Studienarbeit

Evolutive Optimierung von L-System-generierten Pflanzen

-

Das Fitting von L-Systemen zur Simulation konkreter, vorgegebener Morphologien

von Mathias Radicke, Informatikstudent, 11. Semester

BTU Cottbus

April 2004

INHALTSVERZEICHNIS

Zu dieser Studienarbeit gehört zusätzlich eine CD mit dem Programm, dem Quellcode und einer ausführlichen Dokumentation des Quellcodes.

1. Einleitung

Das Ziel dieser Studienarbeit war die Entwicklung einer Software, die L-Systeme entwickelt und evolutionär optimiert, um damit vorgegebene Morphologien nachzubilden.

Dazu sollten genetische Algorithmen entwickelt werden, die eine genetische Evolution simulieren und die die evolutionäre Optimierung bewerkstelligen. Die L-Systeme sollten so codiert werden, dass sie für genetische Algorithmen anwendbar sind. Das bedeutet, dass die neue Codierung der L-Systeme genetische Mechanismen, wie zum Beispiel Mutation oder Rekombination, unterstützt oder fördert. Um die evolutionär entwickelten L-Systeme zu bewerten, sollte die Morphologie der interpretierten L-Systeme analysiert werden. Mit Hilfe aussagekräftiger Fitnessfunktionen sollte jedem entwickelten L-System ein Fitnesswert zugeordnet werden, um damit die genetische Selektion zu bewerkstelligen.

Genetische Algorithmen erschienen für die Erschaffung von L-Systemen nicht so gut geeignet wie genetische Programmierung. Da die genetische Programmierung hervorragend auf baumartige Strukturen anzuwenden ist und L-Systeme aufgrund ihrer Grammatik sehr gut baumartig dargestellt werden können, fiel die Wahl auf genetische Programmierung zur Simulation der Evolution.

Um nicht eine Software für genetische Programmierung neu zu schreiben, wurde eine Anwendung benutzt, die für diese Aufgabe entwickelt wurde und die es erlaubt, mit Hilfe diverser Parametereinstellungen und einer Problemformulierung Individuen zu erzeugen und diese zu evolvieren. Diese Anwendung heißt „ECJ – A Java-based Evolutionary Computation and Genetic Programming Research System“ und wurde in der Version 10 später Version 11 benutzt.

Des Weiteren musste eine Codierung der L-Systeme zur Verwendung bei der genetischen Programmierung vollzogen werden. Dazu wurde eine Grammatik in BNF erstellt, die jedoch nicht die komplette L-System-Grammatik umfasst. Diese Grammatik lässt erstens eine einfache Darstellung von L-Systemen als Bäume zu und zweitens ist diese Grammatik sehr einfach in die von „ECJ“ gewünschte Form von Parametern zu konvertieren.

Genauere Ausführungen zur genetischen Programmierung und „ECJ“ sind im Kapitel 2 nachzulesen.

In Kapitel 3 wird die verwendete Grammatik und die Codierung der L-Systeme beschrieben.

Um die L-Systeme miteinander zu vergleichen, mussten sie in irgendeiner Weise gemessen werden. Da vorgegebene Morphologien nachgebildet werden sollten, wurden die Morphologien der erzeugten L-Systeme analysiert. Dazu mussten die Regeln der L-Systeme angewendet werden und die dadurch erzeugte geometrische Struktur der Systeme musste statistisch analysiert werden. Dabei half die am Lehrstuhl entwickelte Anwendung „GroIMP“, die die geometrische Interpretation der L-Systeme vornahm.

Bei der Analyse der geometrischen Struktur wurden unter anderem die Längen der Linien der Struktur und deren Verteilung gemessen oder die Anzahl der Enden der Linien der Struktur.

Mit den Ergebnissen der Messungen der erzeugten L-Systeme und dem vorgegebenen System wurde bestimmt, wie nahe die erzeugten Systeme dem vorgegebenen System waren. Dazu wurden verschiedene Distanzfunktionen erstellt und ab-

hängig von der Entfernung vom vorgegebenen System wurden Fitnesswerte für die erzeugten L-Systeme gebildet. Mit Hilfe der Fitnesswerte erfolgte die Selektion der L-Systeme für die nächste Generation.

Eine genaue Beschreibung der Analyse ist im Kapitel 4 zu finden.

Die Fitnessfunktionen sind im Kapitel 5 im einzelnen beschrieben.

Die Software zur Erzeugung und evolutionären Optimierung von L-Systemen zur Nachbildung vorgegebener Morphologien – „Evolve“ – besteht aus mehreren Teilen:

- aus dem Teil, der Berechnungen durchführt,
- aus einem Teil, der die Evolution simuliert,
- aus einem Teil, der die L-Systeme interpretiert und zeigt und
- aus der grafischen Benutzeroberfläche.

Der Aufbau und die Funktionsweise von „Evolve“ werden in Kapitel 6 erklärt.

Die Anbindung von „Evolve“ an „GroIMP“ wird in Kapitel 7 und die Anbindung an „ECJ“ in Kapitel 8 erläutert.

In Kapitel 9 werden die Ergebnisse von Tests mit den Fitnessfunktionen vorgestellt. Es wurden unterschiedliche Gewichtungen einzelner Bestandteile der Fitnessfunktionen vorgenommen, um zu sehen, wie sich diese Veränderungen auf den Erfolg der Evolution auswirken.

Ebenso wurden einige Tests mit den Evolutionsparametern von „ECJ“ vorgenommen, um deren Auswirkungen auf die Evolution zu sehen. Diese Ergebnisse sind in Kapitel 10 nachzulesen.

2. Funktionsweise der genetischen Programmierung

Genetische Algorithmen und das genetische Programmieren sind Methoden, die der biologischen Evolution nachempfunden sind. Dabei können verschiedene noch nicht optimale Lösungen des Problems verändert werden, um neue - vielleicht bessere - Lösungen zu finden. Sie dienen meistens dazu, große Suchräume nach optimalen Lösungen zu durchsuchen.

Um Probleme mit Hilfe genetischen Programmierens zu lösen, müssen diese Probleme und ihre Lösungen in einer Form definiert werden, die es möglich macht, genetische Methoden darauf anzuwenden. Das bedeutet im Einzelnen:

Erstens: die Lösungen müssen in einer Art Genom codiert werden, welches - wie die Genome von Lebewesen - verändert werden kann. Die Veränderungen können punktuell geschehen wie bei einer Punktmutation oder durch die Kreuzung von zwei Genomen, wobei ganze Teile eines Genoms verändert werden können.

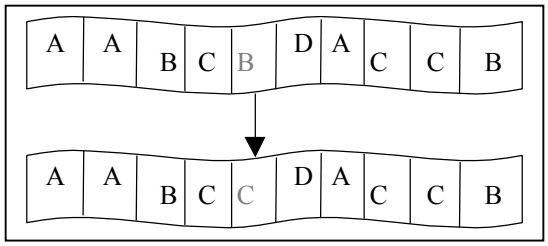


Abbildung 1: Punktmutation

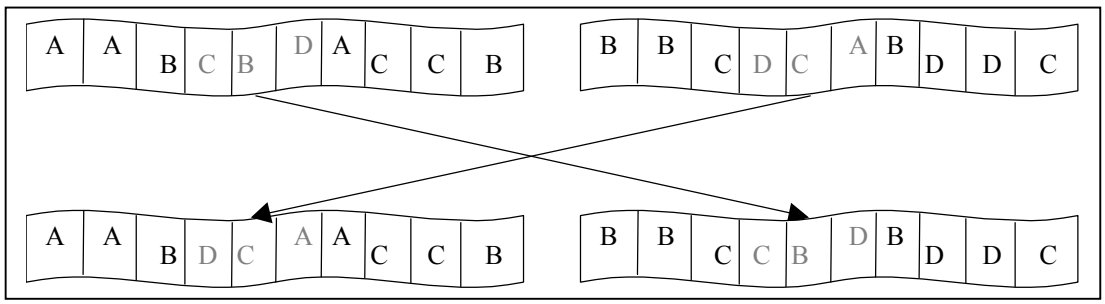


Abbildung 2: Kreuzung

Zweitens: für das Problem müssen Funktionen formuliert werden, die die Entfernung der erschaffenen Lösungen von der optimalen Lösung errechnen. Diese Funktionen nennt man Fitnessfunktionen.

Wenn diese beiden Voraussetzungen erfüllt sind, funktioniert das genetische Programmieren wie folgt. Es werden initiale Lösungen erschaffen und bilden die erste Generation von Lösungen. Die einzelnen Lösungen werden analysiert und mit ihrer Fitness (Entfernung zur optimalen Lösung) bewertet. Dann werden Lösungen an Hand bestimmter Verfahren ausgewählt, danach durch Mutation oder Kreuzung verändert und diese neuen Lösungen bilden die nächste Generation. Dann folgt wieder die Analyse und Bewertung der neuen Lösungen. Dieses Verfahren wiederholt sich, bis eine optimale Lösung gefunden wird.

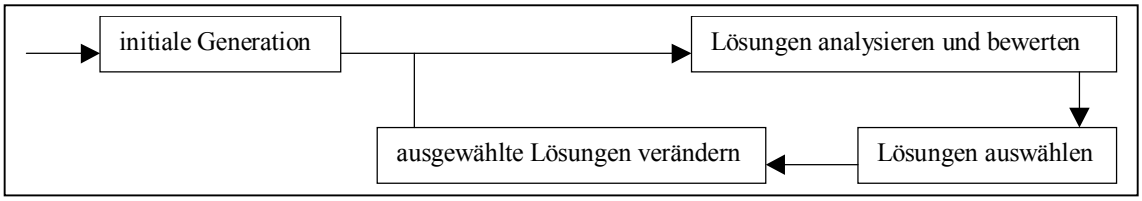


Abbildung 3: Schema der Funktionsweise der genetischen Programmierung

Beim genetischen Programmieren werden die Lösungen nicht wie in den Abbildungen 1 und 2 als Zeichenketten codiert, sondern die Lösungen werden etwas abstrakter als Bäume codiert. Eine Punktmutation verändert dann also einen Knoten des Baumes und eine Kreuzung zweier Bäume tauscht Unterbäume dieser beiden Individuen aus.

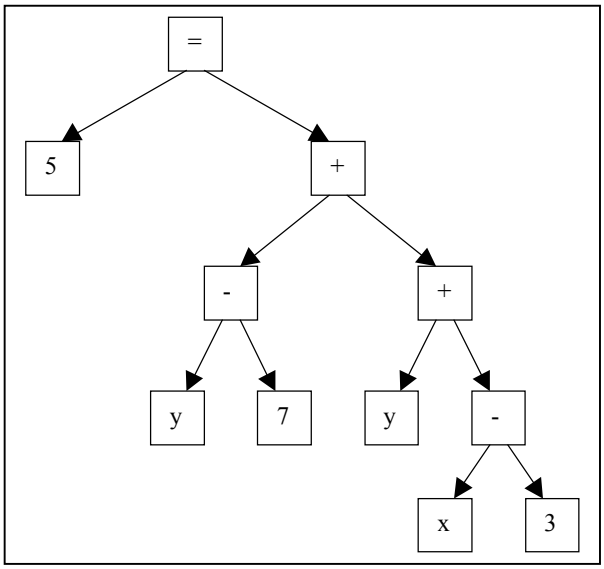


Abbildung 4: Codierung einer Lösung einer genetischen Programmierung

Die Anwendung „ECJ – A Java-based Evolutionary Computation and Genetic Programming Research System“, zu finden unter http://cs.gmu.edu/~eclab/projects/ecj/index.html, übernimmt einen großen Teil der genetischen Programmierung. Die oben genannten Voraussetzungen, nämlich die Lösungen zu codieren und das Problem und dessen Fitnessfunktionen zu formulieren, müssen selbst erstellt werden. Aber der Ablauf der genetischen Programmierung, wie in Abbildung 3 beschrieben, ist vollständig implementiert. Die Verfahren der initialen Erzeugung, der Selektion, der Mutation oder der Kreuzung sind in zahlreichen unterschiedlichen Varianten vorhanden. Die Parameter zur Beeinflussung der Evolution sind dazu leicht zu verändern.

„ECJ“ wurde anfangs in Version 10 und später in der aktuellen Version 11 eingesetzt. An den Evolutionsläufen der Software „Evolve“ hat sich zwischen beiden Versionen jedoch nichts geändert. Die Einbindung von „ECJ“ in „Evolve“ wird in Kapitel 8 noch genau beschrieben. Ausführliche Informationen zu „ECJ“ erhält man unter http://cs.gmu.edu/~eclab/projects/ecj/docs/index.html oder im „GroIMP“-Verzeichnis der CD unter ./ec/docs/index.html. Dort wird auch in 4 Tutorials sehr einfach, aber tiefgreifend auf die Funktionsweise von „ECJ“ eingegangen.

3. Codierung der L-Systeme

Wie im Kapitel 2 beschrieben, müssen die Lösungen eines Problems, das man mit Hilfe genetischer Programmierung lösen möchte, so codiert sein, dass Veränderungen an der Lösung leicht möglich sind. Beim genetischen Programmieren werden die Lösungen als Bäume codiert. Da die Lösungen in diesem Falle aus L-Systemen bestehen sollen, ist es notwendig, die L-Systeme als Bäume darzustellen und zu codieren.

Dazu wurde eine vereinfachte Grammatik für L-Systeme erstellt, mit der man leicht Bäume für konkrete L-Systeme bilden kann. Die Grammatik enthält nur Symbole der L-System-Sprache, die eine zweidimensionale Darstellung zulassen. Außerdem gibt es nur ein Symbol für die Längenkontraktion und es wird ein Standardwinkel verwendet. Alle diese Einschränkungen lassen sich durch einfache Erweiterungen der Grammatik beheben. Es war jedoch für den Umfang dieser Arbeit notwendig, einige Einschränkungen zu machen.

Im Folgenden ist genau beschrieben, wie ein L-System erschaffen wird. Die erstellte Grammatik in BNF wird aufgezeigt.

