Universität Kassel
Fachgebiet Verteilte Systeme
Prof. Dr. Kurt Geihs

Projekt: Distibuted Genetic Programming Framework (DGPF)
Projektleiter: Dipl. Inf. Thomas Weise

Lösen mathematischer Funktionen und Sudokus mit Hilfe des DGPF

von Stefan Niemczyk

Inhaltsverzeichnis

1. Einleitung

Ziel dieses Projektes war es, Tests zu schaffen an Hand derer das DGPF [1,2,3] (Distributed Genetic Programming Framework) getestet und mit anderen ähnlichen Projekten verglichen werden kann. Für diese Tests wurden mehrdimensionale mathematische Funktionen benutzt, die von allen drei bis jetzt implementierten Suchalgorithmen bewältigt werden mussten. Hierauf wird im dritten Abschnitt genauer eingegangen.

Ein weiterer Test ist der Sudoku-Solver, der jedoch nicht zum Vergleich dient, sondern eine andere, ebenfalls wichtige Aufgabe hat. Er soll die Einarbeitung in das Framework erleichtern. Eine genaue Beschreibung des Sudoku-Solvers befindet sich im zweitem Abschnitt.

Grundlegende Kenntnisse über Genetische Algorithmen sind nötig um diese Ausarbeitung verstehen zu können. Des Weiteren sind Kenntnisse über das DGPF hilfreich. Für ein leichteres und schnelleres Verständnis, und um nicht auch noch Java Kenntnisse voraus zu setzten, wurde in dieser Ausarbeitung Pseudocode anstatt des Original Java Codes verwendet.

2. Sudoku-Solver

Hauptziel des Sudoku-Solvers ist es, späteren Entwicklern und Benutzern die Einarbeitung in den Framework zu erleichtern. Hierbei bietet die grafische Oberfläche, mit Hilfe der Einstellungsmöglichkeiten für die unterschiedlichen Suchverfahren, bereits einen kleinen Überblick über die Suchalgorithmen. Anhand der Statusausgaben kann beobachtend werden, wie sich ein Individuum im Laufe der Zeit entwickelt. Das eine solche Entwicklung auch in eine falsche Richtung gehen kann und somit erneut von vorne begonnen werden muss ist ebenfalls beim Beobachten der Statusinformationen zu sehen.

Mit Hilfe des Sudoku-Solvers ist es möglich ein vordefiniertes oder ein eigenes Sudoku zu lösen. Hierbei ist jedoch zu berücksichtigen, dass bei der Eingabe kein Fehler gemacht werden darf, da ein nicht lösbares Sudoku nicht erkannt wird und der Solver so niemals terminieren wird. Bei einer korrekten Eingabe oder beim Verwenden eines vordefinierten Feldes wird der Solver irgendwann die Lösung finden, wobei hier die Betonung auf irgendwann liegt. Der Grund, dass es teilweise ziemlich lange dauern kann, ist, dass Sudoku unter anderem ein recht simples Rätsel ist. Das Framework ist jedoch dazu gedacht sehr komplexe Dinge zu lösen. Ein deterministisches Suchverfahren, bzw. Brute-Force wäre hier auf jeden Fall wesentlich schneller. Hinzu kommt noch, dass die verwendeten Genetischen Algorithmen noch nicht so Optimal an die Problemstellung angepasst sind. Die maximale Geschwindigkeit ist also noch nicht erreicht. Sudokus in neuer Rekordzeit zu lösen war hier jedoch auch nicht das Ziel, hierbei ging es eher um die Machbarkeit.

2.1 Was ist Sudoku

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Abbildung 1, Sudoku-Feld Quelle [4]

Sudoku [4] ist ein recht einfach zu verstehendes, wenn vielleicht auch nicht einfach zu lösendes, Rätsel. Es handelt sich dabei um ein Rätsel bei dem ein Feld von 9x9 Elementen ausgefüllt werden muss, insgesamt also 81 Elemente. Dieses Feld besteht aus 9 3x3 Feldern. Es wird mit Zahlen von 1 bis 9 gefüllt. Angemerkt werden sollte hier, dass es nicht unbedingt Zahlen seien müssen, sondern auch abstrakte Symbole seien könnten. Der Übersicht halber werden jedoch in der Regel Zahlen benutzt. In jeder Spalte, Zeile und in jedem 3x3 Feld darf eine Ziffer jeweils nur einmal vorkommen. Meistens sind 22 bis 36 Felder vorgegeben, wobei die Anzahl der vorgegebenen Felder nicht proportional zum Schwierigkeitsgrad ist. Ziel ist es nun die leeren Felder zu füllen, wobei die eben genannten Regeln eingehalten werden müssen.

2.2 Die Grafische Oberfläche

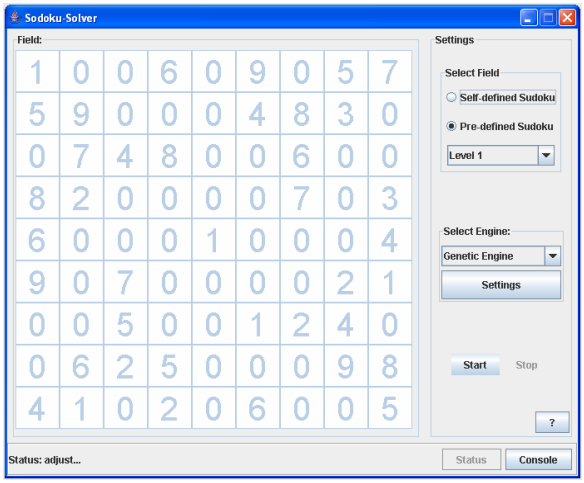

Abbildung 2, Hauptfenster der Oberfläche

Das Hauptfenster ist nicht für das Verstehen des Frameworks relevant. Daher wird hier nicht genauer darauf eingegangen. Es wird die Möglichkeit gegeben ein Feld zu wählen, ob nun ein eigenes oder ein vordefiniertes, den anzuwendenden Suchalgorithmus zu wählen und entsprechend einzustellen, den Solver zu starten und zu stoppen und sich den momentan aktuellen Status anzuschauen. Mit Hilfe der Konsole kann man sich eine Textausgabe ansehen. Diese muss jedoch bei den Einstellungen eingeschaltet werden.

Wesentlich interessanter hingegen sind die bereits erwähnten Einstellungsfenster. Diese Fenster geben die Möglichkeiten die wichtigsten Einstellungen des jeweiligen Suchalgorithmus zu ändern. In jedem Einstellungsfenster kann auch eine von Menschen lesbare Ausgabe aktiviert werden. Diese Textausgabe kann dann in der Konsole angesehen werden.

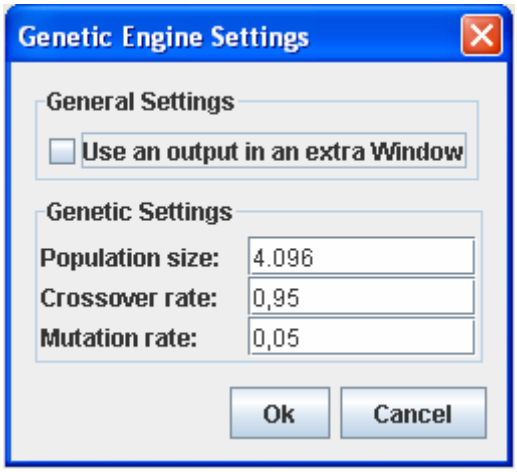

Abbildung 3, Einstellungsfenster für die Genetischen Algorithmen

Das Einstellungsfenster der genetischen Engine besteht aus den zusätzlichen drei veränderbaren Parametern Populationsgröße, Crossover-Rate und Mutations-Rate. Die Populationsgröße gibt an, wie viele Individuen in einer Generation erzeugt und überprüft werden. Die Crossover- und Mutations-Rate geben an, wie viel Prozent der Population durch Mutation bzw. Crossover erzeugt werden. Wie eine Mutation bzw. ein Crossover genau bei dem Sudoku-Solver abläuft wird in Abschnitt 2.3 beschrieben.

