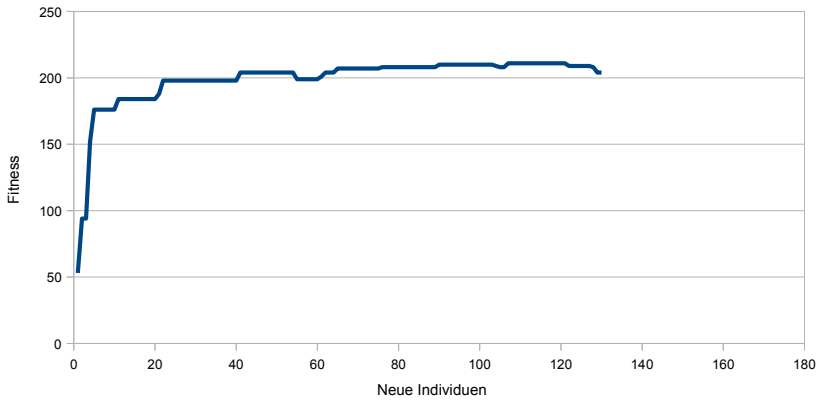


Abbildung 8: Lernkurve mit höherer Mutation und weniger Crossover

Abbildung 9 zeigt die Lernkurve mit den Parametern p_crossover=15, p_mutate=25, max_mutate=6, pop_size=10, optimize_power=1. Anscheinend führt die Verringerung der Populationsgröße zu einem geringeren Anstieg der Fitness.

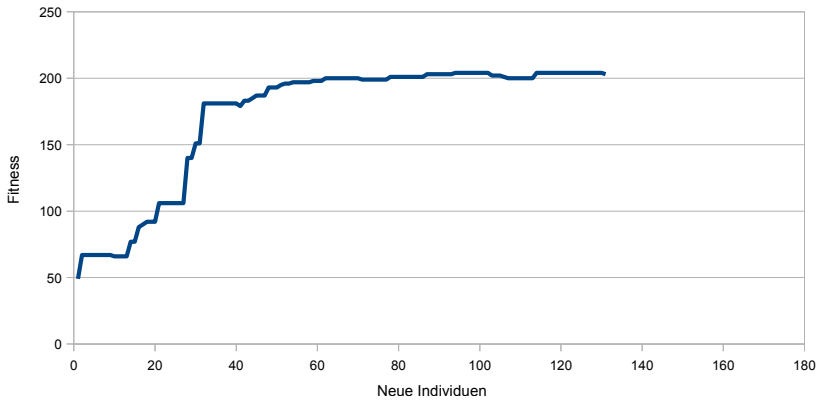


Abbildung 9: Lernkurve mit kleinerer Population

Abbildung 10 zeigt die Lernkurve mit den Parametern p_crossover=15, p_mutate=55, max_mutate=6, pop_size=15, optimize_power=1. Die hohe Mutationrate in diesem Test führt dazu, dass zwar schnell ein fittes Individuum gefunden werden kann, dieses jedoch auch schnell durch Kinder mit einer wesentlich schlechteren Fitness verdrängt werden kann. Eventuell würde eine Erhöhung der Populationsgröße diesen Nachteil ausgleichen.