Jedes L-System startet mit dem String:

"\angle " + `angle` + ", * # a"

dann folgt ein String, der sich abhängig von `numberOfRules` bildet.

Bei `numberOfRules` = 1

", a # "

und dann folgt ein String, der sich nach folgender Grammatik bildet:

N = { Folge, FolgeGlied, Move, Stack, Plus, Minus, LHalf, F, Variable, Empty }
T = { [,], +, -, L*0.5, F, a }
S = { <Folge> }
P = { <Folge> → <FolgeGlied> <Folge> | <Empty>
<FolgeGlied> → <Move> | <Stack>
<Move> → <Plus> | <Minus> | <LHalf> | <F> | <Variable>
<Stack> → "[" <Folge> "]"
<Plus> → "+"
<Minus> → "-"
<LHalf> → "L*0.5"
<F> → "F"
<Variable> → "a"
<Empty> → "" }

Bei `numberOfRules` > 1

“““

und dann folgt ein String, der sich nach folgender Grammatik bildet:

N = { Folgen, Folge, FolgeGlied, Move, Stack, Plus, Minus, LHalf, F, Variable, Empty }
T = { [,], +, -, L*0.5, F, a, b, c, d, e }
S = { <Folgen> }
P = { <Folgen> → “, a # “ <Folge> (Anzahl der <Folge> = `numberOfRules`)
“, b # “ <Folge>
“, c # “ <Folge>
“, d # “ <Folge>
“, e # “ <Folge>
<Folge> → <FolgeGlied> <Folge> | <Empty>
<FolgeGlied> → <Move> | <Stack>
<Move> → <Plus> | <Minus> | <LHalf> | <F> | <Variable>
<Stack> → “[“ <Folge> “]”
<Plus> → “+”
<Minus> → “-”
<LHalf> → “L*0.5”
<F> → “F”
<Variable> → “a” | “b” | “c” | “d” | “e” (Anzahl der Alternativen = `numberOfRules`)
<Empty> → “” }

Man kann die obige Grammatik durch eine Vielzahl von Symbolen erweitern und man kann die Grammatik auch durch Hinzufügen weiterer Hilfssymbole exakter gestalten, aber man sollte bedenken, dass durch jedes weitere Symbol der Suchraum und damit die Suchzeit gegebenenfalls erheblich vergrößert werden.

Die Lösungen, die sich aus der obigen Vorschrift bilden, werden als Bäume dargestellt. Dabei können bei der Veränderung, die beim genetischen Programmieren an den Lösungen vorgenommen wird, Symbole nur durch syntaktisch gleiche Symbole (im Sinne der obigen Grammatik) ersetzt werden.

Es können sowohl Terminal- als auch Nicht-Terminal-Symbole mutiert oder rekombiniert werden. Die Wahrscheinlichkeit, dass Terminal-Symbole von Änderungen betroffen sind, ist meist jedoch weit aus höher als dass Nicht-Terminal-Symbole geändert werden. In dem verwendeten Framework „ECJ“ ist standardmäßig eingestellt: Wahrscheinlichkeit (Terminal-Symbol) 0.9 und Wahrscheinlichkeit (Nicht-Terminal-Symbol) 0.1.

Im Sinne der obigen Grammatik sind nur Änderungen an syntaktisch gleichen Symbolen erlaubt. Änderungen können gleichermaßen Mutation oder Rekombination beim Kreuzen zweier Individuen sein. Das bedeutet:

<Move> → <Plus> | <Minus> | <LHalf> | <F> | <Variable>

<Plus>, <Minus>, <LHalf>, <F> und <Variable> können miteinander vertauscht werden. Diese Symbole stellen im Sinne der im Framework „ECJ“ verwendeten Parameter-Datei (siehe dazu auch Kapitel 8) schon die Terminal-Symbole dar, die dadurch eine größere Änderungswahrscheinlichkeit haben.

<Folge> → <FolgeGlied> <Folge> | <Empty>
<FolgeGlied> → <Move> | <Stack>

<Move>, <Stack> und <FolgeGlied> können miteinander vertauscht werden.
Spezialfall (siehe dazu auch Kapitel 8), <FolgeGlied> ist nur ein Hilfs-Symbol:
<Folge> und <Empty> können miteinander vertauscht werden.

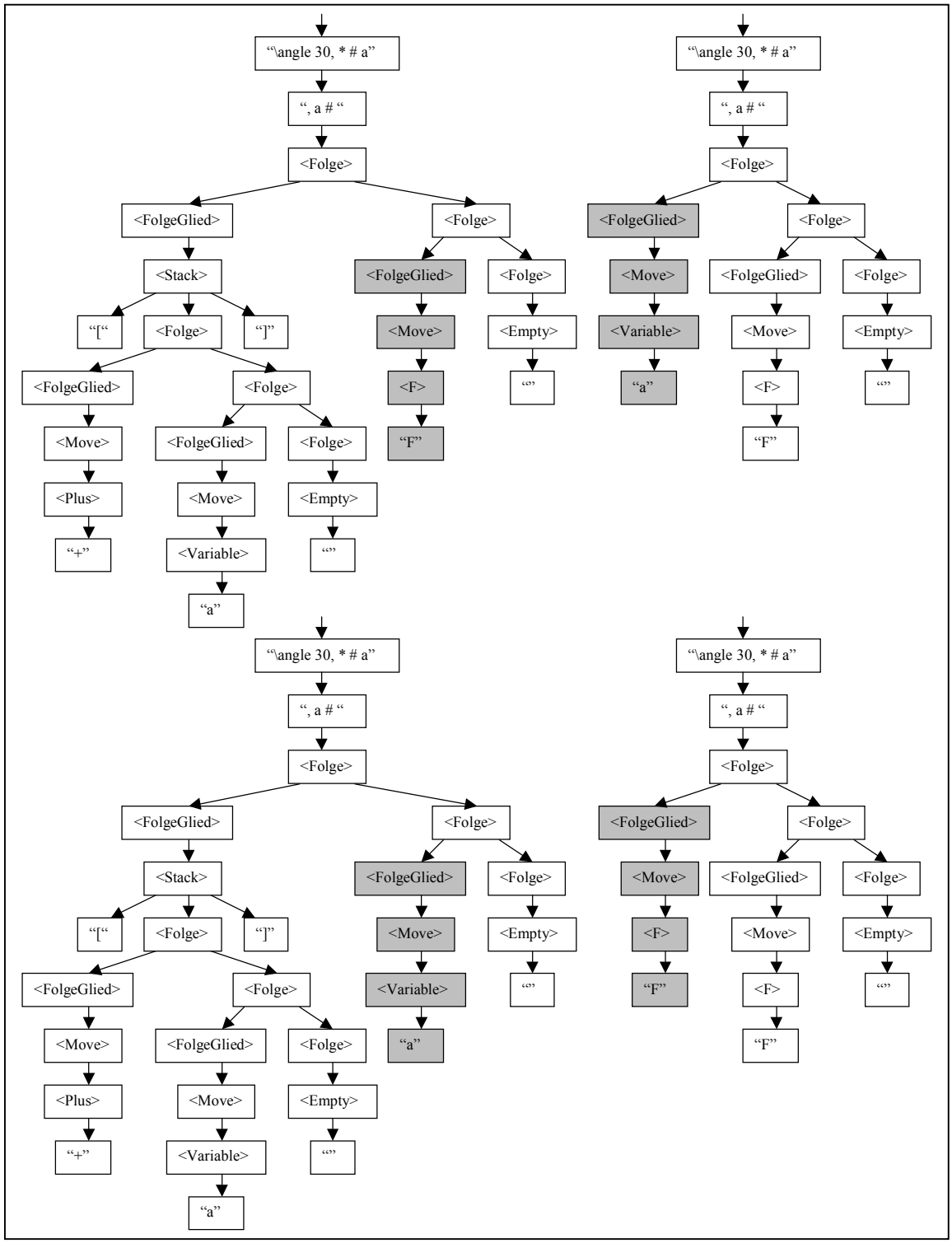


Abbildung 5: Kreuzung zweier L-System-Bäume

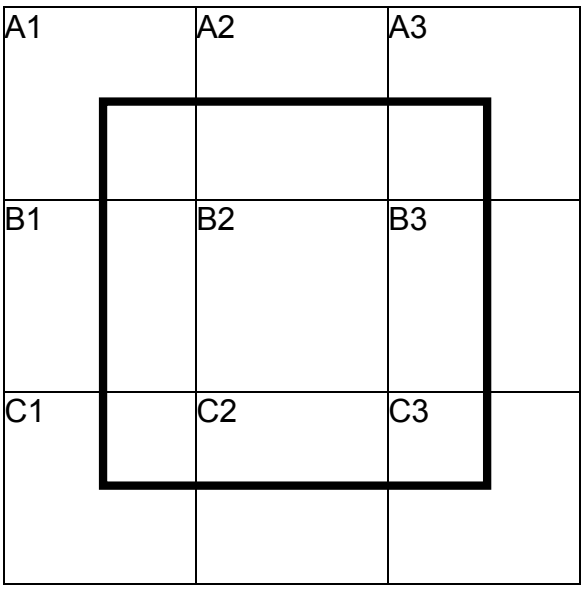
4. Statistische Analyse der L-Systeme

Um die L-Systeme miteinander vergleichen zu können und dadurch zu ermitteln, wie nah man der gewünschten Lösung ist, erfolgt eine Analyse der L-Systeme. Es ist das Ziel, eine vorgegebene Morphologie mittels eines L-Systems nachzubilden. Das heißt, man analysiert möglichst exakt die Struktur des vorgegebenen Systems und auf die gleiche Weise werden auch die Strukturen der erschaffenen L-Systeme analysiert. Dazu werden die erschaffenen L-Systeme interpretiert und die daraus entstandene Struktur wird analysiert.

In der Software „Evolve“ werden folgende Ausbildungen einer Struktur gemessen:

- die Anzahl der Äste,
- die Anzahl der endständigen Äste,
- die Länge der Äste,
- den tiefsten und den höchsten Punkt der Struktur,
- die Höhe und Breite der Struktur,
- das Verhältnis von Breite zu Höhe und
- die Verteilung der Astlängen in einem zweidimensionalen Gitter.

Der letzte Punkt ist wahrscheinlich der wichtigste für das exakt gleiche Aussehen einer Struktur. Alle anderen Punkte, die oben aufgeführt sind, lassen mehrere Strukturen zu, die genau dieselben Werte ausweisen. Aber die Astlängenverteilung in dem zweidimensionalen Gitter ist, je nach Auflösung, sehr unterschiedlich für verschiedenartige Strukturen.



Zelle	Astlängen-verteilung	Zelle	Astlängen-verteilung
A1	12,5%	B3	12,5%
A2	12,5%	C1	12,5%
A3	12,5%	C2	12,5%
B1	12,5%	C3	12,5%
B2	0,0%	Ge-samt	100,0%

Abbildung 6: Astlängenverteilung in einem 2D-Gitter

An Hand der analysierten Werte, die jeder Struktur und somit jedem erschaffen L-System zugeordnet werden, kann man nun die L-Systeme miteinander und mit der vorgegebenen Struktur vergleichen.

Die analysierten Werte werden bei der Berechnung der Fitness der L-Systeme durch die Fitnessfunktionen weiter verarbeitet.

5. Fitnessbewertung der L-Systeme

Die im Kapitel 4 vorgestellten Werte, die bei der Analyse der Strukturen der interpretierten L-Systeme entstehen, werden bei der Fitnessberechnung weiter verarbeitet. Mit Hilfe der Fitnessfunktionen werden jedem L-System Fitnesswerte zugeordnet, die im Wesentlichen einen Abstand zur vorgegebenen Struktur - also der Lösung - angeben. An Hand der Fitnesswerte werden L-Systeme ausgewählt, die die Grundlage für die nächste Generation von L-Systemen bilden.

Der Fitnesswert wird aus dem Unterschied zwischen dem aktuellen erschaffen L-System und der vorgegebenen Struktur ermittelt. In der Software „Evolve“ setzt sich der inverse Fitnesswert eines L-Systems aus 5 inversen Fitnessfunktionen zusammen, die verschiedene Unterschiede beider Strukturen messen:

- Unterschied der Astanzahl
- Unterschied der endständigen Astanzahl
- Unterschied der absoluten Dimension der Struktur
- Unterschied des Breite-Höhe-Verhältnisses der Struktur
- Unterschied der Astlängenverteilung in einem 2D-Gitter

Beispiel für die Berechnung des Unterschieds der Astanzahl:

```
double shoots; // Astanzahl des erschaffenen L-Systems
double target; // Astanzahl der vorgegebenen Struktur

double difference= Math.abs(shoots - target);
// The maximum difference is limited to target.
// So that systems with a value of 0 don't be better
// than systems with a value larger than 2*target.
difference= Math.min(difference, target);
double shootDifference= difference / target; // [0.0;1.0]
if( Double.isNaN(shootDifference) ||
    Double.isInfinite(shootDifference) ){
  shootDifference= 1.0;
}
```

Beispiel für die Berechnung des Unterschieds der Astlängenverteilung:

Im Kapitel 4 wurde am Beispiel eines 3 x 3 – Gitters gezeigt, wie die Astlänge in den einzelnen Gitterzellen bestimmt wird. Im Folgenden wird beschrieben, wie daraus der Unterschied zwischen zwei Strukturen berechnet wird, um dadurch einen inversen Fitnesswert abzuleiten.

Für jede Zelle der beiden zu vergleichenden Strukturen wird das Verhältnis `expectedRatio` bzw. `testRatio` zwischen der Astlänge in dieser Zelle `ExpectedAnalyse.cellSum[i][j]` bzw. `Analyse.cellSum[i][j]` und der gesamten Astlänge der Struktur `ExpectedAnalyse.lengthSum` bzw. `Analyse.lengthSum` berechnet.

Aus den Astlängenverhältnissen `expectedRatio` und `testRatio` der korrespondierenden Zellen beider Strukturen wird die Differenz `cellSumDifference` gebildet.

Die Differenzen `cellSumDifference` werden über alle Zellen des Gitters (z.B. 9 = 3 x 3) aufsummiert `sum`.