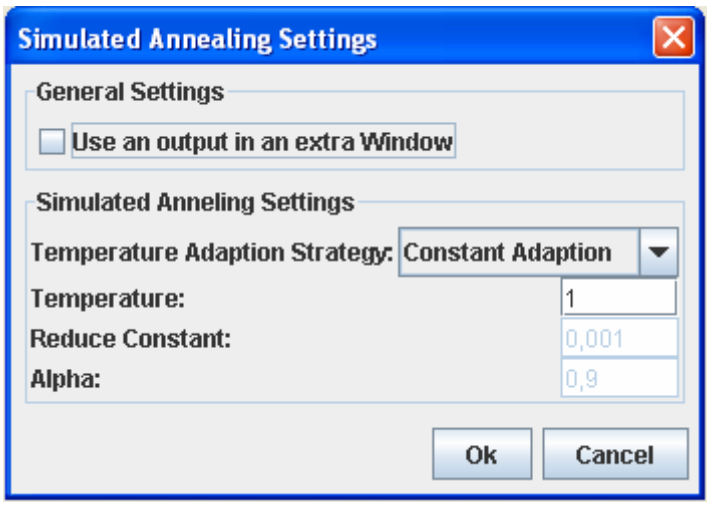

Abbildung 4, Einstellungsfenster für Simulated Annealing

Simulated Annealing bietet vier Einstellungsmöglichkeiten. Das Adaptionsverfahren für die Temperatur ist wählbar. Es stehen die drei Verfahren zur Verfügung: Constant Adaption, Arithmetic Adaption und Geometric Adaption. Die Temperatur ist die Starttemperatur mit der gestartet wird. Diese wird dann entsprechend adaptiert. Die Reduce Constant gehört zur arithmetischen Adaption und Alpha zur geometrischen Adaption. Wie diese Adaptionsverfahren genau funktionieren, kann der Ausarbeitung von Marc Kirchhoff [5] entnommen werden.

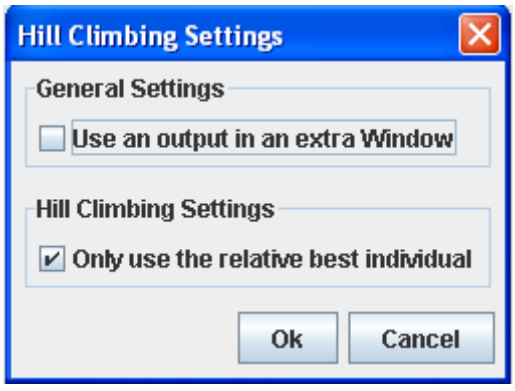

Abbildung 5, Einstellungsfenster für Hill Climbing

Hill Climbing ist ein relativ einfaches Suchverfahren. Daher gibt es hier auch nur eine Einstellungsmöglichkeit, die auch nicht zum reinen Hill Climbing Algorithmus gehört, sondern eine zusätzliche kleine Erweiterung des Projektleiters Thomas Weise darstellt.

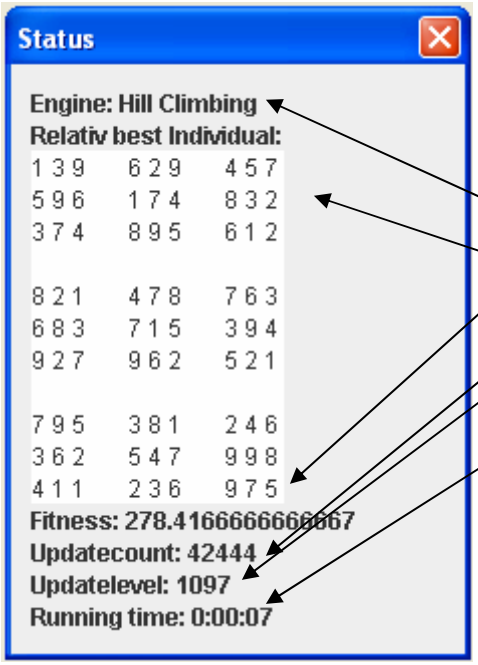

Abbildung 6, Statusfenster

Das Statusfenster zeigt die Entwicklung zur Laufzeit. Es werden die wichtigsten Informationen dargestellt. Für eine detaillierte Ausgabe sollte die Konsolenausgabe verwendet werden.
Es wird:

- der momentan verwendete Suchalgorithmus,
- das aktuell beste Individuum mit entsprechendem Fitnesswert,
- die Anzahl aller Updates,
- die Anzahl der Updates seit dem letzten Reset
- und die verstrichene Zeit

angezeigt.

Man kann sehr schön die Entwicklung beobachten. Wenn zum Beispiel die Entwicklung nicht weiter voran geht, wird die Engine resettet und das Updatelevel zurückgesetzt.

2.3 Funktionsweise

2.3.1 Grundlegender Aufbau

Der grundlegende Aufbau ist relativ einfach. Es wird ein vorgegebenes zweidimensionales Array mit Integer Werten angelegt. In diesem Feld sind die bereits vorgegebenen Zahlen eingetragen. Die noch leeren Stellen werden mit ganzzahligen Zufallszahlen zwischen einschließlich 1 bis 9 gefüllt. Das Feld wird immer wieder modifiziert bis die maximale Fitness, welche bei der verwendeten Fitness-Funktion 324 beträgt, erreicht ist oder ein Reset stattfindet. Ist die maximale Fitness erreicht, ist das Sudoku gelöst.

2.3.2 Erzeugen neuer Individuen

Eine statische Klasse stellt drei Methoden zur Verfügung mit denen ein Sudoku-Feld und eine Liste mit allen nicht gesetzten Feldern innerhalb dieses Feldes erzeugt werden kann. Das Feld wird gesetzt sobald die Engine gestartet wird. Aus diesem Feld wird dann die Liste der leeren Felder generiert.

Es gibt drei Funktionen die neue Individuen erzeugen können, `do_create()`, `do_mutate(altes_individuum)` und `do_crossover(altes_individuum_1, altes_individuum_2)`. `do_create()` erzeugt komplett neue Individuen, während die beiden anderen aus einem existierenden Individuum durch Mutation, bzw. Crossover aus zwei existierenden Individuen ein neues erschaffen. Die genaue Funktionsweise der drei Methoden wird mit Hilfe von Pseudocode erläutert. Wobei der Code nicht 1:1 dargestellt wird. Vielmehr wird darauf Wert gelegt, dass man die Funktionalität versteht.

2.3.2.1 do_create()

```
do_create()
{
     //Es wird das Feld und die Liste der leeren Elemente aus
     //der statischen Klasse geholt
     Feld = Sudoku.holeFeld();
     Liste = Sudoku.holeListeMitLeerenFeldern();

     //Jedes leere Feld wird gefüllt
     for(jedes Listenelement)
     {
          Feld[Element aus der Liste]= neueZahlZwischen1und9();
     }

     //Das neue Feld wird zurückgegeben
     return Feld;
}
```

2.3.2.2 do_mutate(altes_individuum)

```
do_mutate(altes_individuum)
{
     //Es wird eine Kopie des alten Feldes angelegt und es wird
     //eine Liste der leeren Elemente geholt
     Feld = altes_Feld.clone();
     Liste = Sudoku.holeListeMitLeerenFeldern();

     //Es werden eins bis zehn Felder geändert
     for(zufälligeZahlZwischen1und10())
     {
          //Bei einem zufälligen Feld wird die Zahl um eins
          //erhöht bei einer zehn (9+1) wird daraus eine eins
          Feld[Liste.zufälligesFeld()].erhöheUm1();
     }

     //Das neue Feld wird zurückgegeben
     return Feld;
}
```

