Inhaltsverzeichnis

1 Einleitung **1**

2 Optimale Harmonie in Musik finden **2**

3 Harmony Search Algorithmus **5**

3.1 Begriffe des Harmony Search Algorithmus 5

3.2 Schritte des Harmony Search Algorithmus 6

3.2.1 parameter initialization 7

3.2.2 harmony memory initialization 8

3.2.3 new harmony improvisation 9

3.2.4 harmony memory update 11

3.2.5 termination criterion check 11

3.3 Beispiel einer Bewertungsfunktion 12

3.4 Beispiel zum Durchlauf des Harmony Search Algorithmus 12

3.5 Vergleich mit anderen Metaheuristiken 15

3.5.1 Vergleich mit Genetischen Algorithmen 15

3.6 Implementierung in Java anhand der Rosenbrock Funktion 15

3.6.1 Testlauf 1 16

3.6.2 Testlauf 2 17

3.6.3 Testlauf 3 17

3.6.4 Testlauf 4 18

3.6.5 Testlauf 5 19

A Anhang **I**

Glossar **V**

Stichwortverzeichnis **V**

Abbildungsverzeichnis

2.1 3 Musikinstrumente Improvisieren eine Tonkombination 2
2.2 die Tonkombination der 3 Musikinstrumente auf der Tonleiter 3
2.3 die Tonkombination der 3 Musikinstrumente als Dreiklang 3

3.1 Begriffe des Harmony Search Algorithmus 1 5
3.2 Parameter Initialisierung 7
3.3 Harmony Matrix 8
3.4 Harmony Search - Neue Improvisation 1 13
3.5 Harmony Search - Neue Improvisation 2 13
3.6 Harmony Search - Neue Improvisation 3 14
3.7 Harmony Search - Neue Improvisation 4 14

1 Einleitung

Bei vielen existierenden Problemen handelt es sich um Optimierungsprobleme. Um diese zu lösen, können traditionelle mathematische Techniken angewendet werden. Diese Techniken haben jedoch ihre Nachteile bei der praktischen Anwendung. Globale Optimas werden zwar ermittelt, jedoch ohne Beachten der Laufzeit des Algorithmus. Allein durch eine Erhöhung der zu betrachtenden Variablen eines Problems könnte die Laufzeit so stark ansteigen, dass ein Auswerten in annehmbarer Zeit nicht mehr möglich ist (Fluch der Dimensionalität). Denkbar wären auch Fälle, wo eine Funktion für die Bestimmung des globalen Optimums nicht differenzierbar ist. Auch dann könnte die mathematische Technik kein Ergebnis liefern.

Das globale Optimum für solche Probleme kann durch metaheuristische Algorithmen verlässlich und in annehmbarer Zeit angenähert werden.

Der Algorithmus der hier vorgestellt werden soll ist Harmony Search. Er wurde von Dr. Zong Woo Geem entwickelt und ist ein recht neuer Algorithmus (erschienen im Jahr 2000). Wie viele andere metaheuristische Algorithmen hat auch Harmony Search seinen Ansatz im Nachbilden von Vorgängen aus der Natur. Er stammt aus den Bereichen "Soft Computing" und "Evolutionäre Algorithmen" und wurde durch den Prozess des Improvisierens unter Musikern inspiriert. Jeder Musiker spielt eine Note und alle Musiker zusammen versuchen eine beste Harmonie zu erreichen.

Dieser Ansatz des Findens einer optimalen Harmonie unter Musikern kann dann auch auf andere Probleme der Praxis (Rosenbrock Funktion, Fahrplanerstellung, Travelling Salesman, Wasserversorgungssysteme...) angewandt werden.

2 Optimale Harmonie in Musik finden

Ich werde hier nun die Vorgehensweise des Harmony Search Algorithmus als erstes anhand seiner eigentlichen Problemstellung / dem Suchen nach der optimalen Harmonie unter Musikern, die miteinander Musik improvisieren beschreiben. Hierzu die Anmerkung, dass diese Beschreibung nur eine Veranschaulichung ist und die Suche nach der optimalen Harmonie unter Musikern nicht auch für ein “optimales” Musikstück stehen soll oder kann. Wie in der folgenden Abbildung 2.1 ersichtlich, betrachtet der Harmony Search Algorithmus in seinem Basiskonzept eine Reihe von Musikinstrumenten (hier: Doppelbass, Gitarre, Saxophon).