Am Ende der Berechnung wird die Summe `sum` normiert, so dass ein Wert zwischen 0.0 und 1.0 entsteht.

```
private static double measureCellDifference(){
  double sum= 0.0;
  int layers= (int) Parameters.getParameterValue(Parameters.layers);
  for( int i= 0; i < layers; i++ ){
    for( int j= 0; j < layers; j++ ){
      double expectedRatio=
        ExpectedAnalyse.cellSum[i][j] / ExpectedAnalyse.lengthSum;
      double testRatio= Analyse.cellSum[i][j] / Analyse.lengthSum;
      double cellSumDifference= Math.abs(expectedRatio - testRatio);
      if( Double.isNaN(cellSumDifference) ||
          Double.isInfinite(cellSumDifference) ){
        return 1.0;
      }else{
        sum+= cellSumDifference;
      }
    }
  }
  // the sum of all cell ratios of one structure is 1
  // so the sum of the differences of the single cells can maximum be 2
  // and to normalize the return value, divide it by 2
  return sum / 2.0; // [0.0;1.0]
}
```

Die Werte der 5 einzelnen Funktionen können jeweils zwischen 0.0 (für kein Unterschied) und 1.0 (für großer Unterschied) liegen. Die einzelnen Funktionen können unterschiedlich gewichtet werden. Eine genauere Untersuchung der Auswirkungen dieser Gewichtung ist im Kapitel 9 zu finden. Die Werte der einzelnen inversen Fitnessfunktionen werden summiert und ergeben somit einen Wert zwischen 0.0 und 5.0, wobei ein niedriger Wert für geringe Unterschiede steht und ein hoher für große Unterschiede.

Wenn ein erschaffenes L-System trivial ist, das heißt es gibt keine Regel (außer der Startregel), die in eine Variable abgeleitet wird, wird dieses L-System mit einem inversen Fitnesswert von 6.0 versehen. Falls ein L-System nicht korrekt gebildet wurde und somit syntaktische Fehler beinhaltet, wird dieses L-System mit einem inversen Fitnesswert von 7.0 bestraft.

Dieses Verhalten ist jedoch sehr erwünscht, da L-Systeme mit hohen inversen Fitnesswerten höchstwahrscheinlich nicht in die nächste Generation übernommen werden. Somit werden sehr weit von der Lösung entfernte, triviale und syntaktisch falsche L-Systeme schnell aus dem Suchraum ausgeschlossen.

6. Aufbau und Funktionsweise der Software „Evolve“

Die Software „Evolve“ ist verantwortlich für die Steuerung der Evolution der L-Systeme. Sie steuert die Erzeugung, die Analyse, die Bewertung und die Auswahl der L-Systeme für die nächste Generation. Außerdem besitzt „Evolve“ eine grafische Benutzeroberfläche, die für die Eingaben und Ausgaben eines Evolutionlaufes verantwortlich ist. Optional ist es auch möglich, ohne Interaktion mit dem Benutzer per Batchmodus die Evolution zu steuern. Des Weiteren ist „Evolve“ das Bindeglied zu „GroIMP“ auf der einen und „ECJ“ auf der anderen Seite.

Alle Dateien, die direkt zu „Evolve“ gehören, sind im „GroIMP“-Verzeichnis der CD unter ./evolve/ zu finden. Die Klassen, die von „Evolve“ benötigt werden, bilden das Package `evolve`. Ein weiteres Package, das die Tokens für die Bildung eines Baumes gemäß Kapitel 3 enthält, heißt `evolve.tokens`.

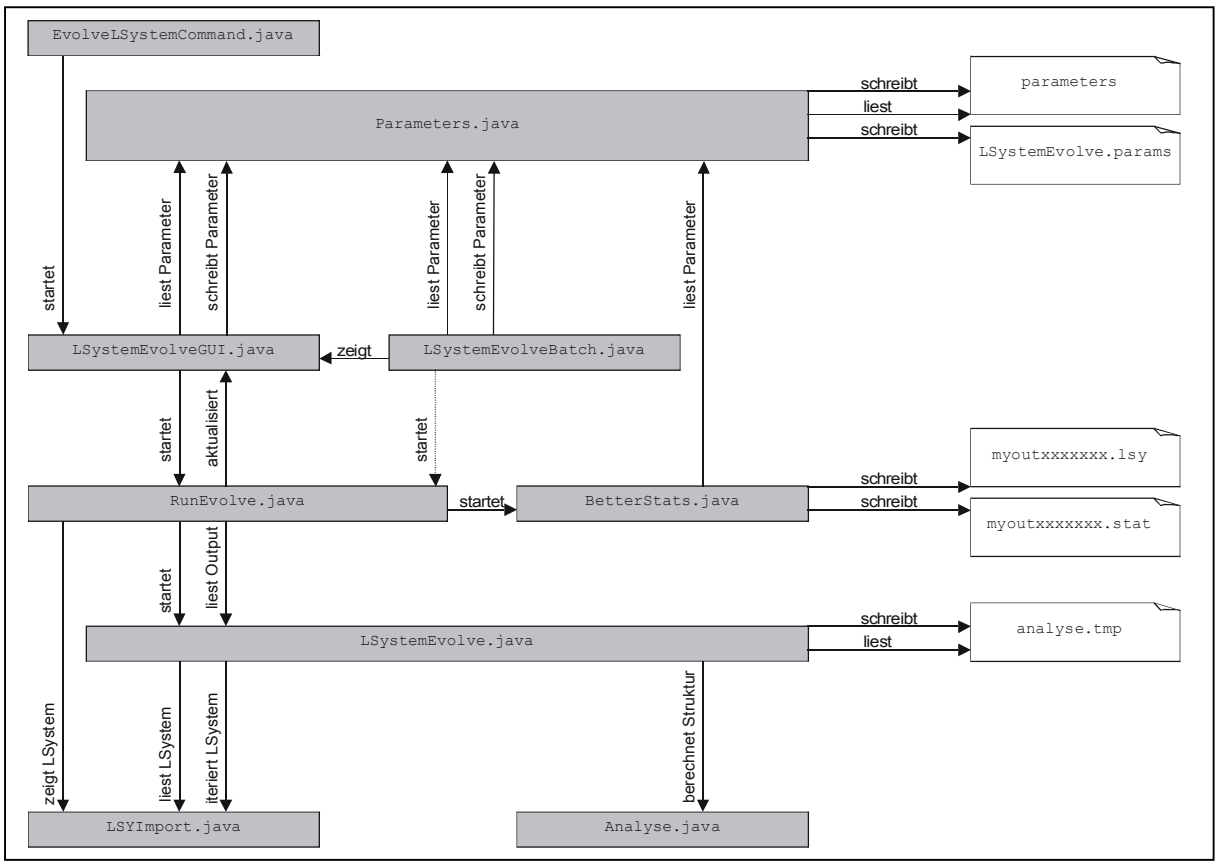


Abbildung 7: Diagramm der Funktionsweise der Software „Evolve“ auf Klassenbasis

Wenn man eine Struktur in der Software „GroIMP“ betrachtet, kann man unter dem Menüpunkt „Evolve“ den Eintrag „Run“ aktivieren. Dadurch wird ein `EvolveLSystemCommand` erzeugt. Dieses Kommando startet die `LSystemEvolveGUI` und übergibt ihr den Wurzelknoten der betrachteten Struktur. In der GUI kann der Benutzer verschiedene Parameter, die den Lauf der Evolution beeinflussen, verändern und damit die Evolution beschleunigen oder verlangsamen. Dadurch kann man auch gut Parameterkombinationen finden, die für eine jeweilige Struktur günstig sind. Einige Tests und deren Ergebnisse zu den Parameterkombinationen sind in den Kapiteln 9 und 10 nachzulesen.

Wenn der Benutzer in der GUI den „Start“-Knopf betätigt, werden die aktuellen Parameter gespeichert, die aktuelle Struktur analysiert und der Evolutionslauf gestartet.

Die aktuellen Parameter werden in einer nicht temporären Datei `parameters` gespeichert. Beim nächsten Aufruf der GUI werden die gespeicherten Parameter als Anfangswerte vorgeschlagen. Dieses Verhalten ist günstig, wenn man von Lauf zu Lauf jeweils nur einen Parameter ändern möchte. Außerdem wird durch die Klasse `Parameters` auch noch eine Datei namens `LSystemEvolve.params` gespeichert, die Parameter für die Anwendung „ECJ“ enthält.

Die aktuelle Struktur wird an Hand des übergebenen Wurzelknotens analysiert und die berechneten Werte werden in der temporären Datei `analyse.tmp` gespeichert.

Der Evolutionslauf wird durch den Aufruf von `RunEvolve` gestartet.

Die GUI dient während eines Laufes zur Ausgabe des aktuellen Evolutionsfortschrittes und zeigt die zur Zeit beste Lösung an. Man kann über den „Stop“-Knopf den Evolutionslauf vorzeitig beenden.

Wenn man die Evolution nicht interaktiv über die Software „GroIMP“ starten möchte, sondern lieber im Batchbetrieb, um mehrere Läufe unbeaufsichtigt zu absolvieren, muss man `LSystemEvolveBatch` starten. Diese Klasse speichert analog zur `LSystemEvolveGUI` ebenfalls die verwendeten Parameter, die Werte der analysierten Struktur und startet den Evolutionslauf. Im Falle des Batchbetriebs wird jedoch eine Struktur übergeben, die das Ziel des Evolutionslaufs darstellt. Dies kann eine LSY-Datei sein, in der ein L-System definiert ist, oder eine DTD-Datei, in der eine Struktur beschrieben ist.

Durch den Aufruf `java LsystemEvolveBatch help` kann man eine Übersicht über die Parameter, die man übergeben kann, erlangen:

```
LSystemEvolveBatch (lsystem file | dtd file) [parameter value]

lsystem file - file with a Lsystem
dtd file - file with a DTDSystem
parameter value - value for the corresponding parameter

parameters and their current values are:
numberOfRules 1.0
elite 0.0
xover 0.9
ratioWeight 1.0
cellWeight 1.0
iterations 4.0
angle 30.0
mutate 0.5
terminalShootWeight 1.0
population 1000.0
dimensionWeight 1.0
layers 3.0
shootWeight 1.0
seed 4357.0
reprod 0.1
generations 50.0

Example:
LSystemEvolveBatch lsystem beispiel.lsy seed 4392 iterations 4
```

Wie erwähnt, wird der Evolutionslauf durch Aufruf von `RunEvolve` gestartet. `RunEvolve` startet die Anwendung „ECJ“ mit dem Problem `LSystemEvolve` in einem extra Prozess.

`LSystemEvolve` ist verantwortlich für die Analyse und Bewertung der erschaffenen L-Systeme. Diese Klasse wird von der Anwendung „ECJ“ für jedes Individuum einer Generation aufgerufen. Die Analyse und Bewertung der L-Systeme durch die Klasse `LSystemEvolve` wird folgendermaßen vollzogen.

Ein L-System (eigentlich ein Individuum, das wie in Kapitel 3 baumartig aufgebaut ist) wird an die Klasse `LSystemEvolve` übergeben. Dort wird das L-System aus dem Individuum durch Traversierung des Baumes erzeugt. Danach wird das L-System durch die Klasse `LSYImport` iteriert und geometrisch interpretiert. Die geometrische Struktur des L-Systems wird durch die Klasse `Analyse` analysiert. In dieser Klasse werden Werte berechnet, die Aussagen über die morphologische Struktur, wie in Kapitel 4 beschrieben, geben sollen. An Hand der von `Analyse` berechneten Werte wird in `LSystemEvolve` durch die Fitnessfunktionen, wie in Kapitel 5 beschrieben, dem L-System ein Fitnesswert zugeordnet. Danach übernimmt die Anwendung „ECJ“ wieder die Kontrolle und führt die Selektion der Individuen für die nächste Generation und deren Reproduktion an Hand der übergebenen Parameter aus. Als Selektionsmethode dient „Tournament“-Selektion mit 7 Individuen, zur Reproduktion dienen eine Mutations-, eine Crossover- und eine Reproduktionspipeline, wie in Abbildung 8 dargestellt. Die Wahrscheinlichkeiten für die einzelnen Pipelines werden durch die eingegebenen Parameter bestimmt. Die neu erschaffenen Individuen werden dann wieder von `LSystemEvolve` analysiert und bewertet.