2.3.2.3 do_crossover(altes_individuum_1, altes_individuum_2)

```
do_crossover(altes_individuum_1, altes_individuum_2)
{
     //Es wird ein neues Feld angelegt
     Feld;

     //Das neue Feld wird mit Hilfe der zwei alten gefüllt. Zu
     //50% wird eine Zahl aus dem ersten Feld, ansonsten aus
     //dem zweiten Feld verwendet
```

```
     for(jedes Feldelement)
     {
          if(zufallsBooleanWert())
          Feld[Feldelement]=altes_individuum_1[Feldelement];
          else
          Feld[Feldelement]=altes_individuum_2[Feldelement];
     }

     //Das neue Feld wird zurückgegeben
     return Feld;
}
```

2.3.3 Die Fitness-Funktion

Die Fitness-Funktion bestimmt wie gut oder schlecht ein erzeugtes Individuum ist. Anhand dieses Wertes wird bestimmt, welche Individuen weiterverwendet werden und welche verworfen werden. Hierbei handelt es sich um einen wichtiges Element bei der Verwendung des DGPFs und von Genetischen Algorithmen allgemein.
Die Fitness-Funktion für Sudoku zählt das Vorkommen der Zahlen von 1 bis 9 für jede Spalte, jede Zeile und jeden 3x3 Block. Des Weiteren wird bei jedem einzelnen Feld gezählt, wie viele Duplikate der Zahl es in der entsprechenden Zeile und Spalte gibt.

Anhand dieser Werte wird dann die Fitness für das Individuum auf folgende Art bestimmt:

```
1/Vorkommen der Zahl
```

Wenn die Zahl gar nicht vorkommt, wird dies wie ein doppeltes Vorkommen behandelt, um eine Division durch Null zu verhindern. Bei dem Zählen des Vorkommens für ein Feld in Zeile und Spalte ist der Idealfall eine Zwei, da die Zahl dann genau einmal pro Spalte und einmal pro Zeile vorkommt. Die maximale Fitness beträgt 324, sobald dieser Wert erreicht ist, terminiert die Engine und gibt das entsprechende Individuum als Lösung aus.

3. Mathematische Funktionen und Vergleichstests

Es wurden vier mathematische Funktionen gewählt und leicht modifiziert implementiert um das Distributed Genetic Programming Framework zu bewerten. Als Vergleichsumgebung nutzen wir das von Marion Riedel an der TU Chemnitz im Rahmen einer Diplomarbeit in der Programmiersprache C entwickelt Framework [6]. Vorweg sei gleich gesagt, dass ein in C geschriebener Framework allein wegen der in Java vorhandenen Virtuel Maschine schneller ist und daher die Ergebnisse, was die Zeit angeht, nicht eins zu eins verglichen werden können. Ein um den Faktor zehn schlechteres Ergebnis ist nicht unbedingt ein schlechtes Ergebnis.

3.1 Multiobjektivität

Ein wichtiger Faktor bei diesen Tests war es, sie multiobjektiv zu implementieren, d.h. sie mit mehr als einer Fitness-Funktion umzusetzen. Dies ist möglich da alle Funktionen ein Summenzeichen besitzen. Das Summenzeichen wurde aufgespalten und jedes Element als eine einzelne Fitness-Funktion implementiert. Diese mit

Gewalt geschaffene Multiobjektivität hat einerseits den Vorteil, dass das Framework bezüglich der Komparatoren, welche bei mehreren Fitness-Funktionen entscheiden welches Individuum weiterverwendet wird, getestet werden kann, jedoch auch den Nachteil, dass es dadurch noch mal etwas langsamer wird.

3.2 Funktionsweise und Aufbau

3.2.1 Grundlegender Aufbau

Im Gegensatz zum Sudoku gibt es hier kein Feld sondern einen Vektor mit reellen Zahlen. Jede einzelne Stelle des Vektors wird durch eine separate Fitness-Funktion ausgewertet. Von Generation zu Generation werden die Zahlen modifiziert bis das Maximum innerhalb des Definitionsbereiches, von -100.999 bis 100.999 der reellen Zahlen, gefunden ist. Das Maximum wurde im Vorfeld mit Hilfe von mathematischen Programmen bestimmt und ist daher bekannt. Sobald das Maximum erreicht ist, terminiert die Engine.

3.2.2 Umformen der Funktionen

Wie bereits erwähnt wurden alle verwendeten Funktionen leicht modifiziert. Das vorhandene Summenzeichen wurde zu Gunsten der Multiobjektivität aufgelöst. Des Weiteren wurden alle Funktionen so umgeformt, dass sie nun innerhalb des Definitionsbereiches (von -100.999 bis 100.999 der reellen Zahlen) immer positiv sind. Negative Werte sind als Fitnesswerte im DGPF nicht möglich und werden vom Framework als 0 angenommen. Es wird auch im Gegensatz zu dem Framework von Marion Riedel nicht mehr nach dem Minimum sondern nach dem Maximum gesucht.

3.2.3 Erzeugen neuer Individuen

Auch hier gibt es dieselben drei Funktionen, wie bei Sudoku, die neue Individuen erzeugen können, nämlich `do_create()`, `do_mutate(altes_individuum)` und `do_crossover(altes_individuum_1, altes_individuum_2)`. Diese Funktionen sind nötig um mit dem Framework zu arbeiten und müssen für jedes Problem neu implementiert werden. Die Implementierung der drei Methoden wird ebenfalls in Pseudocode gezeigt. Wobei der Code nicht 1:1 dargestellt wird. Vielmehr wird darauf Wert gelegt, dass man die Funktionalität versteht.

3.2.3.1 do_create()

```
do_create()
{
     //Erzeugen eines neuen Double-Arrays
     double_array[] = new double[];

     //Füllen des Array mit Zufallszahlen aus dem
     //Definitionsbereich
     for(jedes Element des Arrays)
     {
          double_array[index] = zufallszahlAusDefbereich();
     }
```

```
     //Liefert das Array zurück
     return double_array ;
}
```

3.2.3.2 do_mutate(altes_individuum)

```
do_mutate(altes_individuum)
{
     //Legt eine Kopie des alten Individuums an
     neues_array[] = altes_inidiuum.clone();

     //Die Zahl die modifiziert wird.
     modZahl = neues_array[zufälligesElement];

     //Es gibt mehrere Möglichkeiten wie die Zahl modifiziert
     //werden kann. Dieser Vorgang wird so lange wiederholt,
     //bis die modifizierte Zahl innerhalb des Definitions-
     //bereiches liegt.
     do
     {
          //Mit einer sehr geringe Wahrscheinlichkeit (1:30)
          //wird eine neue Zahl erzeugt.
          if(zufallsZahlZwischen0Und30 = 0)
          {
               modZahl = zufallszahlAusDefbereich();
          }
          //Wenn die alte Zahl den Wert null hat, wird sie
          //durch eine gaussverteilte Zufallszahl ersetzt
          else if(modZahl == 0.0)
          {
               modZahl = gaussverteilteZufallszahl();
          }
          //Ansonsten wird die Zahl entweder um einen großen
          //oder einen kleinen Wert geändert.
          else
          {
          //Das Verhältnis von größer zu kleiner Änderung ist
          //3:2. Die neue Zahl ist eine Normalverteilte
          //Zufallszahl um die alte Zahl.
               if(zufallszahlZwischen0Und5 <= 2)
               {
               //Der stddef ist hier von der Zahl selber
               //abhängig, dies lässt große Sprünge zu.
               modZahl = normalverteileZufallszahl(modZahl,
                         variabler_stddef);
               }
               else
               {
               //Der stddef ist hier fest 1,25. Dies lässt nur
               //kleinere Sprünge zu.
```

```
                modZahl = normalverteileZufallszahl(modZahl,
                          fester_stddef);
                }
          }
     } while(Zahl außerhalb def. Bereich)

     //Die Zahl wird wieder in das Array zurückgeschrieben
     //Bei zufälligesElement handelt es sich um dasselbe
     //Element wie beim entnehmen der Zahl
     neues_array[zufälligesElement] = modZahl;

     //Liefert das neue modifizierte Array zurück
     return neues_array;
}
```