Instrument	Tonhöhe	Tonhöhe
	**E** ⇨ **3** (82 Hz) **F** ⇨ **4** (87 Hz) **G** ⇨ **5** (98 Hz)	**X1** = {3,4,5}
	**G** ⇨ **5** (98 Hz) **A** ⇨ **6** (110 Hz) **H** ⇨ **7** (123 Hz)	**X2** = {5,6,7}
	**C‘** ⇨**8** (130 Hz) **D‘** ⇨ **9** (147 Hz) **E‘** ⇨ **10** (165 Hz)	**X3** = {8,9,10}
x = {3, 5, 8 } ⇨ f (**x**) = f (3, 5, 8)		

Abbildung 2.1: 3 Musikinstrumente Improvisieren eine Tonkombination

Jedes dieser Musikinstrumente hat eine Menge von Tonhöhen (z.B. **X3** = {C',D',E'} für das Saxophon), die auf diesem speziellen Musikinstrument spielbar sind. Eigentlich umfasst das Saxophon in der realen Welt natürlich nicht nur 3 Tonhöhen sondern in etwa 2 1/2 Oktaven. Wir betrachten hier auch nur die Stammtöne der westlichen Musik (E,F,G,A,H,C,D) damit das Beispiel möglichst gut nachvollziehbar ist.

Wenn diese drei Instrumente nun zusammen Musik improvisieren, spielt jedes Instrument einen Ton aus seinem Tonrepertoir an. Im Beispiel spielt der Doppelbass ein E, die Gitarre ein G und das Saxophon ein C'. Es heißt hier deshalb C', da dieser Ton des Saxophons aus der nachfolgenden Oktave stammt (siehe Abbildung 2.2). Das Saxophon spielt also einen recht hohen Ton, während der Doppelbass einen ziemlich tiefen Ton spielt.

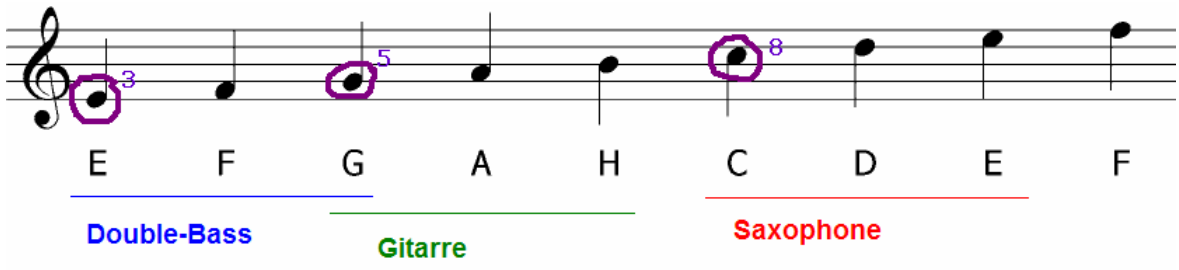

Abbildung 2.2: die Tonkombination der 3 Musikinstrumente auf der Tonleiter

Diese Tonkombination der drei Instrumente (E,G,C') wird nun im Vektor **x** gespeichert. Dieser Vektor kann durch eine Bewertungsfunktion (objective function) ausgewertet werden. Diese Bewertungsfunktion beurteilt dann, wie harmonisch diese Tonkombination ist.

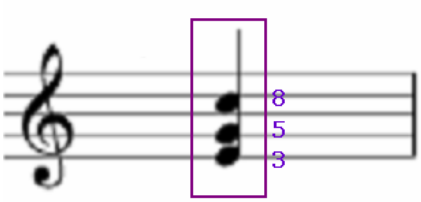

Abbildung 2.3: die Tonkombination der 3 Musikinstrumente als Dreiklang

In den Quellen auf die ich mich hier beziehe[3] wird auch auf das Verletzen der Harmonie eingegangen. Beispielsweise gibt es in bekannten und beliebten Musikstücken auch oft Verletzungen der optimalen Harmonie (z.B. bei Beethoven, der sogenannte Parallel Fifth verwendete). Der Algorithmus ist in seiner aktuellen Form nicht dazu gedacht, auch

Missklänge in einem größeren Kontext des Musikstücks als harmonisch zu bewerten. Er kann aktuell lediglich anhand europäischem Harmonieempfinden bewerten. Trickreiche Jazz-Improvisationen sind hier also nicht berücksichtigt sondern eher aufeinander folgende harmonische Tonkombinationen wie sie beispielsweise bei Musik von J. Sebastian Bach üblich ist. Es kommt natürlich auch ganz auf die Bewertungsfunktion an, in wie weit der Algorithmus eine Tonfolge als harmonisch bewertet.