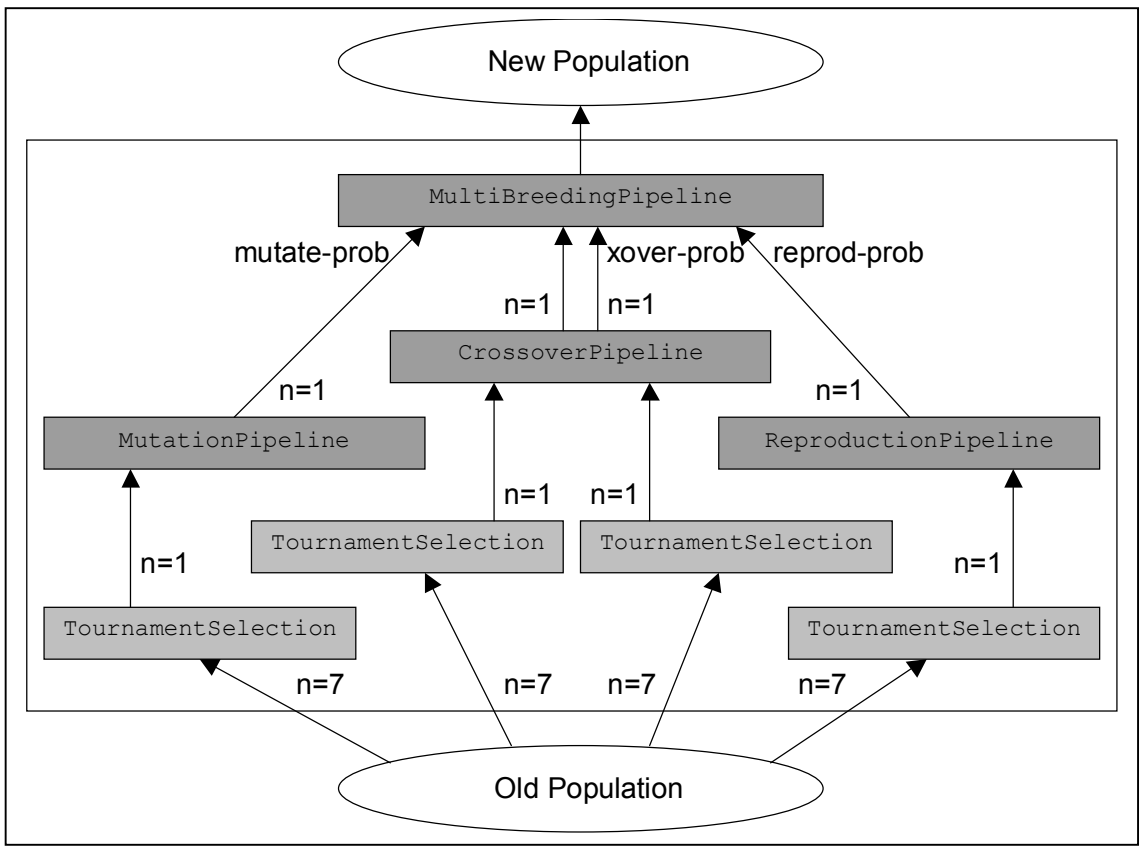


Abbildung 8: Diagramm der Funktionsweise der Selektion und Reproduktion

Der Output, der von `LSystemEvolve` erzeugt wird, wird von `RunEvolve` analysiert und an die GUI weitergegeben. Am Ende eines jeden Laufes wird von „ECJ“ automatisch eine Protokolldatei `out.stat` erstellt. Diese Datei wird von der Klasse `BetterStats`, die durch `RunEvolve` am Ende eines Laufes aufgerufen wird, in ein leichter verständliches Format gebracht und unter `myoutxxxxxxx.stat` gespeichert. Außerdem werden die Parameter, die dem Lauf zu Grunde lagen, zusätzlich in die Datei `myoutxxxxxxx.stat` übernommen. Des Weiteren wird auch das beste L-System durch `RunEvolve` an `BetterStats` übergeben. `BetterStats` speichert diesen String in einer separaten Datei `myoutxxxxxxx.lsy` im LSY-Format. Das `xxxxxxx` in dem Dateinamen steht für die aktuelle Rechnerzeit, damit mehrere `myout`-Dateien existieren können, was wichtig für den Batchbetrieb ist.

Eine ausführliche Dokumentation aller Klassen, Methoden und Attribute der Software „Evolve“ ist in der Java-Dokumentation im „GroIMP“-Verzeichnis der CD unter ./evolve/doc/index.html nachzulesen.

7. Anbindung der Software an „GroIMP“

Wenn in „GroIMP“ eine Struktur dargestellt wird, existiert in der Menüzeile ein Eintrag namens „Evolve“. Unter diesem Menü befindet sich die Aktion „Run“, wodurch die Software „Evolve“ mit ihrer GUI gestartet wird.

Um die Software „Evolve“ in „GroIMP“ einzubinden, mussten nur wenige Klassen und Dateien von „GroIMP“ verändert beziehungsweise angepasst werden. Diese Anpassungen werden im Folgenden etwas genauer erläutert.

Die Klasse `EvolveLSystemCommand.java` ist komplett neu und startet die GUI von „Evolve“. Die Klasse erbt jedoch von `yam.ui.Command` und ist deshalb im Package `yam.ui.commands` von „GroIMP“ zu finden.

Die Klasse `Global.java` im Package `yam.basis` wurde um zwei Items erweitert, die für das oben genannte Menü verantwortlich sind.

```
public static final Item
  /** Menu directory for evolution. */
  EVOLVE = ACTION_POOL.createDirectory (null, ”evolve”),
  /** Menu item for evolution. */
  EVOLVE_RUN = ACTION_POOL.createItem (null, ”evolve/run”,
                                       new EvolveLSystemCommand()),
```

Die Datei `Resources.properties` im „GroIMP“-Verzeichnis ./yam/ wurde ebenfalls um zwei Einträge erweitert, die für die textuelle Repräsentation der oben genannten Menüpunkte verantwortlich sind.

```
action-evolve=Evolve L-System
action-evolve/run=Run
```

Und schließlich wurde noch die Klasse `LSYImport.java` im Package `rgg` um 3 Methoden und deren dazugehörige Attribute erweitert.

Die Methode `readLSystem()` ist dazu da, Zeichenketten als L-Systeme einzulesen. Die Methode `iterateLSystem()` iteriert das vorher eingelesene L-System ein Mal. Und die letzte hinzugefügte Methode `showLSystem()` zeigt ein L-System, das geometrisch interpretiert wird, in einem Editorfenster.

8. Anbindung der Software an „ECJ“

„Evolve“ führt, wie in Kapitel 6 erwähnt, die Anwendung „ECJ“ mit dem Problem `LSystemEvolve` in einem extra Prozess aus.

Alle Dateien, die zur Anwendung „ECJ“ gehören, befinden sich im „GroIMP“-Verzeichnis der CD unter ./ec/. Eine ausführliche Beschreibung der Software inklusive Tutorials ist unter ./ec/index.html und ./ec/docs/index.html zu finden. „ECJ“ wurde anfangs in der Version 10 und wird aktuell in der Version 11 benutzt.

Damit die Anwendung „ECJ“ die richtige Aufgabe lösen kann, muss ein Problem definiert werden. Die Klasse, die das Problem enthält, ist `LSystemEvolve.java` aus dem Package `evolve`. In dieser Klasse werden die einzelnen Individuen analysiert und mit einem Fitnesswert versehen, wie in den Kapiteln 4 und 5 beschrieben.

Des Weiteren sind zwei Dateien mit Parametern für „ECJ“ im Verzeichnis ./evolve/ vorhanden. Zum einen `LSystemEvolveBase.params`, welche Parameter für das Problem und einige wenige Parameter für den Evolutionslauf enthält, und zum anderen `LSystemEvolve.params`, welche Informationen über die Bildung eines Individuums, gemäß der Grammatik, die in Kapitel 3 beschrieben wurde, enthält. Dazu enthält `LSystemEvolve.params` die Parameter, die in „Evolve“ eingestellt wurden. Da die Parameter bei jedem Evolutionslauf geändert werden können, wird diese Parameterdatei auch vor jedem Lauf von der Klasse `Parameters.java` neu geschrieben.

Weitere Klassen, die von „ECJ“ gebraucht werden, befinden sich im Package `evolve.tokens`. Diese Klassen enthalten die Tokens, die zum Aufbau eines Individuums benötigt werden. Die Datei `LSystemEvolve.params` verweist in vielen Parametern auf die Klassen des Package `evolve.tokens`. Die Tokens korrespondieren mit den Non-Terminal-Symbolen der Grammatik, die in Kapitel 3 beschrieben wurde.

Der Zusammenhang zwischen der Grammatik aus Kapitel 3, den Tokens des Package `evolve.tokens`, der Datei `LSystemEvolve.params` und der Klasse `Parameters.java` wird im folgenden kurz dargestellt.

Ein Auszug aus der Grammatik lautet:

N = { Folge, FolgeGlied, Move, Stack, Plus, Minus, LHalf, F, Variable, Empty }
T = { [,], +, -, L*0.5, F, a }
S = { <Folge> }
P = { <Folge> → <FolgeGlied> <Folge> | <Empty>
<FolgeGlied> → <Move> | <Stack>
<Move> → <Plus> | <Minus> | <LHalf> | <F> | <Variable>
<Stack> → “[“ <Folge> “]”
<Plus> → “+”
<Minus> → “-”
<LHalf> → “L*0.5”
<F> → “F”
<Variable> → “a”
<Empty> → “” }

Im Package `evolve.tokens` sind auch die Klassen `Folge.java`, `Stack.java`, `Plus.java`, `Minus.java`, `LHalf.java`, `F.java`, `Variable.java`, `Empty_Literal.java` vorhanden, die die entsprechenden Tokens beinhalten.

Der dazugehörige Teil der Parameterdatei lautet:

```
gp.fs.0.func.0 = evolve.tokens.Empty_Literal
gp.fs.0.func.0.nc = EmptyNC
```
bedeutet: Funktion 0 ist <Empty>

```
gp.nc.7 = ec.gp.GPNodeConstraints
gp.nc.7.name = EmptyNC
gp.nc.7.returns = Empty
gp.nc.7.size = 0
```
bedeutet: Bedingung 7 ist <Empty> hat 0 Kinder

```
gp.type.s.0.name = Empty
gp.type.s.0.size = 1
gp.type.s.0.member.0 = Folge
```
bedeutet: <Folge> → <Empty>

```
gp.fs.0.func.1 = evolve.tokens.Plus
gp.fs.0.func.1.nc = MoveNC
gp.fs.0.func.2 = evolve.tokens.Minus
gp.fs.0.func.2.nc = MoveNC
gp.fs.0.func.3 = evolve.tokens.LHalf
gp.fs.0.func.3.nc = MoveNC
gp.fs.0.func.4 = evolve.tokens.F
gp.fs.0.func.4.nc = MoveNC
gp.fs.0.func.5 = evolve.tokens.Variable
gp.fs.0.func.5.nc = MoveNC
```
bedeutet: <Move> → <Plus> | <Minus> | <LHalf> | <F> | <Variable>

```
gp.nc.8 = ec.gp.GPNodeConstraints
gp.nc.8.name = MoveNC
gp.nc.8.returns = Move
gp.nc.8.size = 0

gp.type.s.1.name = Move
gp.type.s.1.size = 1
gp.type.s.1.member.0 = FolgeGlied
```
bedeutet: <FolgeGlied> → <Move>

```
gp.fs.0.func.6 = evolve.tokens.Folge
gp.fs.0.func.6.nc = FolgeNC

gp.nc.9 = ec.gp.GPNodeConstraints
gp.nc.9.name = FolgeNC
gp.nc.9.returns = Folge
gp.nc.9.size = 2
gp.nc.9.child.0 = FolgeGlied
gp.nc.9.child.1 = Folge
```
bedeutet: <Folge> → <FolgeGlied> <Folge>
```
gp.type.a.1.name = FolgeGlied
gp.type.a.2.name = Folge
```

```
gp.fs.0.func.7 = evolve.tokens.Stack
gp.fs.0.func.7.nc = StackNC

gp.nc.10 = ec.gp.GPNodeConstraints
gp.nc.10.name = StackNC
gp.nc.10.returns = Stack
gp.nc.10.size = 1
gp.nc.10.child.0 = Folge
```
bedeutet: <Stack> → <Folge>

```
gp.type.s.2.name = Stack
gp.type.s.2.size = 1
gp.type.s.2.member.0 = FolgeGlied
```
bedeutet: <FolgeGlied> → <Stack>

```
gp.tc.0.returns = Folge
```
bedeutet: <Folge> ist Startsymbol

```
gp.fs.0.size = 8
gp.nc.size = 11
gp.type.s.size = 3
gp.type.a.size = 3
```

Die Klasse `Parameters.java` schreibt bei jedem Evolutionslauf die obige Parameterdatei neu. Wenn man also die Grammatik erweitern möchte, muss man die entsprechenden Token-Klassen hinzufügen und die Klasse `Parameters.java` entsprechend erweitern.

Innerhalb der Software „Evolve“ wird die Anwendung „ECJ“ mit dem Problem `LSystemEvolve` gestartet. Dies wird in der Klasse `RunEvolve` erledigt, die außerdem die Kommunikation zwischen „ECJ“ und „Evolve“ sichert.

`RunEvolve` ist als Thread implementiert, damit die Software nicht während eines Evolutionslaufes blockiert wird. Die Anwendung „ECJ“ wird in einem extra Prozess gestartet, weil es sonst nicht möglich war, mehr als einen Evolutionslauf ohne Beendigung von „GroIMP“ nacheinander zu starten. Die Kommunikation zwischen „ECJ“ und „Evolve“ wird über Streams bewerkstelligt. Dazu werden in den Output, den `LSystemEvolve` erzeugt, bestimmte Zeichenketten eingebaut. Diese Zeichen werden von `RunEvolve` gesucht und dann mit bestimmter Bedeutung in der GUI dargestellt. So werden Informationen über die aktuelle Generation, die aktuelle Fitness und den aktuellen L-System-String hervorgehoben.

9. Tests und Ergebnisse mit den Fitnessfunktionen

Wie schon in Kapitel 5 erwähnt, wurden verschiedene Tests zur Gewichtung der 5 Fitnessfunktionen gemacht, um deren Einfluss auf die Evolutionsergebnisse zu bestimmen.