3.2.3.3 do_crossover(altes_individuum_1, altes_individuum_2)

```
do_crossover(altes_individuum_1, altes_individuum_2)
{
     //Legt ein neues Array an
     neues_array[;

     //Das neue Array wird mit Hilfe der zwei alten gefüllt.
     //Dabei gibt es drei unterschiedliche Operationen. Das
     //Verhältnis der Operationen ist 2:2:1
     for(jedes Element des Arrays)
     {
          if(zufallszahlZwischen0und100 >= 40)
          {
               //Es wird die Zahl des ersten alten Individuums
               //genommen.
               neues_array[Feldelement] =
                    altes_individuum_1[Feldelement];
          }
          else if(zufallszahlZwischen0und100 >= 80)
          {
               //Es wird die Zahl des zweiten alten Individuums
               //genommen.
               neues_array[Feldelement] =
                    altes_individuum_2[Feldelement];

          }
          else
          {
               //Ansonsten wird der Mittelwert zwischen den
               //beiden Zahlen gebildet.
               neues_array[Feldelement]=
                    (altes_individuum_1[Feldelement] +
               altes_individuum_2[Feldelement])/2;
          }
     }
```

```
     //Liefert das neue modifizierte Array zurück
     return neues_array;
}
```

3.2.4 Die Fitness-Funktion

Auf Grund der Einfachheit der Fitness-Funktion lässt sie sich auch am leichtesten mit Pseudocode erläutern.

```
fitness_funktion(array_mit_double_werten[])
{
     //Bestimmen der Fitness mit Hilfe der Funktion die aktuell
     //berechnet wird ( also f(x) wobei x der entsprechende
     //Wert aus dem übergebenem Array ist )
     Fitness=aktuelleFunktion(zugeordneter Wert aus dem Array);

     //Liefert den Fitnesswert zurück
     return Fitness;
}
```

3.3 Funktionen

3.3.1 Sphärenfunktion von Rechenberg

Die Sphärenfunktion von Rechenberg ist die einfachste der vier Testfunktionen. Aufgrund ihres recht kontinuierlichen Aufbaus sollte sie schnell zu lösen sein.

Die ursprüngliche und von Riedel verwendete Funktion sieht folgendermaßen aus:

$$f\left(\vec{x}\right) = \sum_{i=1}^{n} x_i^2$$

Das Summenzeichen wurde von mir aufgelöst, die Funktion invertiert und um 10200.798 ins Positive verschoben. Durch diese Änderungen liegt die Methode innerhalb des Definitionsbereiches (von -100.999 bis 100.999 der reellen Zahlen) vollständig im positiven Bereich und besitzt dort ebenfalls einen Hochpunkt. Die umgeformte und verwendete Funktion hat nun folgende Gestallt:

$$f_1(x_i) = -x_i^2 + 10200.798$$

Das Maximum von f_1 für jede einzelne Funktion, nicht für die Summe aller Funktionen, befindet sich bei 10200.798 und wird erreicht mit x = 0.

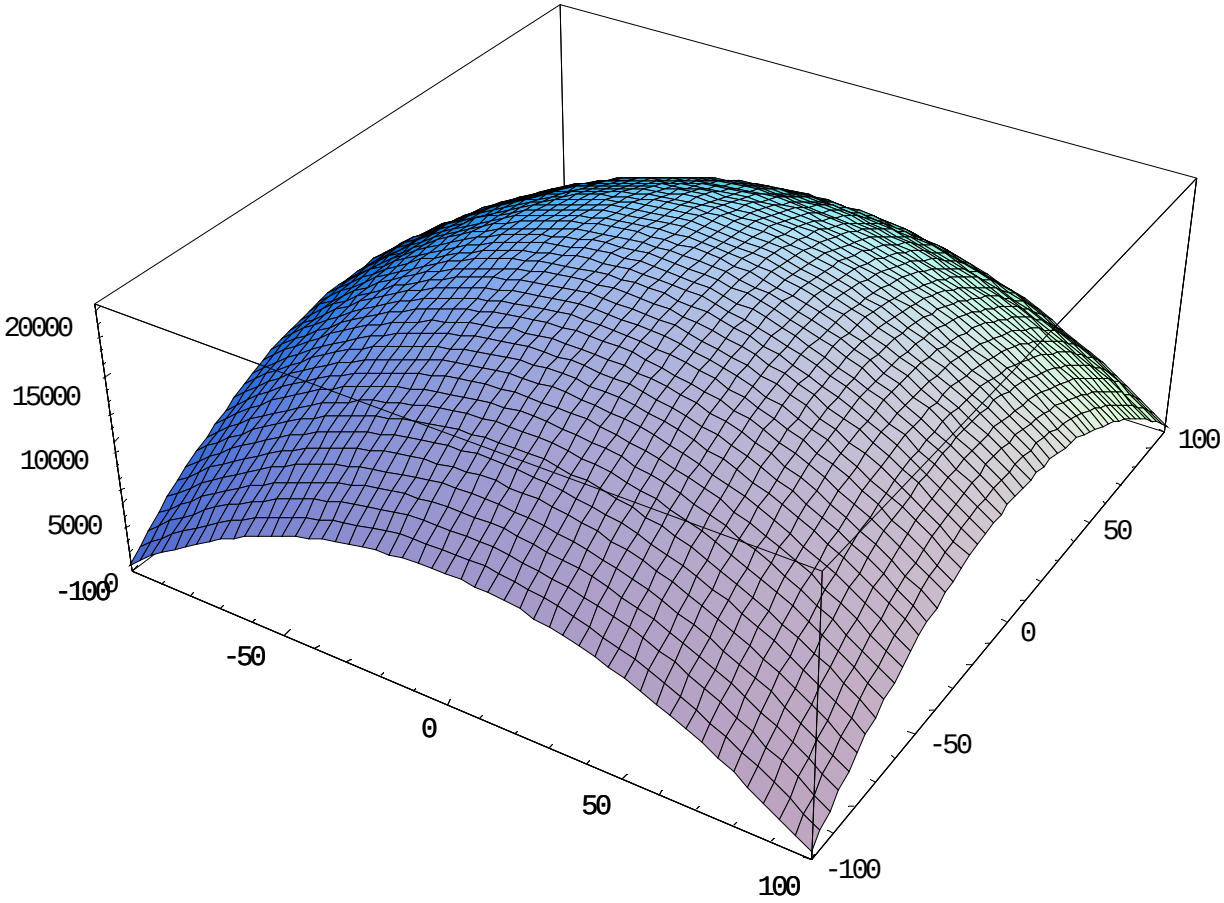

Abbildung 7, Abgeänderte Sphärenfunktion von Rechenberg im $\mathbb{R}^3$

3.3.2 Treppenfunktion

Die Treppenfunktion bereitet vielen Optimierungsverfahren Probleme, da sie über viele Flächen verfügt, wo jeder Punkt den gleichen f(x) Wert und somit auch den gleichen Fitnesswert hat. Dies ist für den DGPF nicht allzu problematisch. Der Grund ist, dass die Genetischen Algorithmen so viele Punkte pro Generation erzeugen und überprüfen, dass es selten nicht weiter voran geht. Simulated Annealing kann auch für einige Schritte mit einem nicht besseren Wert weiterrechen, womit das lokale Optimum wieder verlassen werden kann. Hill Climbing verfügt hier, dank der von Thomas Weise hinzugefügten Option, ebenfalls über die Möglichkeit mit einem nicht besseren Wert weiterzurechen. Aber auch wenn diese Option ausgeschaltet ist, ist das Suchverfahren noch in der Lage das Maximum zu finden. Diese Funktion lässt sich auf Grund des permanenten Abrundens sogar sehr schnell lösen, da der Wertebereich doch erheblich kleiner wird. Er beschränkt sich hier sozusagen nur auf die natürlichen Zahlen zwischen -100.999 und 100.999. Intern wird jedoch weiter mit Fließkommazahlen gerechnet.