3 Harmony Search Algorithmus

3.1 Begriffe des Harmony Search Algorithmus

Als erstes werde ich nun die verschiedenen Begriffe des Harmony Search Algorithmus beschreiben. Diese Begriffe orientieren sich an der im vorherigen Kapitel beschriebenen Problemstellung und sind deshalb in der Literatur auch teilweise mit Begriffen aus der Musikwelt belegt worden.

f (**x**)	objective function
x	solution vector
x1`, x2`, x3`, ... , xN`	decision variables (xi` ∈ **Xi**, xi` ∈ **x**)
X1, **X2**, **X3**, ... , **XN**	set of possible range of values for each decision variable (Menge der Tonhöhen eines Instruments)
x1(1), x2(1), x3(1), ... , xN(1) x1(2), x2(2), x3(2), ... , xN(2) ... x1(K), x2(K), x3(K), ... , xN(K)	candidate values (Tonhöhen eines Instruments) xi(1) < xi(2) < ... < xi(K)

Abbildung 3.1: Begriffe des Harmony Search Algorithmus 1

Die Bewertungsfunktion (objective function) bewertet den Vektor **x** (solution vector) und ermittelt als Ergebnis, wie nahe dieser Vektor am globalen Maximum/globalen Minimum dieser Funktion liegt.

Der Vektor $\mathbf{x}$ (solution vector) besteht aus N improvisierten Tonhöhen (decision variables). Wobei mit N hier die Anzahl der Musikinstrumente gemeint ist, die an der Improvisation beteiligt sind. Für jedes Musikinstrument existiert also genau eine improvisierte Tonhöhe im Vektor $\mathbf{x}$

Jedes Musikinstrument wiederum hat eine Menge an Tonhöhen, die es spielen kann (set of possible range of values for each decision variable). Jede Tonhöhe aus dieser Menge ist ein Kandidat für die Improvisation. Daher nennt man die Tonhöhen aus dieser Menge auch 'candidate values'.

3.2 Schritte des Harmony Search Algorithmus

Der Algorithmus hat die folgenden 5 Schritte, auf die ich jetzt näher eingehen werde:

- parameter initialization
- harmony memory initialization
- new harmony improvisation
- harmony memory update
- termination criterion check

3.2.1 parameter initialization

Der Algorithmus startet nun damit, einige wichtige Parameter zu initialisieren.

HMS	harmony memory size (number of simultaneous solution vectors **x** in harmony memory)
HMCR	harmony memory considering rate (probability) example: 0.9
PAR	pitch adjusting rate (probability) example: 0.5
number of improvisations	number of objective function evaluations
N	number of instruments

Abbildung 3.2: Parameter Initialisierung

Um die 'harmony matrix' zu erzeugen, benötigt er als Angabe über die Größe dieser Matrix den Parameter 'harmony memory size'. Dieser Parameter gibt nun an, wieviele Vektoren **x** (solution vector) sich in der 'harmony matrix' gleichzeitig befinden dürfen. Auf die Bedeutung dieser Matrix wird später noch näher eingegangen.

Weitere Parameter sind die 'harmony memory considering rate' und die 'pitch adjusting rate'. Beides sind Probabilitäten und werden daher im Bereich von 0 bis 1 angegeben. Sie geben die Warscheinlichkeit an, mit welcher der Algorithmus eine bestimmte Entscheidung trifft. Ein Feintuning dieser beiden Parameter durch den Benutzer ist meist erforderlich, damit der Algorithmus schnell zu einem guten Optimum gelangt.

Als letztes benötigt der Algorithmus nun noch die Anzahl der Durchläufe (number of improvisations). Diese gibt an, wie oft der Algorithmus einen neuen Vektor **x** (solution vector) improvisieren soll, bevor er das Ergebnis ausgibt.

3.2.2 harmony memory initialization

Angelehnt an die Inspiration aus der Musikwelt, besitzt der Harmony Search Algorithmus eine sogenannte 'harmony matrix'. Diese Matrix stellt das Gedächtnis des Musikers dar, der beim Improvisieren von Tönen entweder einen Ton aus seinem Gedächtnis spielt oder sich diesen wirklich neu ausdenkt und ihn in sein Gedächtnis aufnimmt (Improvisation).