Es wurde immer versucht, die folgenden 2 Strukturen, die nach 4 Iterationen entstehen, nachzubilden (vgl. Abbildung 9):

Struktur 1 - \angle 30,* # a(200),a(x) # F(x/2) [+ a(x/2)] [- a(x/2)]

Struktur 2 - \angle 30,* # a,a # F[-F][F][+F]a

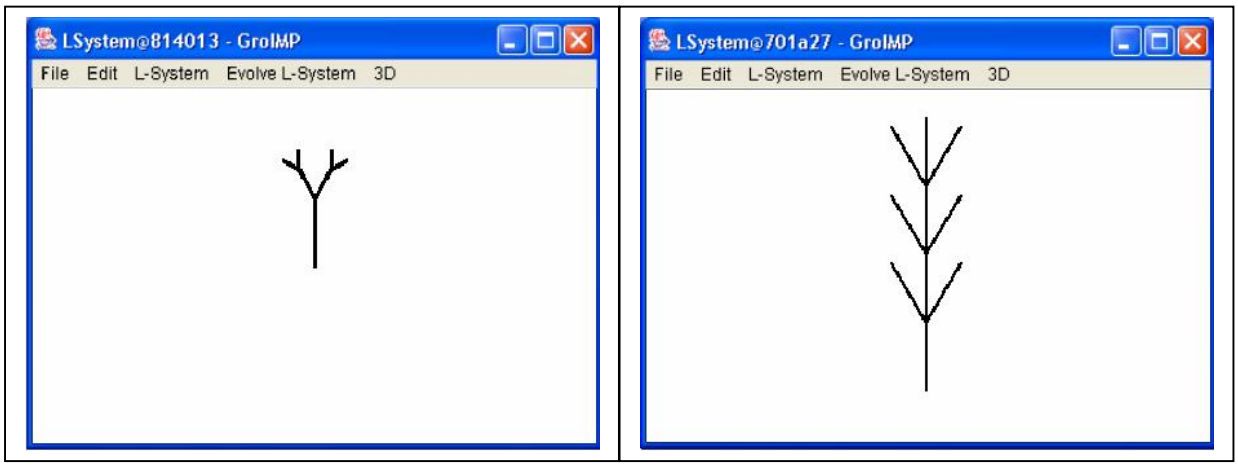


Abbildung 9: Struktur 1 (links) und Struktur 2 (rechts) nach jeweils 4 Iterationen

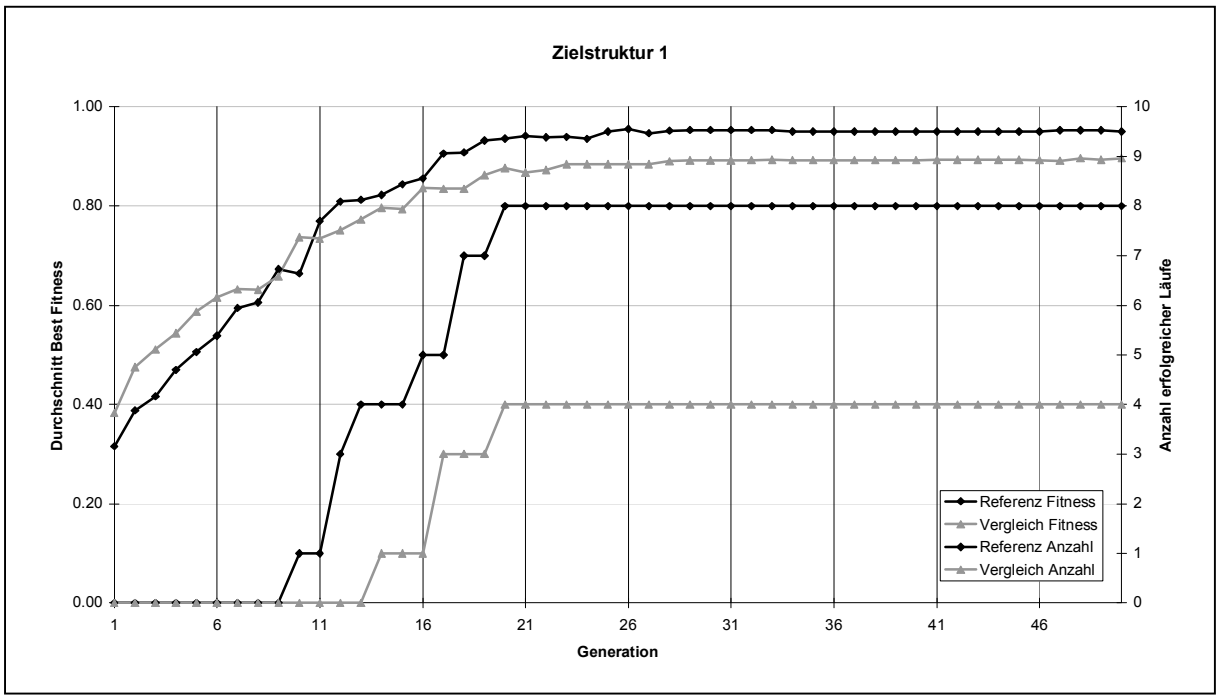
Zu jeder Struktur wurden 10 Testläufe mit unterschiedlichen Zufallszahlengeneratoren gemacht. Das wurde mit dem Parameter `seed` bewerkstelligt, der jeweils von 4357 bis 4366 lief.

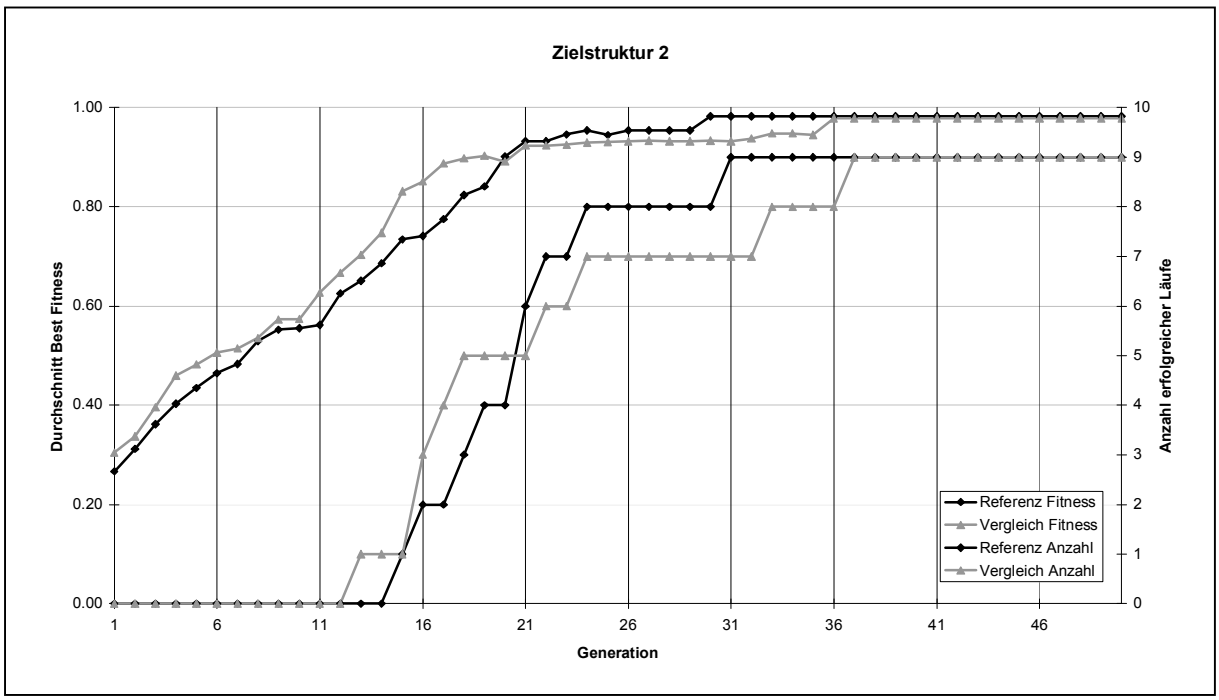
Die Läufe waren auf 50 Generationen begrenzt. Zu jeder Generation wurde der beste Fitnesswert notiert. Es wurde jedoch nicht direkt der Fitnesswert von „Evolve“ genommen, sondern ein von „ECJ“ normalisierter Fitnesswert, der in der Datei `out.stat` zu finden ist. Bei diesem Fitnesswert bedeutet 0 die schlechteste und 1 die beste Fitness. Wenn ein Lauf vor dem Erreichen der 50. Generation zur Lösung kam, wurde für die folgenden Generationen der beste Fitnesswert auf 1 gesetzt. Aus den besten Fitnesswerten der 10 Läufe zu jeder Struktur wurde generationsweise der Durchschnitt gebildet.

$$bestFitnessAverage = \frac{\sum_{seed=4357}^{4366} bestFitness_{seed}}{10}$$ für die aktuelle Generation

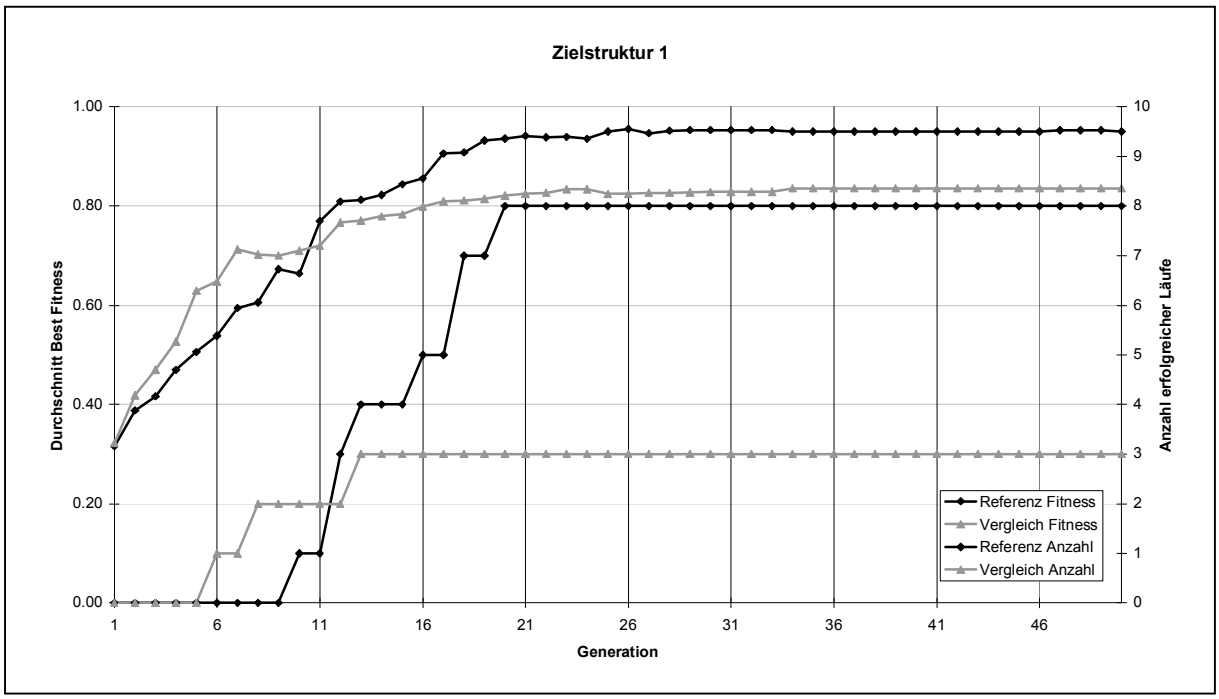
In den folgenden Diagrammen wurde der durchschnittlich beste Fitnesswert und die Anzahl der erfolgreichen Läufe zu jeder Generation eingetragen. Es gibt jeweils eine Referenzkonfiguration – Referenz –, bei der alle 5 Fitnessfunktionen gleichgewichtet sind, in jedem Diagramm, um besser mit der getesteten Konfiguration – Vergleich – vergleichen zu können.

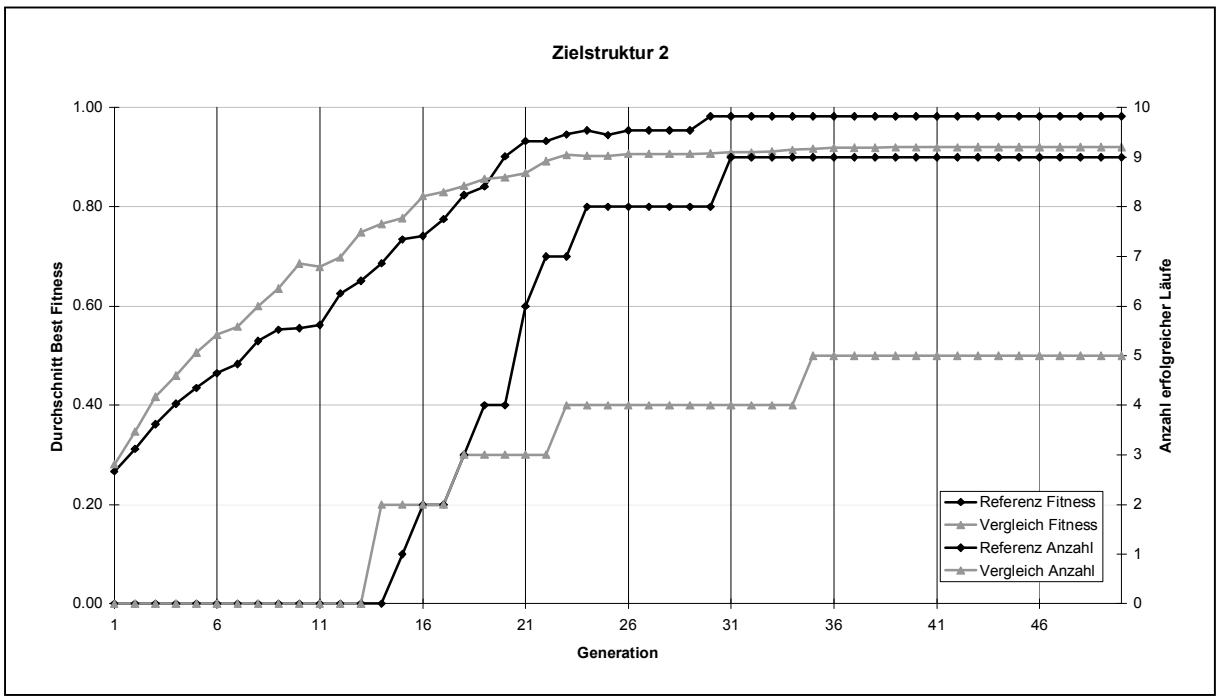
Referenz: CellWeight 1, RatioWeight 1, DimensionWeight 1, ShootWeight 1, TerminalWeight 1
Vergleich: CellWeight 1, RatioWeight 1, DimensionWeight 1, ShootWeight 4, TerminalWeight 1