Die ursprüngliche und von Riedel verwendete Funktion sieht folgendermaßen aus:

$$f_2(x_i) = \lfloor x_i \rfloor + 100$$

Das Summenzeichen wurde von mir aufgelöst und die Funktion wurde um 100 ins Positive verschoben und genügt somit den Anforderungen. Die umgeformte und verwendete Funktion hat nun folgende Gestalt:

$$f(x) = \sum_{i=1}^{n} \lfloor x_i \rfloor$$

Das Maximum von f_2 für jede einzelne Funktion, nicht für die Summe aller Funktionen, befindet sich bei 200 und wird erreicht mit x = 100.

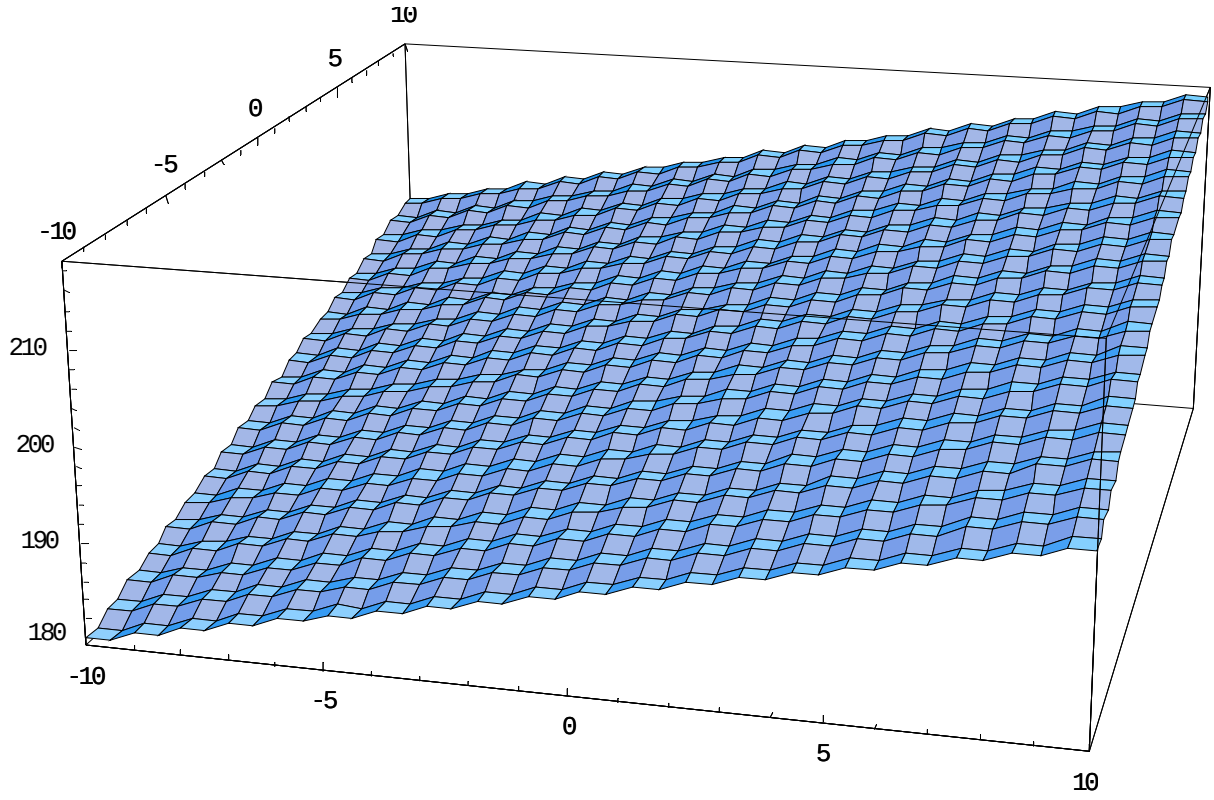

Abbildung 8, Abgeänderte Treppenfunktion im $\mathbb{R}^3$

3.3.3 Rastrigin Funktion

Die Rastrigin Funktion ist zwar nur ein Ableger der Sphärenfunktion von Rechenberg, sie wird jedoch erheblich komplizierter durch den zusätzlichen Kosinus-Term. Es entstehen viele kleine Hügel und Täler die jedoch nur lokale Maxima/Minima sind. Dies macht das Suchen nach dem globalen Maximum/Minimum relativ schwierig.

Die ursprüngliche und von Riedel verwendete Funktion sieht folgendermaßen aus:

$$f(\vec{x}) = 10n + \sum_{i=1}^{n} \left(x_i^2 - 10\cos(2\pi x_i) \right)$$

Der Term +10n wurde entfernt und das Summenzeichen aufgelöst. Anschließend wurde die Funktion noch invertiert und um 10200.798 ins Positive verschoben,

genauso wie die Sphärenfunktion von Rechenberg. Die umgeformte und verwendete Funktion hat nun folgende Gestallt:

$$f_4(x_i) = -\left(x_i^2 - 10\cos(2\pi x_i)\right) + 10200.798$$

Das Maximum von f_4 für jede einzelne Funktion, nicht für die Summe aller Funktionen, befindet sich bei 10210.798 und wird erreicht mit x = 0.