Diese Harmony Matrix wird nun mit 'harmony memory size' mal zufälligen Vektoren **x** initialisiert. Die 'decision variables' der Vektoren **x** wurden hierzu aus den 'candidate values' der Menge der Tonhöhen zufällig ausgewählt.

Jeder Vektor **x** (solution vector) ist in dieser Matrix eine Zeile und besteht aus N 'decision variables'. In der folgenden Abbildung sieht man auch, dass die Harmony Matrix für jeden Vektor **x** das Ergebnis der 'objective function' speichert, denn dieses Ergebnis benötigt der Algorithmus für die weitere Bewertung.

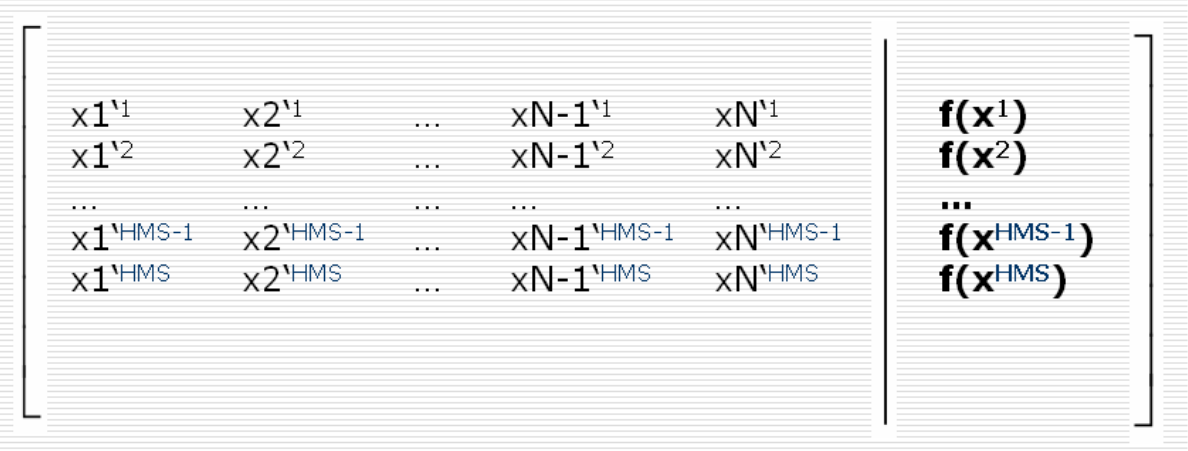

Abbildung 3.3: Harmony Matrix

Sobald der Algorithmus nun startet, improvisiert er 'number of improvisations' mal einen neuen Vektor **x** (solution vector). Diese einzelnen Improvisationen erfolgen nun jedoch nicht mehr zufällig, wie beim Erzeugen der 'harmony matrix', sondern anhand den vorher beschriebenen Regeln. Es wird entweder eine Tonhöhe aus dem Gedächtnis gespielt oder eine neue Tonhöhe improvisiert. Das Gedächtnis ist hierbei die 'harmony matrix'.

3.2.3 new harmony improvisation

ein neuer Vektor **x** = (x1', x2',..., xN') wird nun improvisiert. Jede seiner 'decision variables' kann nun nach den folgenden Fällen erzeugt werden:

- Mit einer Probabilität von 1-HMCR wird zur Erzeugung der 'decision variable' der Fall 'random selection' gewählt. (Beispiel: 1-0.9 = 0.1, also 10 Prozent Warscheinlichkeit). Dieser Fall entspricht dem Improvisieren einer neuen Tonhöhe für das entsprechende Instrument.

- Mit einer Probabilität von HMCR wird zur Erzeugung der 'decision variable' der Fall 'harmony memory consideration' gewählt (Beispiel: 0.9, also 90 Prozent Warscheinlichkeit). Dieser Fall entspricht dem Spielen der Tonhöhe für das entsprechende Instrument aus dem Gedächtnis.

3.2.3.1 Vorgehensweise bei 'random selection'

die 'decision variable' xi' aus **x** wird zufällig aus der Menge der Tonhöhen **XI** des zugehörigen Instruments gewählt (improvisiert).

$$xi' \in \mathbf{XI} \Rightarrow xi' \in \{xi(1), xi(2), \ldots, xi(K)\}$$

3.2.3.2 Vorgehensweise bei 'harmony memory consideration'

die 'decision variable' xi' aus **x** wird aus der Menge der Tonhöhen **XI** gewählt, die sich bereits für das Instrument von xi' im 'harmony memory' befinden (aus dem Gedächtnis spielen).

$$xi' \in \{xi'^1, xi'^2, \ldots, xi'^{HMS}\}$$

In der Literatur steht nirgens explizit, wie diese Tonhöhe aus der Menge der bereits vorhandenen Vektoren **x** / aus der 'harmony matrix' gewählt wird. Ich gehen also im folgenden davon aus, dass ein **zufälliger** Vektor **x** aus der 'harmony matrix' gewählt wird, aus dem diese 'decision variable' dann gewählt wird. Genauso könnte natürlich auch gelten, dass die 'decision variable' immer aus dem Vektor mit dem besten Funktionswert gewählt wird.