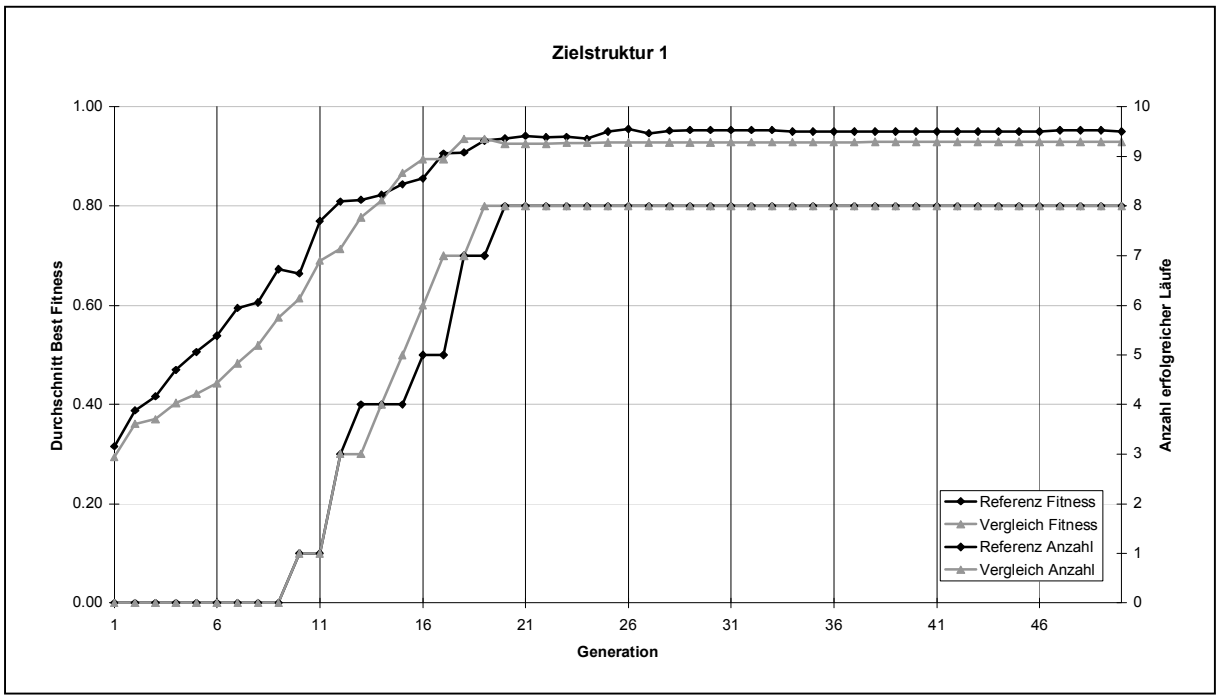


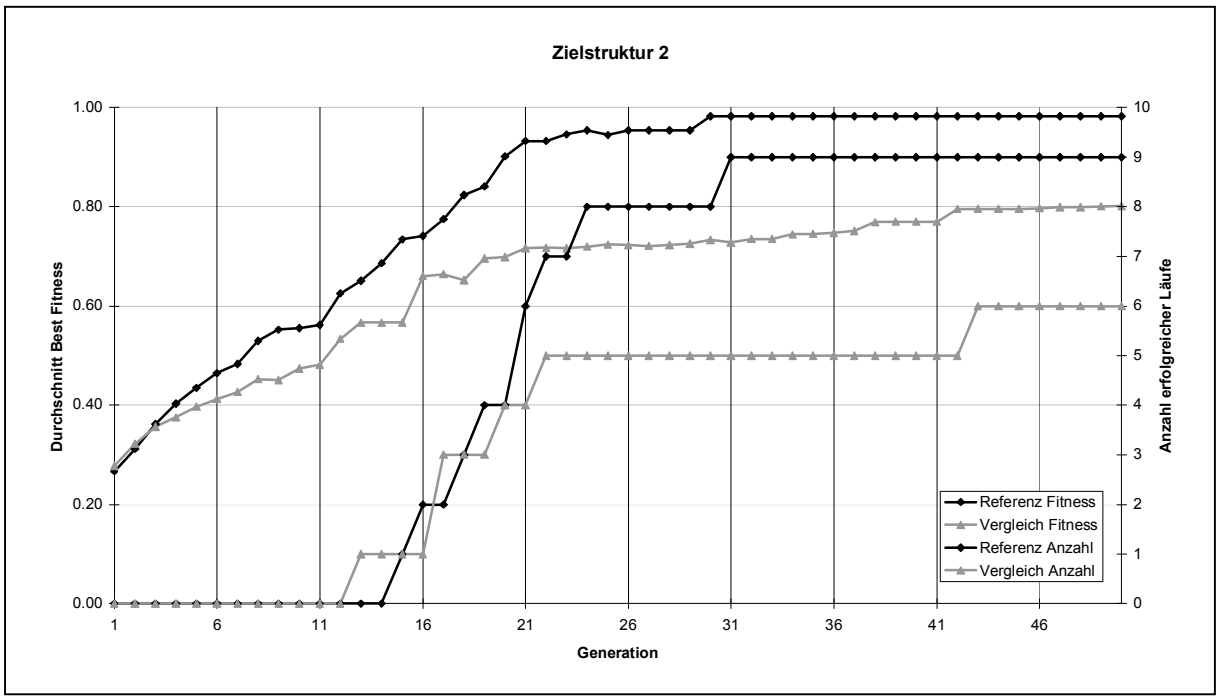
Referenz: CellWeight 1, RatioWeight 1, DimensionWeight 1, ShootWeight 1, TerminalWeight 1
Vergleich: CellWeight 1, RatioWeight 1, DimensionWeight 4, ShootWeight 1, TerminalWeight 1



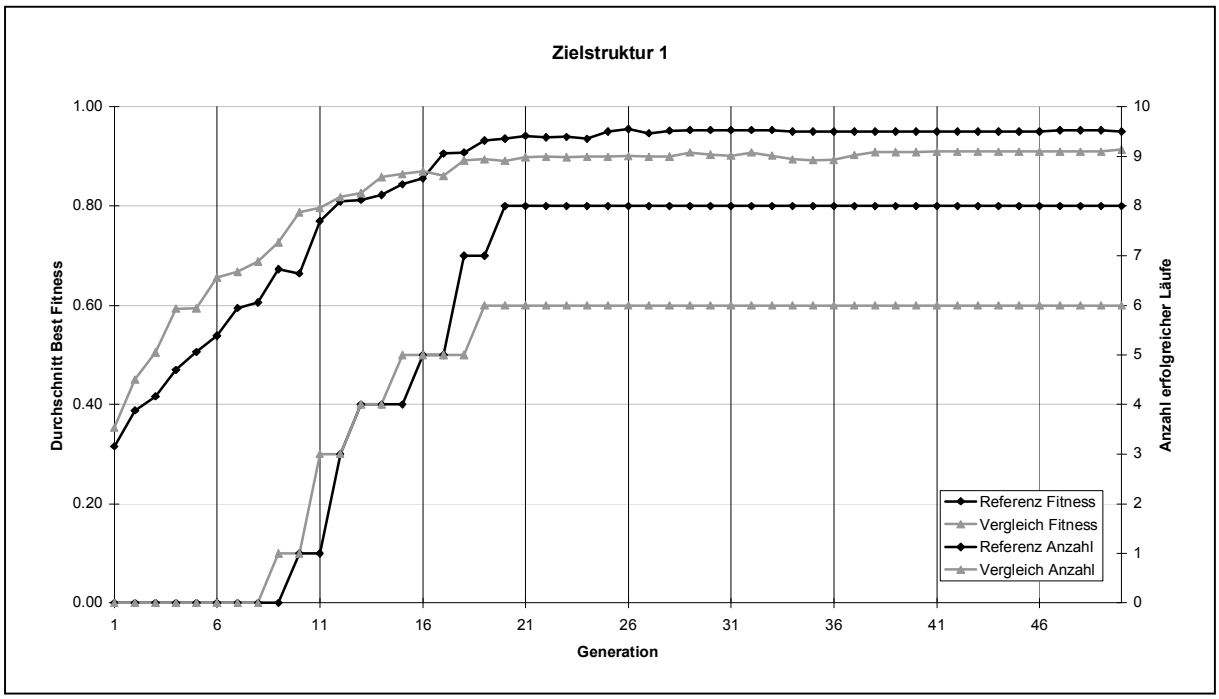


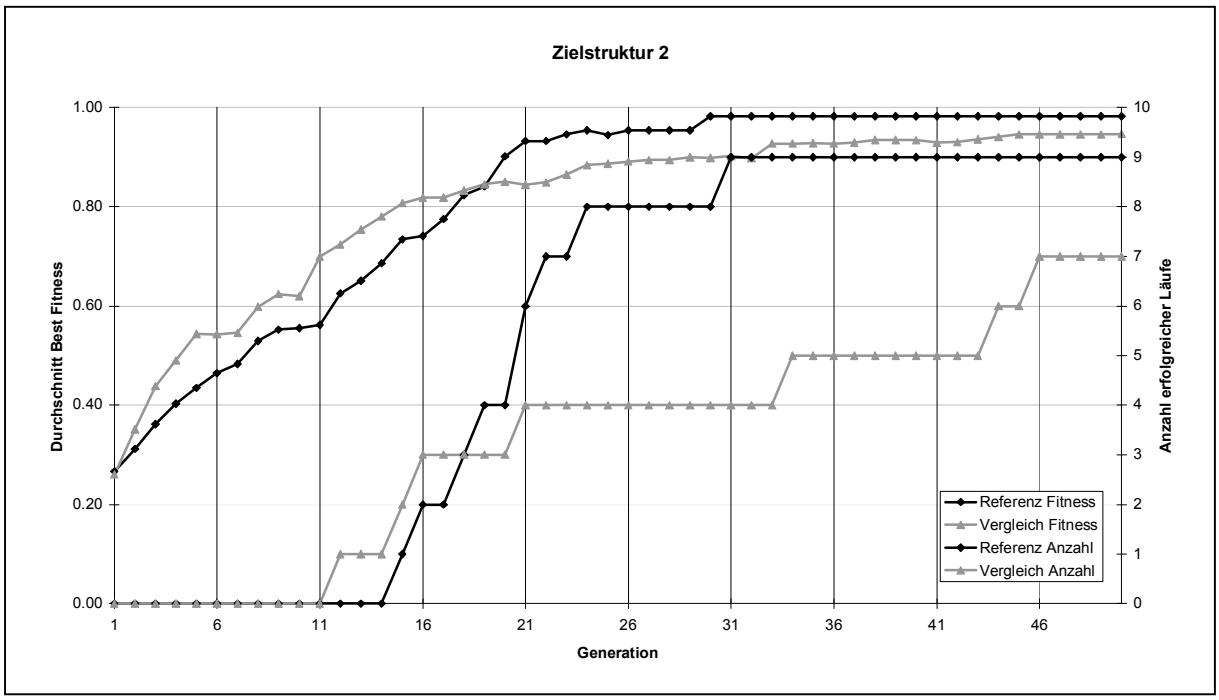
Referenz: CellWeight 1, RatioWeight 1, DimensionWeight 1, ShootWeight 1, TerminalWeight 1
Vergleich: CellWeight 4, RatioWeight 1, DimensionWeight 1, ShootWeight 1, TerminalWeight 1