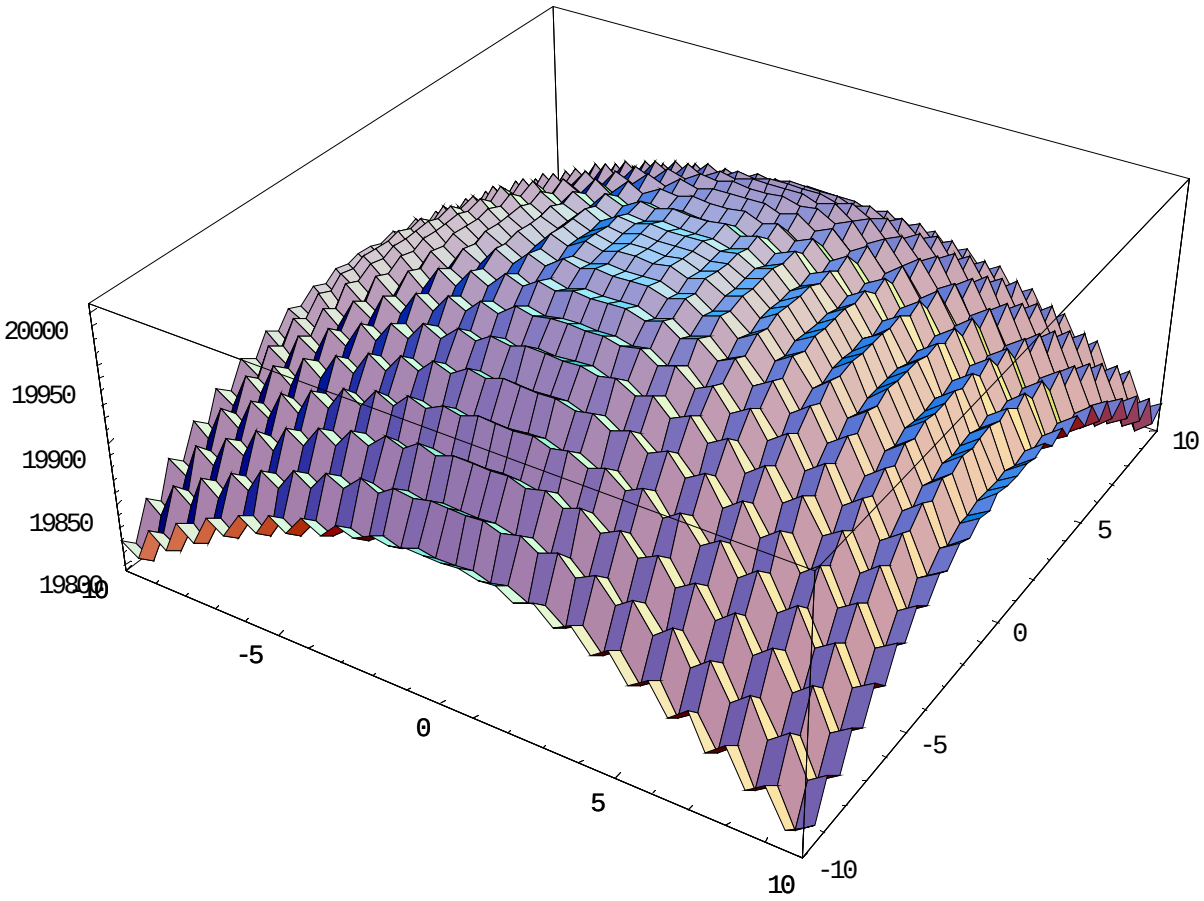

Abbildung 9, Abgeänderte Rastrigin Funktion im ℝ3

3.3.4 Boshafte Schwefel Funktion

Der Namesteil „Boshafte“ kommt bei dieser Funktion nicht von ungefähr. Sie führt auf der Suche nach einem globalen Minimum/Maximum immer wieder in lokale Minima/Maxima, die auch noch sehr ähnliche Werte haben.

Die ursprüngliche und von Riedel verwendete Funktion sieht folgendermaßen aus:

$$f(\vec{x}) = -\sum_{i=1}^{n} x_i \sin\left(\sqrt{|x_i|}\right)$$

Die Umformung dieser Funktion war ebenfalls relativ einfach. Sie wurde erneut invertiert und das Summenzeichen wurde aufgelöst. Anschließend wurde sie noch

um 100 ins Positive verschoben. Die umgeformte und verwendete Funktion hat nun folgende Gestalt:

$$f_5(x_i) = x_i \sin\left(\sqrt{|x_i|}\right) + 100$$

Das Maximum von f_5 für jede einzelne Funktion, nicht für die Summe aller Funktionen, befindet sich bei 163.6349819 und wird erreicht mit x = 65.54786507744413.

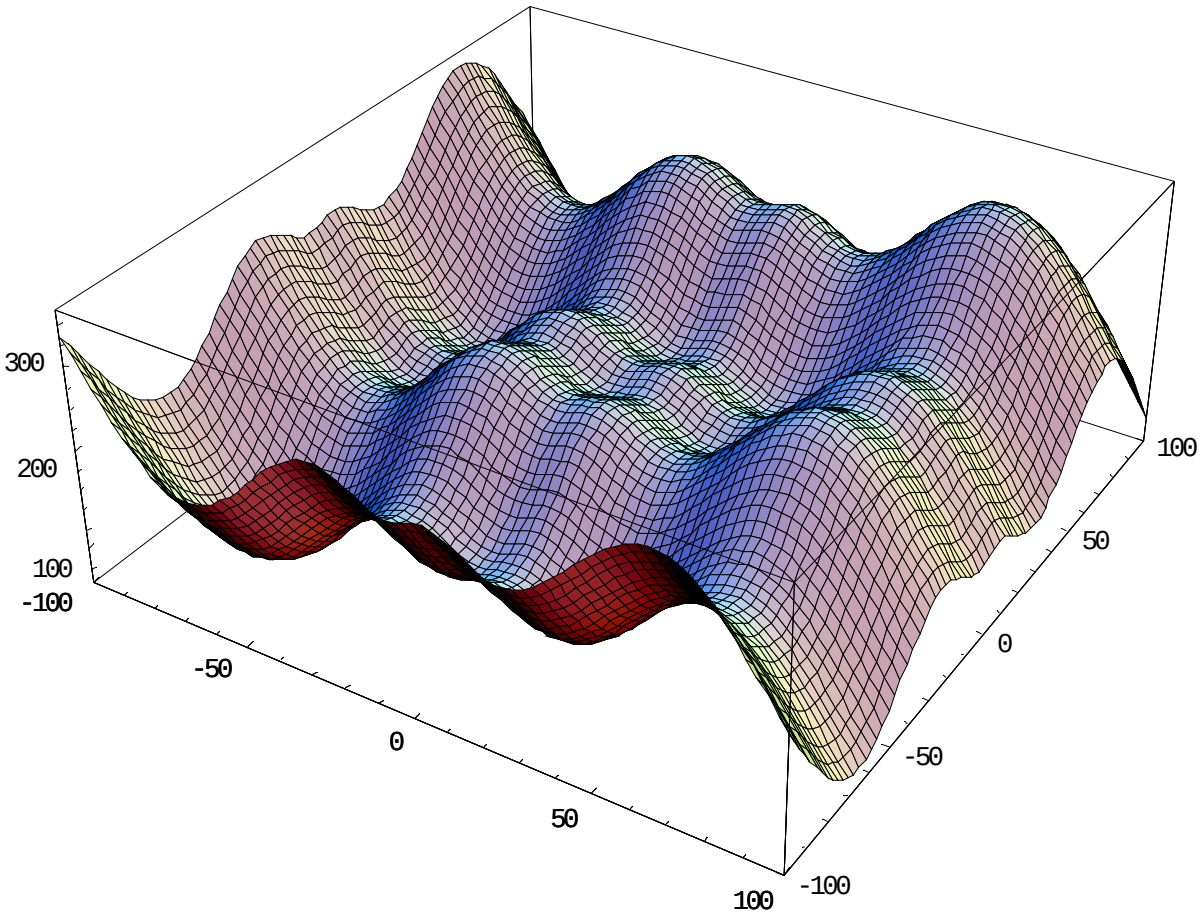

Abbildung 10, Abgeänderte Boshafte Schwefel Funktion im ℝ3

3.4 Aufbau der Tests

Jeder der drei Suchalgorithmen wurde mit jeder der vier Testfunktionen getestet. Jede Funktion wurde sowohl einmal mit einem zehndimensionalen Eingangsvektor, wie auch mit einem zweidimensionalen Eingangsvektor getestet. Des Weiteren wurde jeder Test sowohl mit dem Summen-Komparator, wie mit dem Majority-Komparator durchgeführt. Um ein verwendbares Ergebnis zu erhalten wurde jeder Test zehnmal wiederholt, um ein zufälliges, besonders gutes oder schlechtes Lösen heraus zu filtern.

Es wurde eine maximale Laufzeit von 300000 ms = 5 Minuten festgelegt. Wenn innerhalb dieser Zeit die Lösung nicht gefunden wurde, wird der Durchlauf abgebrochen und als fehlgeschlagen ausgegeben. Da jeder Test jedoch erfolgreich verlaufen ist, ist die Anzahl erfolgreicher Durchläufe nicht in den Tabellen enthalten.

Eine Lösung gilt als gefunden, wenn sie entweder bis zur 6ten Nachkommastelle genau gefunden wurde, oder falls dies nicht möglich ist, bis auf eine Abweichung von 0,000 000 1 genau gefunden wurde. Das heißt bei einer maximalen Fitness von 163.634 981 9 reicht auch 163.634 981, bzw. bei einer maximalen Fitness von 200 reicht auch 199,999 999 9.

Als Testrechner wurde ein Laptop mit einem 1,5 Ghz Intel Centrino Mobil Prozessor und 512 MB Arbeitsspeicher verwendet.

3.4.1 Die Komparatoren

Der Komparator entscheidet bei mehreren Fitness-Funktionen welches Individuum weiterverwendet wird und welches nicht. Hierfür gibt es viele unterschiedliche Strategien. Für die Tests wurden folgende zwei verwendet:

1. Majority-Komparator

Der Majority-Komparator entscheidet sich für das Individuum, was in den meisten Fitnessfunktionen über die Anderen dominiert.

2. Summen-Komparator

Der Summen-Komparator entscheidet sich für das Individuum, das mit der Summe seiner Fitnesswerte den höchsten Wert erreicht. In diesem Fall kann man ihn als eine Art Ersatz für das Summenzeichen in den Funktionen ansehen. Dadurch wird die Suche praktisch auf einen single-objektiven Algorithmus umgerechnet, wie er auch von Riedel verwendet wird.