3.2.3.3 pitch

Für den Fall der 'harmony memory consideration' ist zudem noch vorgesehen, dass der Ton, der aus dem Gedächtnis gespielt wurde noch leicht variiert werden kann. Dies entspricht in etwa dem Konzept der Mutation das bei Genetischen Algorithmen angewandt wird. Auch hier gibt es wieder eine Unterscheidung:

- Mit einer Probabilität von 1-PAR wird die Tonhöhe nicht variiert (Beispiel: 1 - 0.5, 50 Prozent Warscheinlichkeit).

- Mit einer Probabilität von PAR wird die Tonhöhe variiert (Beispiel: 0.5, 50 Prozent Warscheinlichkeit).

Sollte die Tonhöhe variiert werden, gilt eben nun, dass die variierte Tonhöhe aus der unmittelbaren Umgebung der gewählten Tonhöhe stammen muss. Je nach Problemstellung sollte hier in unterschiedlicher Weise ein Nachbarwert gewählt werden, da die unmittelbare Umgebung sich bei einer eher mathematischen Problemstellung wie das Bestimmen der Rosenkreuz Funktion im Nachkommastellenbereich befindet (Beispiel: +0.000001 oder -0.000001). Bei einer anderen Problemstellung kann jedoch vielleicht direkt im Dezimalbereich gewählt werden (Beispiel: +1,-1). Die aktuelle Tonhöhe der 'decision variable' des Vektors **x** lässt sich ja wie folgt definieren:

$$xi' \Leftrightarrow xi(k), xi \in \mathbf{XI}$$

Denn xi' stammt ja aus der Menge der Tonhöhen **XI** für ein Instrument, ist also eine bestimmte Tonhöhe k aus dieser Menge. Die improvisierte Tonhöhe xi'new ist nun eben xi' an der Stelle k in der Menge der Instrumente jedoch verschoben um einen Faktor m, der zum Beispiel +1 oder -1 betragen kann:

$$xi'new = xi(k + m), m \in \{-1, +1\}$$

3.2.4 harmony memory update

Für diesen neu improvisierten Vektor **x** gilt nun: Wenn der neu improvisierte Vektor **x** besser ist als der schlechteste Vektor in der 'harmony matrix', behalte **x** und verwerfe diesen schlechtesten Vektor.

Aus der Sicht des Musikers heißt das, der Musiker findet seine neue Improvisation so gut dass er sie sich merkt, aber dafür eine alte Improvisation vergisst, die nicht so harmonisch war.

Als Kriterium für die Beurteilung für gute/schlechte Vektoren in der 'harmony matrix' wird die Bewertungsfunktion (objective function) verwendet. Auf die Bewertungsfunktion wird später nochmal genauer eingegangen.

3.2.5 termination criterion check

Wenn nun das Abbruchkriterium erreicht ist (number of improvisations), kann abgebrochen werden. Ansonsten wird der nächste Vektor **x** improvisiert. Hierzu führt der Algorithmus wieder den Schritt 'new harmony improvisation' und alle Folgeschritte aus.

3.3 Beispiel einer Bewertungsfunktion

Eine gültige Bewertungsfunktion für die Suche nach einer optimalen Harmonie könnte wie folgt lauten:

$$Minf(\mathbf{x}) = (x1 - 5)^2 + (x2 - 7)^4 + (x3 - 9)^2 + 3$$

Diese Funktion bewertet nun genau eine Kombination von Tonhöhen (5,7,9) als optimal. Alle Vektoren **x** die für diese Funktion möglichst minimale Ergebnisse erzielen, werden besser bewertet. Bei der Suche um eine optimale Harmonie in der Musik handelt es sich um ein 'multi-objective-problem'. Daher wäre diese Funktion noch nicht ausreichend und müsste noch erweitert werden um auch andere Kombinationen von Tönen als Optimas zu erkennen, sie sollte aber hier ausreichen.