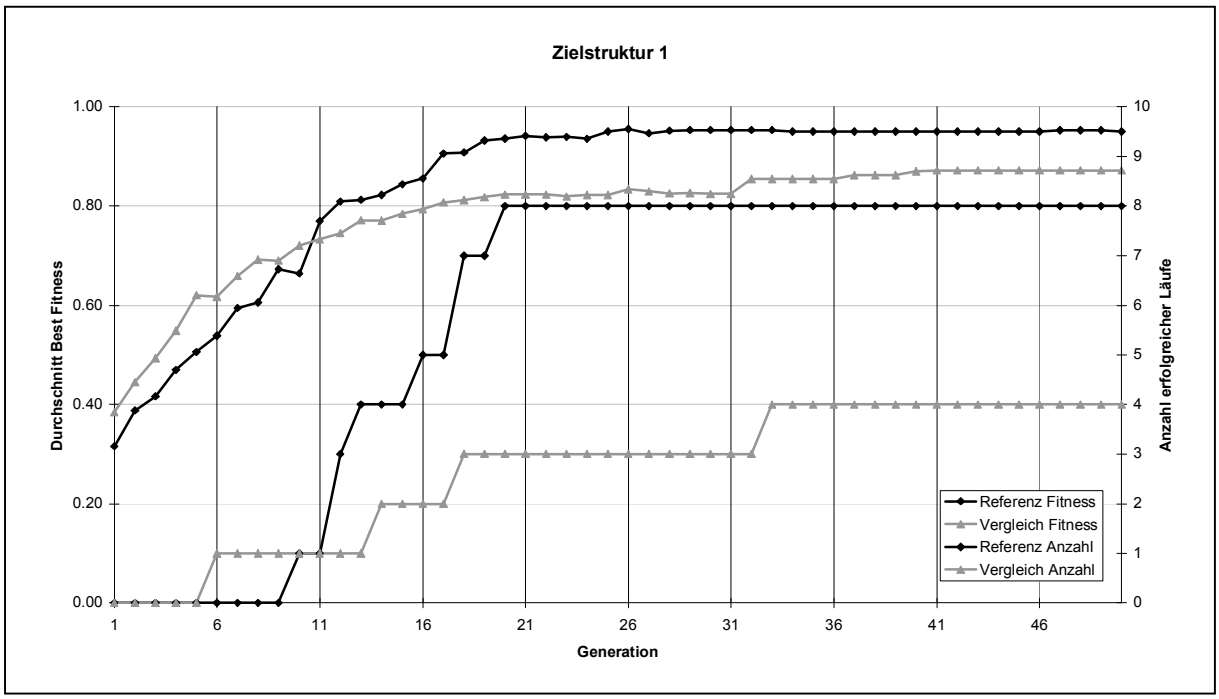


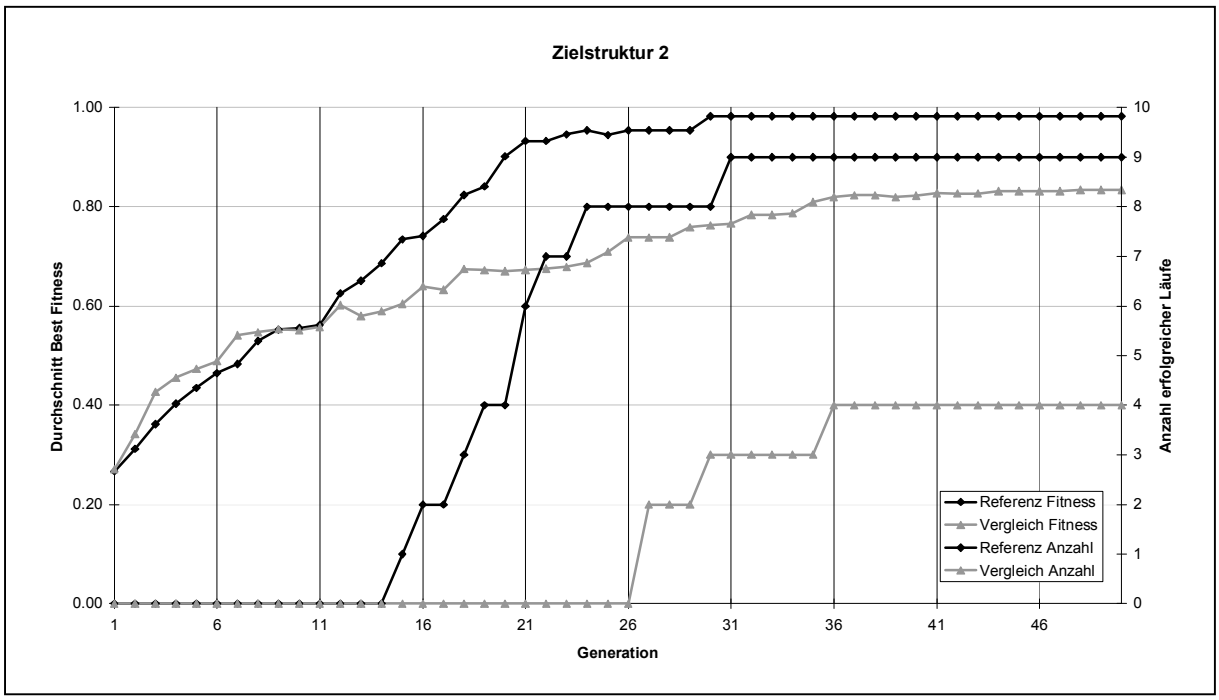
Referenz: CellWeight 1, RatioWeight 1, DimensionWeight 1, ShootWeight 1, TerminalWeight 1
Vergleich: CellWeight 1, RatioWeight 1, DimensionWeight 1, ShootWeight 1, TerminalWeight 4



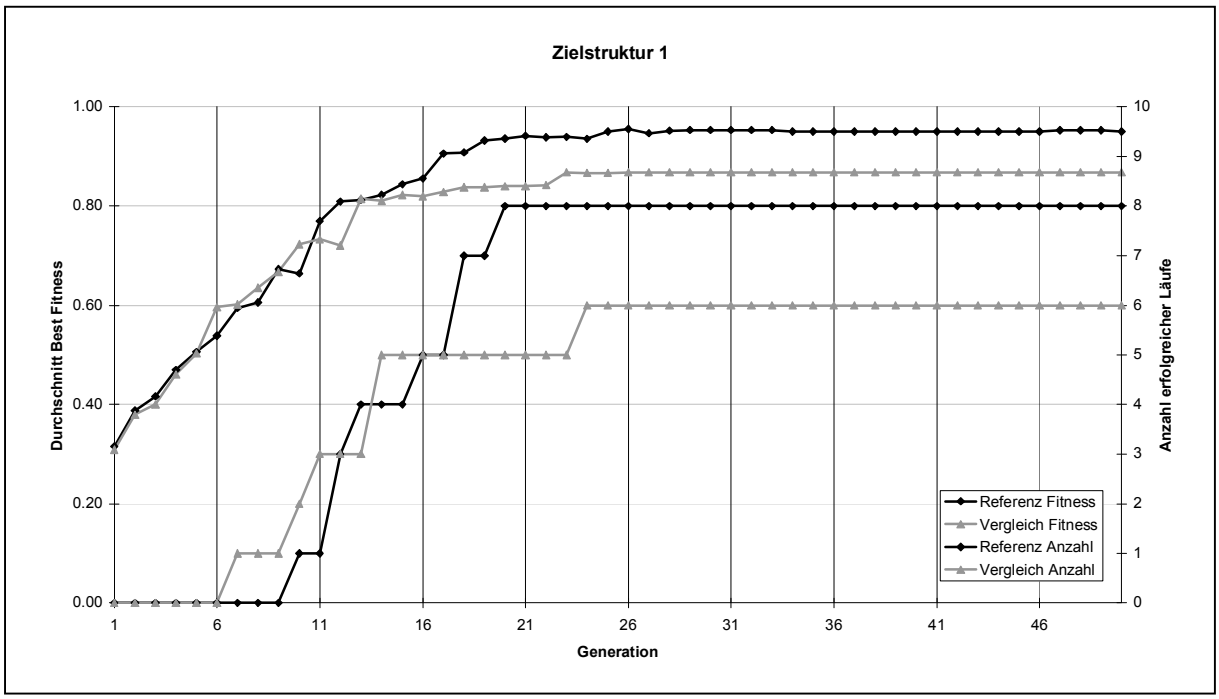


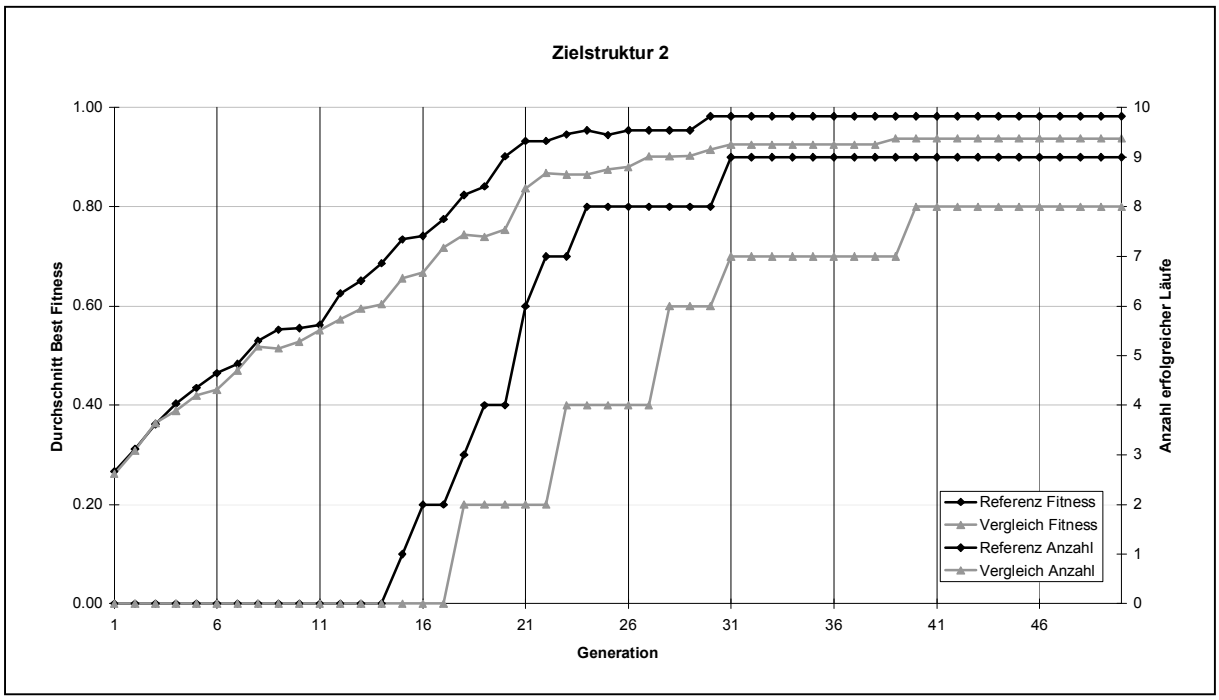
Referenz: CellWeight 1, RatioWeight 1, DimensionWeight 1, ShootWeight 1, TerminalWeight 1
Vergleich: CellWeight 1, RatioWeight 4, DimensionWeight 1, ShootWeight 1, TerminalWeight 1



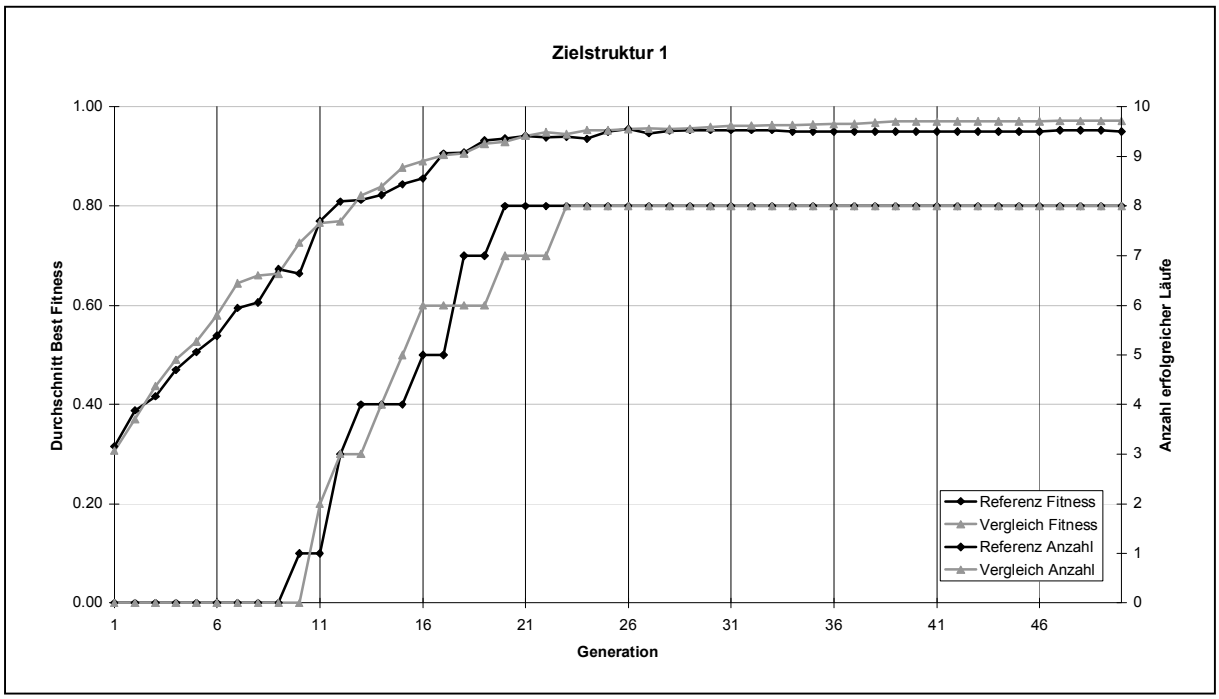


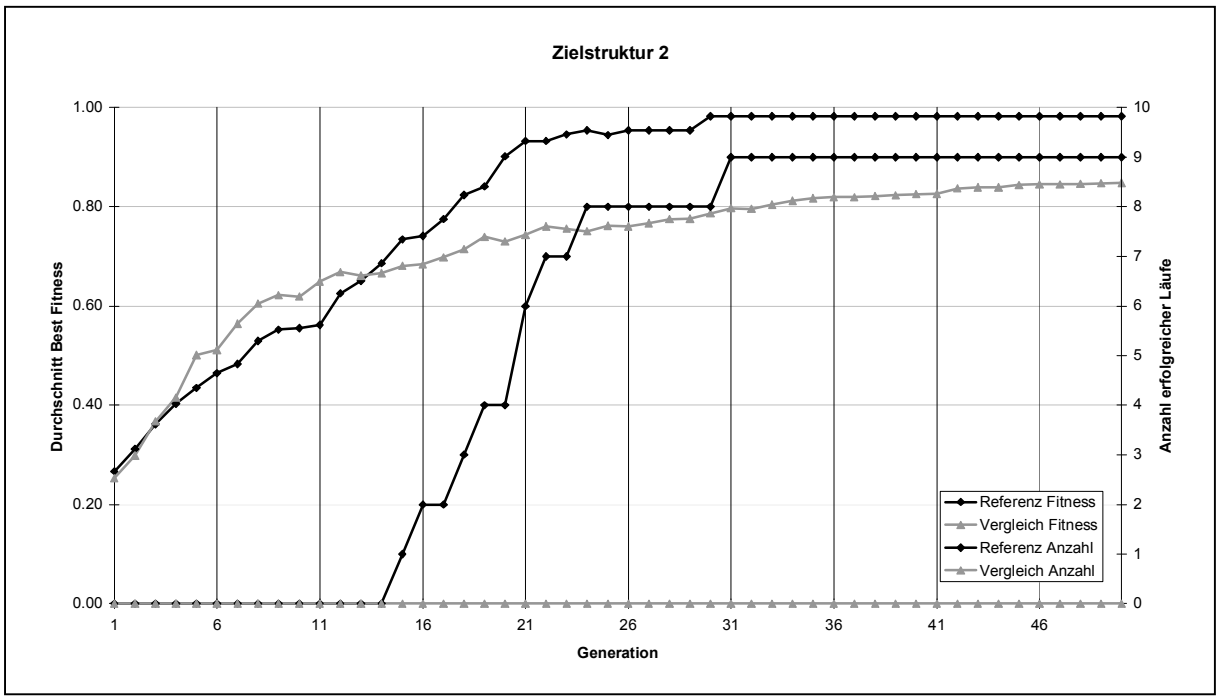
Referenz: CellWeight 1, RatioWeight 1, DimensionWeight 1, ShootWeight 1, TerminalWeight 1
Vergleich: CellWeight 2, RatioWeight 1, DimensionWeight 1, ShootWeight 1, TerminalWeight 2