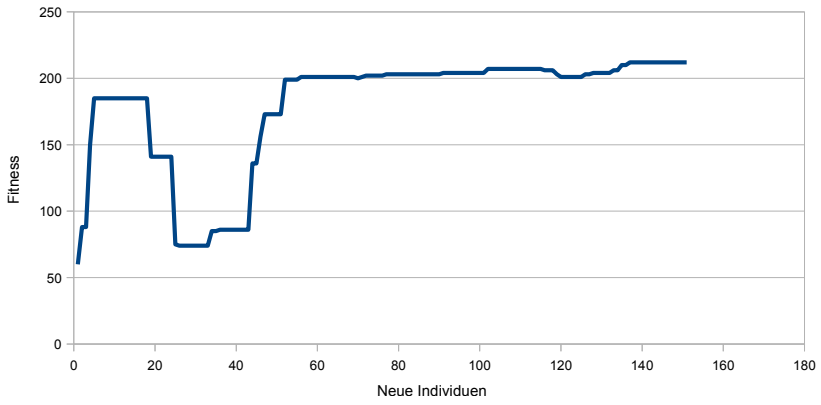


Abbildung 10: Lernkurve mit hoher Mutationsrate

Abbildung 11 zeigt die Lernkurve mit den Parametern p_crossover=15, p_mutate=25, max_mutate=10, pop_size=15, optimize_power=1. Wirkt sich eine Mutation zu stark aus, so tritt der selber Effekt der Verdrängung fitter Individuen auf. Diese Einstellung ist eher weniger empfehlenswert.

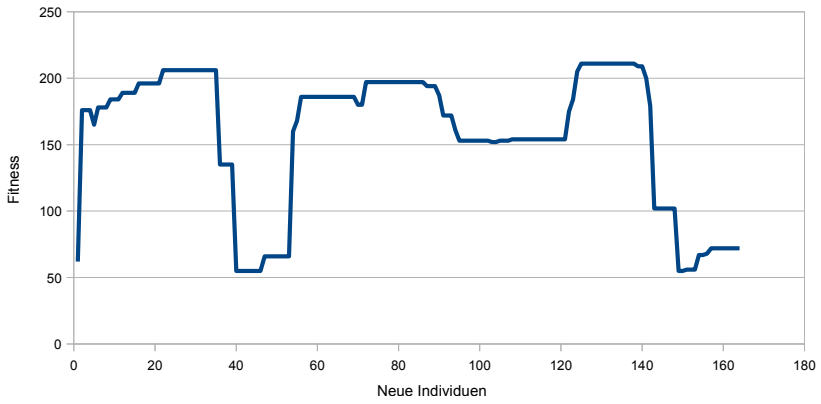


Abbildung 11: Lernkurve mit starker Mutation

Abbildung 12 zeigt die Lernkurve mit den Parametern p_crossover=15, p_mutate=25, max_mutate=6, pop_size=50, optimize_power=1. Bei einer großen Population ist die Wahrscheinlichkeit, dass bereits ein gutes Individuum zufällig erzeugt wurde, höher. Es kann aber passieren, dass die Fitness dann nur noch sehr langsam ansteigt, da die Selektionswahrscheinlichkeit des einzelnen Individuums sinkt. Um dies zu kompensieren, sollte die Turniergröße, also der Parameter best_of_x, immer relativ zu der Populationsgröße gewählt werden, um den Selektionsdruck zu erhöhen.

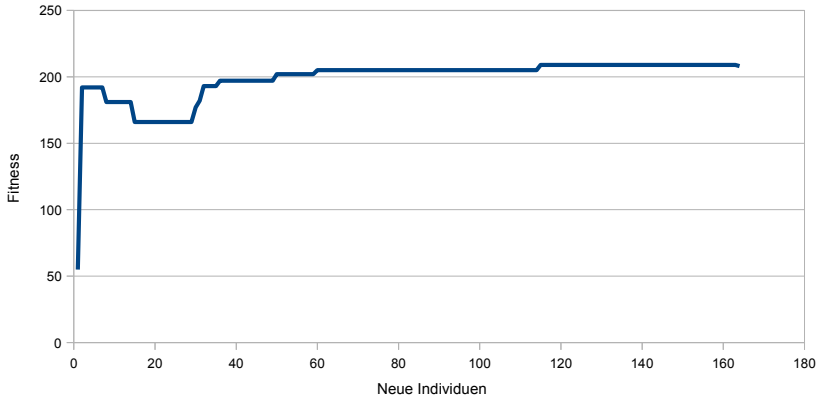


Abbildung 12: Lernkurve mit großer Population