3.4 Beispiel zum Durchlauf des Harmony Search Algorithmus

Für den folgenden Beispieldurchlauf einer Iteration des Harmony Search Algorithmus wird die im vorigen Schritt beschriebene Bewertungsfunktion verwendet. Die 'harmony matrix' wurde für drei Musikinstrumente (**X1**, **X2**, **X3**) erzeugt.

Für HMS wurde der Wert 3 gewählt, es sind daher 3 mit zufälligen Tonhöhen initialisierte Vektoren **x** in der 'harmony matrix'. Für jeden Vektor **x** wurden auch der zugehörige Funktionswert der Bewertungsfunktion bestimmt. Der Wert für 'number of improvisations' beträgt 4.

In der nächsten Abbildung sieht man nun, wie ein neuer Vektor **x** improvisiert wird. Die erste 'decision variable' wurde mittels 'random selection' komplett neu improvisiert und beträgt 5. Die zweite 'decision variable' wurde mittels 'harmony memory consideration' bestimmt und daher aus einem zufällig gewählten Vektor **x** der 'harmony matrix' gewählt.

Vektor	x1'	x2'	x3'	F
x^1	4	7	9	4
x^2	3	5	8	24
x^3	3	6	10	9
Improv. x				

Abbildung 3.4: Harmony Search - Neue Improvisation 1

In diesem Fall ist es der Vektor x^3 mit der 'decision variable' 6. Die dritte 'decision variable' wurde ebenfalls mittels 'harmony memory consideration' gewählt und zwar aus dem ersten Vektor x^1 mit der 'decision variable' 9. Dieser Wert wurde jedoch noch durch die Regel für 'pitch' um -1 auf 8 variiert. Der neu improvisierte Vektor erhält durch die Bewertungsfunktion 5 als Funktionswert.

Vektor	x1'	x2'	x3'	F
x^1	4	7	9	4
x^2	3	5	8	24
x^3	3	6	10	9
Improv. x	**5 (rand)**	**6 (hm)**	**8 (pitch)**	**5**

Abbildung 3.5: Harmony Search - Neue Improvisation 2

Die nächste Abbildung zeigt nun wie der schlechteste Vektor **x** aus der 'harmony matrix' gelöscht wird und durch den neu improvisierten Vektor ersetzt wird, da dieser einen besseren Funktionswert besitzt.

Vektor	x1'	x2'	x3'	F
x^1	4	7	9	4
~~x^2~~	~~3~~	~~5~~	~~8~~	~~24~~
x^3	3	6	10	9
Improv. x	**5 (rand)**	**6 (hm)**	**8 (pitch)**	**5**

Abbildung 3.6: Harmony Search - Neue Improvisation 3

Die neue 'harmony matrix' ist nun in der folgenden Abbildung ersichtlich. Der Wert für 'numbers of improvisations' kann nun um eins verringert werden und ist daher jetzt 3. Folglich können noch 3 weitere Improvisationen folgen bis der Algorithmus terminiert.

Vektor	x1'	x2'	x3'	F
x^1	4	7	9	4
x^2	5	6	8	5
x^3	3	6	10	9
Improv. x				

Abbildung 3.7: Harmony Search - Neue Improvisation 4

3.5 Vergleich mit anderen Metaheuristiken

3.5.1 Vergleich mit Genetischen Algorithmen

Ich kann hier leider nur Vorteile beschreiben die ich der Literatur entnehmen konnte, da in den Papers zu diesem Algorithmus keine Nachteile genannt wurden. Ein Nachteil wäre jedoch gut denkbar. Der Algorithmus ist vom Design her statisch konzipiert. Parameter lassen sich vom Konzept des Algorithmus her nicht im laufenden Programm anpassen, wie es bei genetischen Algorithmen der Fall sein kann. Dadurch kann der Algorithmus bestimmte Problemstellungen nicht perfekt ausschöpfen. Soll heißen: Exploration und Exploitation werden bei ihm eher gleich behandelt. Hier einige Vorteile:

- Schnellere Berechnung - Eine Iteration in Harmony Search ist schneller als eine Iteration in einem genetischen Algorithmus. So steht es in der Literatur. Ob das jedoch unbedingt ein Vorteil ist, bleibt fraglich.

- keine Binär/Dezimalkonversion wie bei Genetischen Algorithmen erforderlich.

- keine so hohe Komplexität wie genetische Algorithmen. Kompakter Algorithmus mit einfacher Vorgehensweise. Diesen Vorteil habe ich nicht aus der Literatur entnommen, aber ich empfand es als größten Vorteil dass er einfach zu implementieren war.