3.4.2 Einstellungen der Suchverfahren

Für die drei Suchalgorithmen (Genetische Algorithmen, Simulated Annealing, Hill Climbing) wurden im Allgemeinen die Standard-Einstellungen verwendet.

Search Definition

Bei der Suchdefinition wurden folgende Einstellungen vorgenommen:

```
this.set_max_improvement_trials(5);
this.set_return_first_improvement(true);
this.set_challenge_parent(false);
```

1. Genetic Algorithme

Für diesen Algorithmus wurden die Einstellungen aus Riedels Arbeit [6] größtenteils übernommen.

- Mutations-Rate: 5%
- Crossover-Rate: 95%
- Populationsgröße: 200

2. Simulated Anneling

Hier wurde als Temperaturadaptions-Methode die geometrische Adaption verwendet. Ansonsten wurde alles bei den Standardwerten belassen.

3. Hill Climbing
Hier wurden keine Änderungen der Standard-Einstellungen vorgenommen.

3.5 Ergebnisse

Die Ergebnisse für die einzelnen Funktionen werden tabellarisch aufgeführt. Zum Vergleich sind ebenfalls die Ergebnisse für die entsprechenden Funktionen von Marion Riedel mit angeführt.

Zu beachten sei jedoch, dass Riedel nach dem globalen Minimum gesucht hat, daher können die Fitnesswerte nicht miteinander verglichen werden. Die Werte für die Laufzeit und für die Anzahl erzeugter Individuen jedoch schon. Des Weiteren hat Riedel keine unterschiedlichen Komparatoren getestet, die zwei unterschiedlichen Komparatoren wurden nur für einen internen Vergleich getestet.

Uns dient vor allem die mittlere Anzahl erzeugter Individuen als Vergleichswert. Ist das DGPF besser als das C Framework sind sie grün hinterlegt, ist das DGPF schlechter rot.

3.5.1 Sphärenfunktion von Rechenberg

Majority-Comparator

Verwendetes Suchverfahren	Eingangs-vektor	Globals Optimum	Mittelwert Fitness	Mittlere Laufzeit	Mittelwert erzeugter Individuen
Genetic Algorithm	R2	10000	9999.999999963879	0,517 s	15060
	R10	10000	9999.999999990812	17,614 s	192640
Simulated Annealing	R2	10000	9999.9999999463	0,203 s	1131
	R10	10000	9999.9999999845	0,906 s	6546
Hill Climbing	R2	10000	9999.999999945769	0,134 s	953
	R10	10000	9999.999999985928	0,988 s	7697
Ergebnis von Marion Riedel	R2	0,000 000	0,000 000 38	0,002 s	775,2
	R10	0,000 000	0,000 000 76	0,02 s	5212,4

Sum-Comparator

Verwendetes Suchverfahren	Eingangs-vektor	Globals Optimum	Mittelwert Fitness	Mittlere Laufzeit	Mittelwert erzeugter Individuen
Genetic Algorithm	R2	10000	9999.999999969952	0,277 s	7240
	R10	10000	9999.999999986538	6,879 s	89360
Simulated Annealing	R2	10000	9999.999999954554	0,192 s	1361
	R10	10000	9999.999999978667	0,929 s	6865
Hill Climbing	R2	10000	9999.999999946316	0,144 s	1203
	R10	10000	9999.99999998266	0,847 s	7201
Ergebnis von Marion Riedel	R2	0,000 000	0,000 000 38	0,002 s	775,2
	R10	0,000 000	0,000 000 76	0,02 s	5212,4

3.5.2 Treppenfunktion

Majority-Comparator

Verwendetes Suchverfahren	Eingangs-vektor	Globals Optimum	Mittelwert Fitness	Mittlere Laufzeit	Mittelwert erzeugter Individuen
Genetic Algorithm	R2	200	200	0,234 s	1680
	R10	200	200	1,266 s	10540
Simulated Annealing	R2	200	200	0,116 s	122
	R10	200	200	0,255 s	818
Hill Climbing	R2	200	200	0,119 s	98
	R10	200	200	0,240 s	818
Ergebnis von Marion Riedel	R2	-12,000 000	-12,000 000	0,015 s	6398,4
	R10	-60,000 000	-60,000 000	0,543 s	146117,9

Sum-Comparator

Verwendetes Suchverfahren	Eingangs-vektor	Globals Optimum	Mittelwert Fitness	Mittlere Laufzeit	Mittelwert erzeugter Individuen
Genetic Algorithm	R2	200	200	0,196 s	1260
	R10	200	200	1,247 s	10420
Simulated Annealing	R2	200	200	0,118 s	101
	R10	200	200	0,241 s	762
Hill Climbing	R2	200	200	0,115 s	86
	R10	200	200	0,288 s	709
Ergebnis von Marion Riedel	R2	-12,000 000	-12,000 000	0,015 s	6398,4
	R10	-60,000 000	-60,000 000	0,543 s	146117,9

3.5.3 Rastrigin Funktion

Majority-Comparator

Verwendetes Suchverfahren	Eingangs-vektor	Globals Optimum	Mittelwert Fitness	Mittlere Laufzeit	Mittelwert erzeugter Individuen
Genetic Algorithm	R2	10010	10009.999999963728	0,879 s	25940
	R10	10010	10009.999999988231	45,357 s	498880
Simulated Annealing	R2	10010	10009.999999951577	0,176 s	1870
	R10	10010	10009.999999980204	1,581 s	11757
Hill Climbing	R2	10010	10009.999999948415	0,268 s	2191
	R10	10010	10009.999999985697	1,427 s	11881
Ergebnis von Marion Riedel	R2	0,000 000	0,000 000 24	0,005 s	1389,1
	R10	0,000 000	0,000 000 53	0,092 s	18549,0

Sum-Comparator

Verwendetes Suchverfahren	Eingangs-vektor	Globals Optimum	Mittelwert Fitness	Mittlere Laufzeit	Mittelwert erzeugter Individuen
Genetic Algorithm	R2	10010	10009.99999995295	0,737 s	21680
	R10	10010	10009.999999976728	15,505 s	211720
Simulated Annealing	R2	10010	10009.999999945436	0,194 s	2216
	R10	10010	10009.999999984362	1,328 s	11585
Hill Climbing	R2	10010	10009.999999939322	0,155 s	1807
	R10	10010	10009.99999998353	1,342 s	12154

Ergebnis von Marion Riedel	R2	0,000 000	0,000 000 24	0,005 s	1389,1
	R10	0,000 000	0,000 000 53	0,092 s	18549,0

3.5.4 Boshafte Schwefelfunktion

Majority-Comparator

Verwendetes Suchverfahren	Eingangs-vektor	Globals Optimum	Mittelwert Fitness	Mittlere Laufzeit	Mittelwert erzeugter Individuen
Genetic Algorithm	R2	163.6349819	163.63498183713983	0,164 s	3520
	R10	163.6349819	163.6349819115747	3,807 s	38560
Simulated Annealing	R2	163.6349819	163.6349817270915	0,074 s	237
	R10	163.6349819	163.63498189439002	0,342 s	2273
Hill Climbing	R2	163.6349819	163.6349817800795	0,075 s	360
	R10	163.6349819	163.63498189915373	0,331 s	1798

Ergebnis von Marion Riedel	R2	-837,965 774	-837,965773	0,007 s	1489,4
	R10	-4189,828 87	-4189,828 87	224,746 s	38930419,8

Sum-Comparator

Verwendetes Suchverfahren	Eingangs-vektor	Globals Optimum	Mittelwert Fitness	Mittlere Laufzeit	Mittelwert erzeugter Individuen
Genetic Algorithm	R2	163.6349819	163.6349817703394	0,106 s	1800
	R10	163.6349819	163.63498186317122	1,083 s	13940
Simulated Annealing	R2	163.6349819	163.6349817567839	0,073 s	238
	R10	163.6349819	163.63498188326875	0,318 s	2213
Hill Climbing	R2	163.6349819	163.63498168185757	0,073 s	254
	R10	163.6349819	163.63498190877866	0,389 s	2465

Ergebnis von Marion Riedel	R2	-837,965 774	-837,965773	0,007 s	1489,4
	R10	-4189,828 87	-4189,828 87	224,746 s	38930419,8

3.6 Auswertung

Die Tests haben gezeigt, dass die Suchalgorithmen Simulated Annealing und Hill Climbing bessere Ergebnisse liefern als die Genetischen Algorithmen. Dies liegt daran, dass der verwendete Mutationsoperator besser ist als der Crossoveroperator. Die Einstellungen der genetischen Engine waren allerdings auch nicht Optimal. Mit einem 1:1 Verhältnis zwischen Mutation und Crossover und mit einer Populationsgröße von 250 erhält man bessere Ergebnisse. Es wurden nicht die optimaleren Einstellungen verwendet, da wir vergleichbare Ergebnisse erhalten wollten.