3.6 Implementierung in Java anhand der Rosenbrock Funktion

Ich habe den Harmony Search Algorithmus in Java implementiert und auf die bekannte Rosenbrock Funktion angewandt die oft für Optimierungsprobleme verwendet wird. Die Rosenbrock Funktion ist wie folgt definiert:

$$f(x, y) = (1 - x)^2 + 100 * (y - x^2)^2$$

Ihr globales Minimum befindet sich bei x=-1 und y=+1. Der komplette Java Code zu dem Programm ist im Anhang ersichtlich.

Für das Programm habe ich einige Testläufe mit verschiedenen Werten für die Parameter durchgeführt. Das Programm wurde bei jedem Testlauf 500 mal gestartet um den besten Wert aus diesen 500 Durchläufen zu bestimmen. Für die Laufzeitberechnung wurde die gesamte Laufzeit für alle 500 Durchläufe bestimmt und durch 500 geteilt (arithmetisches Mittel). Dies war unter anderem deshalb nötig um die Zeitmessungen unter Java besser zu mitteln. Bedingt durch den 'Garbage Collector' wird die Zeitmessung leider leicht verfälscht. Durch Wahl des gemittelten Zeitwertes kann dies ausgeglichen werden.

3.6.1 Testlauf 1

Einstellungen:

- number of improvisations = 50000
- harmony memory size = 20
- HMCR = 0.9
- PAR = 0.05

Ergebnis:

- x = 1.0001506
- y = 1.0003451
- arithmetisches Mittel der 500 Durchläufe = 77 ms

3.6.2 Testlauf 2

Einstellungen:

- number of improvisations = 50000
- harmony memory size = 20
- HMCR = 0.9
- PAR = 0.0001

Ergebnis:

- x = 1.0002527
- y = 1.0005099
- arithmetisches Mittel der 500 Durchläufe = 101 ms

ein extremes Herabsetzen des PAR Parameters auf 0.0001 brachte ein leicht schlechteres Gesamtergebnis im Vergleich zum vorherigen Testlauf.

3.6.3 Testlauf 3

Einstellungen:

- number of improvisations = 50000
- harmony memory size = 20
- HMCR = 0.9

- PAR = 0.3

Ergebnis:

- x = 1.0001014
- y = 1.0002049
- arithmetisches Mittel der 500 Durchläufe = 85 ms

ein Heraufsetzen des PAR Parameters auf 0.3 bringt ein leicht besseres Ergebnis im Vergleich zu Testlauf 1.

3.6.4 Testlauf 4

Einstellungen:

- number of improvisations = 50000
- harmony memory size = 10
- HMCR = 0.9
- PAR = 0.3

Ergebnis:

- x = 1.0000012
- y = 1.0000024
- arithmetisches Mittel der 500 Durchläufe = 73 ms

Durch ein Verringern der 'harmony memory size' und Verwendung des PAR Werts aus Testlauf 3 wurden die Ergebnisse hier genauer und die Laufzeit auch etwas besser als bei Testlauf 1.

3.6.5 Testlauf 5

Einstellungen:

- number of improvisations = 90000
- harmony memory size = 10
- HMCR = 0.9
- PAR = 0.3

Ergebnis:

- x = -1.000000
- y = 1.000000
- arithmetisches Mittel der 500 Durchläufe = 90 ms

durch die erhöhte Anzahl an Improvisationen (90000) und Verwendung der Einstellungen aus Testlauf 4 wurde das Ergebnis nochmal etwas besser, aber die Laufzeit verschlechterte sich leicht.

Insgesamt sind die Testergebnisse bei Testlauf 5 durch die erhöhte Anzahl Improvisationen generell besser. Jedoch ist eben auch die Laufzeit etwas höher. Der Algorithmus hat es hier bei einem der 500 Durchläufe sogar geschafft, das globale Optimum auf 6 Nachkommastellen genau zu finden.

Ein Verändern des HMCR Wertes erwies sich auch nicht als förderlich. 0.9 war schon recht optimal für diese Problemstellung gewählt.

A Anhang

```
public class HarmonySearch {

  private static int HMS = 10;
  private static double HMCR = 0.9;
  private static double PAR = 0.05;
  private static int number_of_improvisations = 100000;
  private static int N = 2;

  private Random rand = new Random(System.currentTimeMillis());


  public class SolutionVector
  {
    public Float[] decisionVariables;
    public Float objectiveFunctionValue;
  }


  private Float objectiveFunction(SolutionVector vector)
  {
    Float x1 = vector.decisionVariables[0];
    Float x2 = vector.decisionVariables[1];

    // minimize the rosenbrock function
    return (1-x1*x1)*(1-x1*x1)+ 100*(x2-x1*x1)*(x2-x1*x1);
  }

  public float getRandomValue()
  {
    // get random float value between -2 and +2
    Float value = ((Integer)rand.nextInt(4)).floatValue();
    value = value -2;
    value = value + rand.nextFloat();
    return value;
  }

  private Float getDecisionVariableFromHarmonyMemory(SolutionVector
      [] harmonyMemory, int decisionVariableIndex)
```

```
{
  Float[] candidates = new Float[HMS];
  for (int i=0; i< HMS; i++)
  {
    candidates[i] = harmonyMemory[i].decisionVariables[
        decisionVariableIndex];
  }

  // return a randomly chosen decision variable from
  // the harmonyMemory for a specified
  // decision variable index
  return candidates[rand.nextInt(HMS)];
}


public Boolean isRandomSelection()
{
  // check with probability, if the new decision
  // variable will be random or from the harmony memory
  Float value = rand.nextFloat();
  if (value < HMCR)
    return false;
  else
    return true;
}

public Boolean isPitchAdjustment()
{
  // check with probability, if the new decision
  // variable should be pitched
  Float value = rand.nextFloat();
  if (value < PAR)
    return false;
  else
    return true;
}

private SolutionVector start()
{
  SolutionVector bestVector = new SolutionVector();

  // 1. create and initialize harmony memory

  SolutionVector[] harmonyMemory = new SolutionVector[HMS];
  for (int i=0; i < HMS; i++)
  {
    harmonyMemory[i] = new SolutionVector();
    harmonyMemory[i].decisionVariables = new Float[N];

    for (int j=0; j < N; j++)
```

```
    harmonyMemory[i].decisionVariables[j] = getRandomValue();

  harmonyMemory[i].objectiveFunctionValue = objectiveFunction(
      harmonyMemory[i]);
}

// 2. improvise new harmony vector
int count = 0;
while (count < number_of_improvisations)
{
  count++;

  SolutionVector improvise = new SolutionVector();
  improvise.decisionVariables = new Float[N];

  // create decision variables of new harmony vector
  for (int i=0; i<N; i++)
  {
    if (isRandomSelection())
    {
      // case 1: random selection
      improvise.decisionVariables[i] = getRandomValue();
    }
    else
    {
      // case 2: hm consideration
      improvise.decisionVariables[i] =
          getDecisionVariableFromHarmonyMemory(harmonyMemory, i);

      if (isPitchAdjustment())
      {
        // pitch adjustment +1 or -1
        if (rand.nextBoolean())
          improvise.decisionVariables[i] = improvise.
              decisionVariables[i] + 0.0000001F;
        else
          improvise.decisionVariables[i] = improvise.
              decisionVariables[i] - 0.0000001F;
      }
    }
  }

  // get the objective function value for the new
  // improvised harmony vector

  improvise.objectiveFunctionValue = objectiveFunction(improvise
      );

  // harmony memory update. If the new improvised vector
  // is better than the worst vector in the harmony memory,
```

```
        // keep the new vector and drop the old

        Float worstValue = 0F;
        int worstVectorPos = 0;
        for (int i=0; i < HMS; i++)
        {
          if (harmonyMemory[i].objectiveFunctionValue > worstValue)
          {
            worstValue = harmonyMemory[i].objectiveFunctionValue;
            worstVectorPos = i;
          }
        }

        if (improvise.objectiveFunctionValue < worstValue)
        {
          harmonyMemory[worstVectorPos] = improvise;
          bestVector = improvise;
        }
      }

      return bestVector;
    }
  }
```

Literaturverzeichnis

[1] Kim Tae Gyun Geem, Zong Woo and Joong Hoon Kim. Optimal layout of pipe networks using harmony search. *International Conference on Hydro-Science and -Engineering*, page 8, 2000.

[2] Zong W. Geem, Joong H. Kim, and G. V. Loganathan. A new heuristic optimization algorithm: Harmony search. *SIMULATION*, 76(2):60–68, February 2001.

[3] Yongjin Park Zong Woo Geem, Kang Seok Lee. Application of harmony search to vehicle routing. *American Journal of Applied Sciences*, 2:6, 2005.