Der Summen-Komparator war jedes Mal schneller als der Majority-Komparator. Dies könnte daran liegen, dass er das Summenzeichen der Funktionen ersetzt. Ein anderer Grund könnte die Struktur der gestellten, sehr einfachen Aufgabe sein. Jede Fitness-Funktion wurde zum selben Wert hin optimiert, daher scheint es logischer das Individuum, welches in der Summe besser ist, weiter zu verwenden anstelle von dem, das die meisten Vergleiche gewinnt. Das Problem multiobjektiv zu betrachten, Verwendung des Majority-Komparators, scheint also keine Geschwindigkeitsvorteile zu bringen, sondern verlangsamt die Sache nur. Der Summen-Komparator ist offensichtlich besser für numerische Approximation geeignet, wo die Funktionen sehr ähnlich sind und keine großen Unterschiede in ihrem Wertebereich aufweisen.

Im direkten Vergleich der zwei Frameworks sieht man, dass wir in etwas über der Hälfte der Tests weniger Individuen brauchen. Vor Allem bei einem größeren Eingangsvektor liefert das DGPF bessere Ergebnisse als das C Framework. Zeitlich jedoch ist das DGPF jedes Mal, ob nun weniger oder mehr erzeugte Individuen, schlechter als das C Framework. Dies liegt zum einen an den verwendeten Programmiersprachen. Das Java langsamer ist als C ist jedoch nicht der einzige Grund. Das DGPF ist vor Allem für komplexe und lange Simulationen ausgelegt. Es bietet viele Werkzeuge an, wie interne Statistiken oder Multiobjektivität. Das vor Allem letzteres keine Vorteil, sonder sogar einen Nachteil ist, sieht man am Vergleich der zwei Komparatoren. Diese Werkzeuge erzeugen einen Overhead bei dem berechnen der Fitness-Funktionen. Dieser Overhead wird jedoch nicht größer bei komplexeren Aufgaben mit Fitness-Funktionen die wesentlich länger brauchen um ausgewertet zu werden. Man kann ihn also als Konstant annehmen, so dass er bei komplexen Aufgaben vernachlässigt werden kann. Bei dieser recht einfachen Aufgabe schlägt er jedoch erheblich zu Buche.

3.7 Wichtige Erkenntnis

Die Tests haben vor Allem gezeigt, dass die Umsetzung der genetischen Operatoren sehr große Auswirkungen haben. Wenn man die Crossover-Methode, die von den Genetischen Algorithmen fast ausschließlich zum Erzeugen neuer Individuen genutzt wird, anders implementiert erhält man erhebliche Geschwindigkeitsunterschiede. Um dies zu verdeutlichen vergleichen wir kurz die Ergebnisse von drei Implementierungen.

1. Implementierung

Hier wurde jeweils ein neues Feld mit den Elementen der zwei alten Felder erzeugt. Zu 50% stammte das Element vom ersten Feld, ansonsten vom zweiten Feld.

2. Implementierung

Hier wurde das Feld jeweils mit den Mittelwerten der entsprechenden Elemente der zwei Elternfelder gefüllt.

```
neues_element[i]=(altes_element_1[i] + altes_element_2[i])/2;
```

3. Implementierung

Hierbei handelt es sich um eine Mischung der zwei ersten Implementierungen. Entweder wird das Element des ersten Feldes, des zweiten Feldes oder ihr Mittelwert weiterverwendet. Das Verhältnis ist 2:2:1. Diese Implementierung kann auch als Pseudocode in Abschnitt 3.2.3.3 nachgelesen werden.

Wenn man jetzt nur die zeitlichen Unterschiede der durchschnittlichen Rechenzeit betrachtet, sieht man wie wichtig eine gute Implementierung ist. Wir betrachten die Ergebnisse der Rastrigen Funktion mit einem zweidimensionalen Eingangsvektor unter Verwendung des Majority-Komparators.

Implementierung	***AVG-Rechenzeit***
1. Implementierung	Weit über 1 Stunde
2. Implementierung	21.350 Sekunden
3. Implementierung	0.879 Sekunden

Man sieht sehr deutlich wie sich die unterschiedlichen Implementierungen auswirken. Dies betrifft natürlich nicht nur den Crossoveroperator, sondern alle drei genetischen Operatoren. Der Crossoveroperator wurde hier nur zur Veranschaulichung des Problems genutzt. Es ist daher ratsam auch für ähnliche Problemstellungen nicht schon vorhandene Operatoren weiter zu verwenden. Vielmehr sollten problemspezifische Operatoren verwendet und entwickelt werden.

4. Fazit

Mit Hilfe des Sudoku-Solvers wurde gezeigt, dass das DGPF auch für Problemstellungen genutzt werden kann, für die es nicht ausgelegt ist. Jedoch ist der Solver nicht soweit ausgereift, dass er für den täglichen Gebrauch genutzt werden kann.

Es wurde gezeigt, dass das DGPF eine effiziente Umgebung ist, die sich auch mit Implementierungen in C vergleichen kann. Keiner der zwei Frameworks lässt sich eindeutig zum Sieger erklären, jedes Framework konnte zwei Tests für sich entscheiden. Es wurde vor Allem die Erkenntnis gewonnen, dass die Wahl der genetischen Operatoren einen ernormen Einfluss auf die Konvergenz des Suchalgorithmus hat. Des Weiteren haben die Tests gezeigt, dass Genetische Algorithmen nicht bei jeder Problemstellung den einfacheren Suchalgorithmen überlegen sind. Hiermit wird die These des Betreuers gestützt, dass gemischt-kooperative Suchverfahren, die sich z.B. aus Genetischen Algorithmen und Simulated Annealing zusammensetzten, oftmals von Vorteil sein können.

5. Abbildungsverzeichnis

6. Quellenangabe

[1] Thomas Weise, Kurt Geihs, "DGPF - An Adaptable Framework for Distributed Multi-Objective Search Algorithms Applied to the Genetic Programming of Sensor Networks", Proceedings of "BIOMA 2006, The 2nd International Conference on Bioinspired Optimization Methods and their Applications", Ljubljana 2006, to appear

[2] Thomas Weise, Kurt Geihs, "Genetic Programming Techniques for Sensor Networks", Proceedings of 5. GI/ITG KuVS Fachgespräch "Drahtlose Sensornetze", 2006

[3] "Distributed Genetic Programming Framework" - SourceForge project, see http://sourceforge.net/projects/dgpf <http://sourceforge.net/projects/dgpf> and http://dgpf.sourceforge.net/ <http://dgpf.sourceforge.net/>

[4] Wikipedia, http://de.wikipedia.org/wiki/Sudoku

[5] Marc Kirchhoff, “Implementierung von Simulated Annealing in einem DGPF“, Kassel, August 2006

[6] Marion Riedel, Diplomarbeit “Parallele Genetische Algorithmen mit Anwendungen“, Chemnitz, Oktober 2002