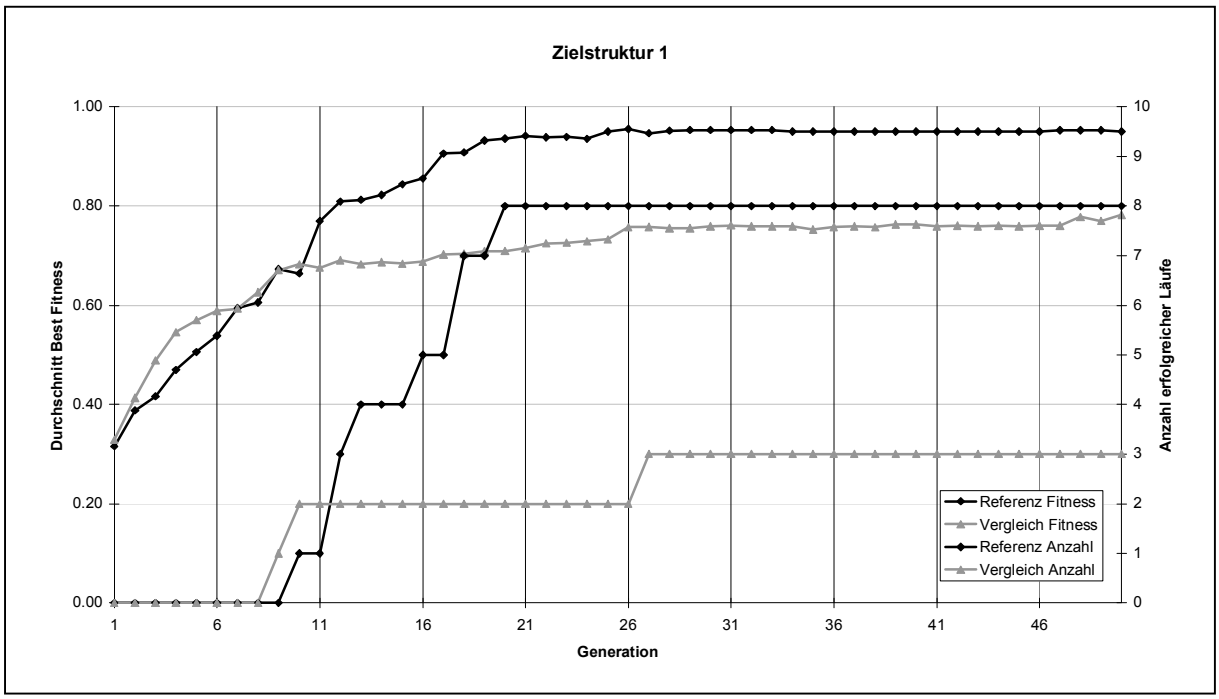


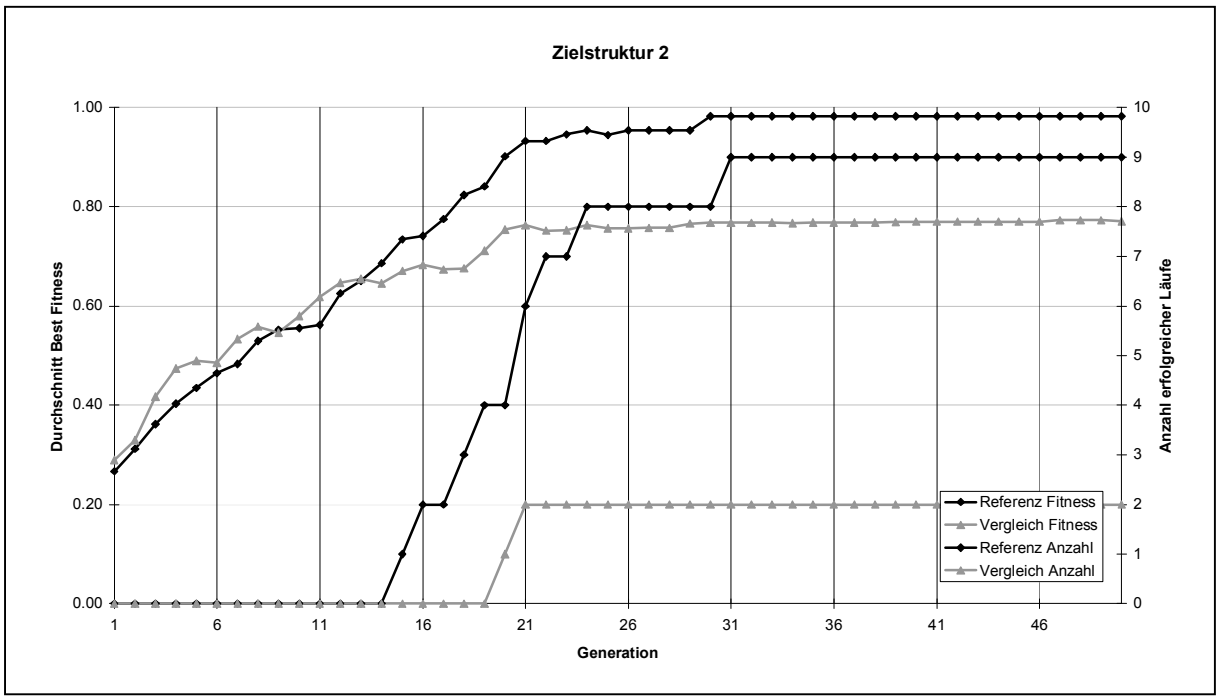
Referenz: CellWeight 1, RatioWeight 1, DimensionWeight 1, ShootWeight 1, TerminalWeight 1
Vergleich: CellWeight 1, RatioWeight 1, DimensionWeight 1, ShootWeight 0, TerminalWeight 1



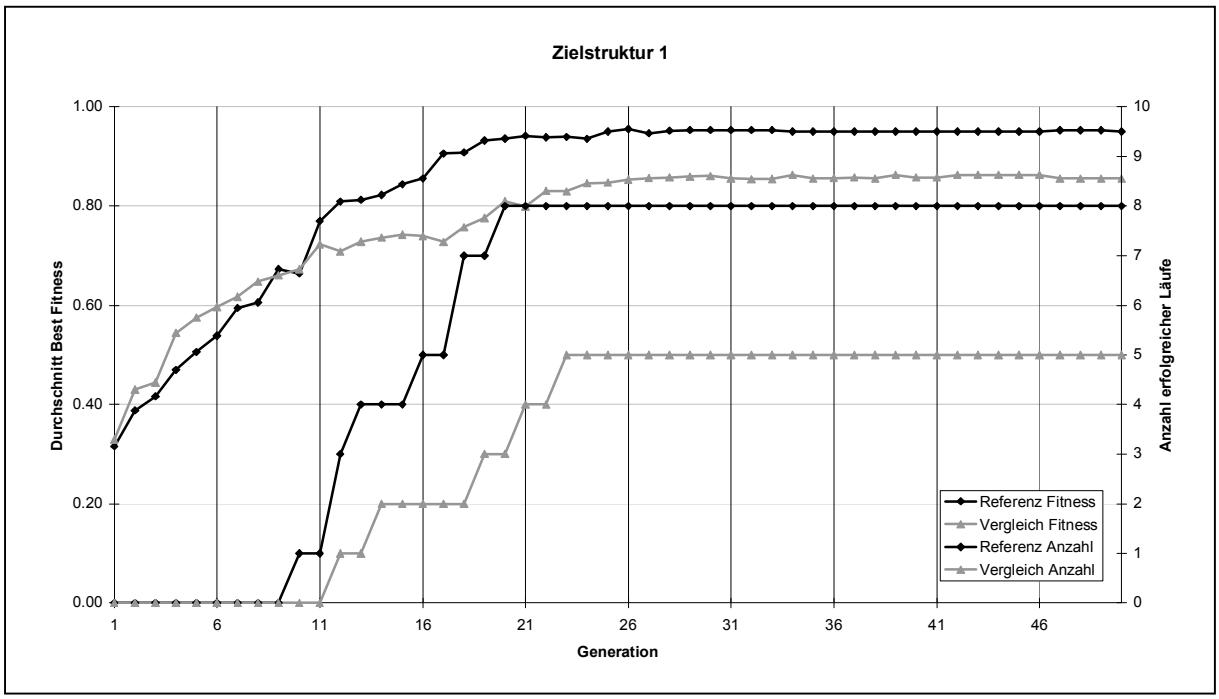


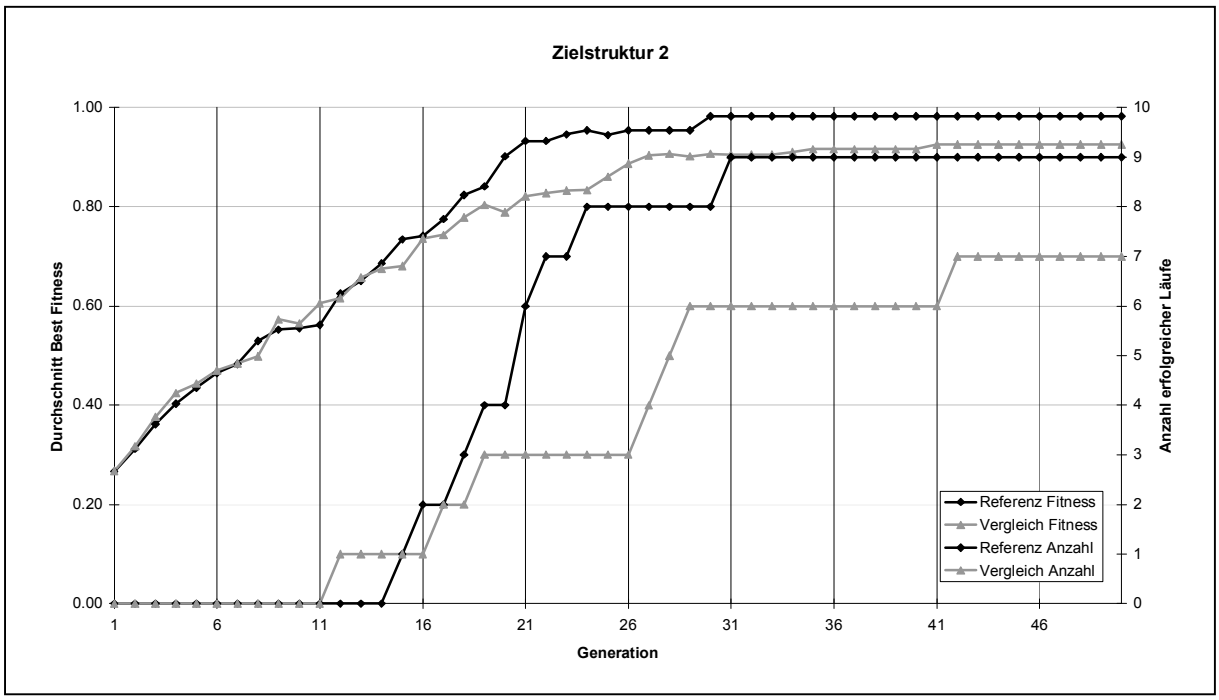
Referenz: CellWeight 1, RatioWeight 1, DimensionWeight 1, ShootWeight 1, TerminalWeight 1
Vergleich: CellWeight 1, RatioWeight 1, DimensionWeight 1, ShootWeight 1, TerminalWeight 0





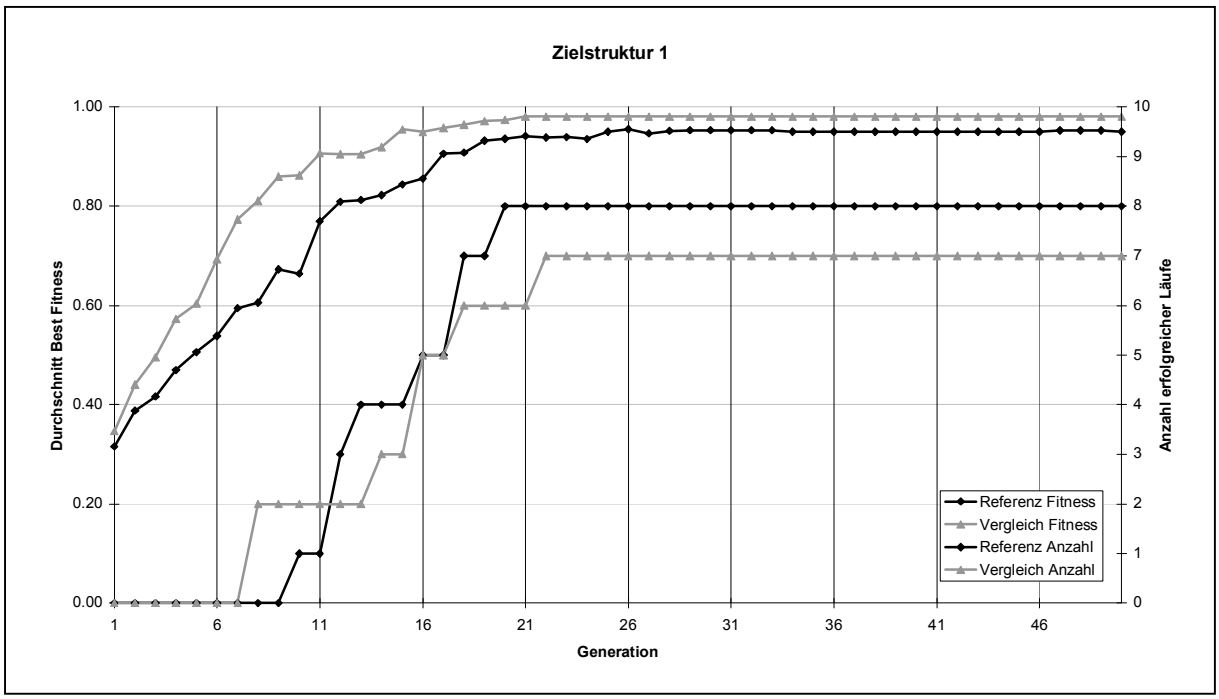
Referenz: CellWeight 1, RatioWeight 1, DimensionWeight 1, ShootWeight 1, TerminalWeight 1
Vergleich: CellWeight 1, RatioWeight 1, DimensionWeight 0, ShootWeight 1, TerminalWeight 1

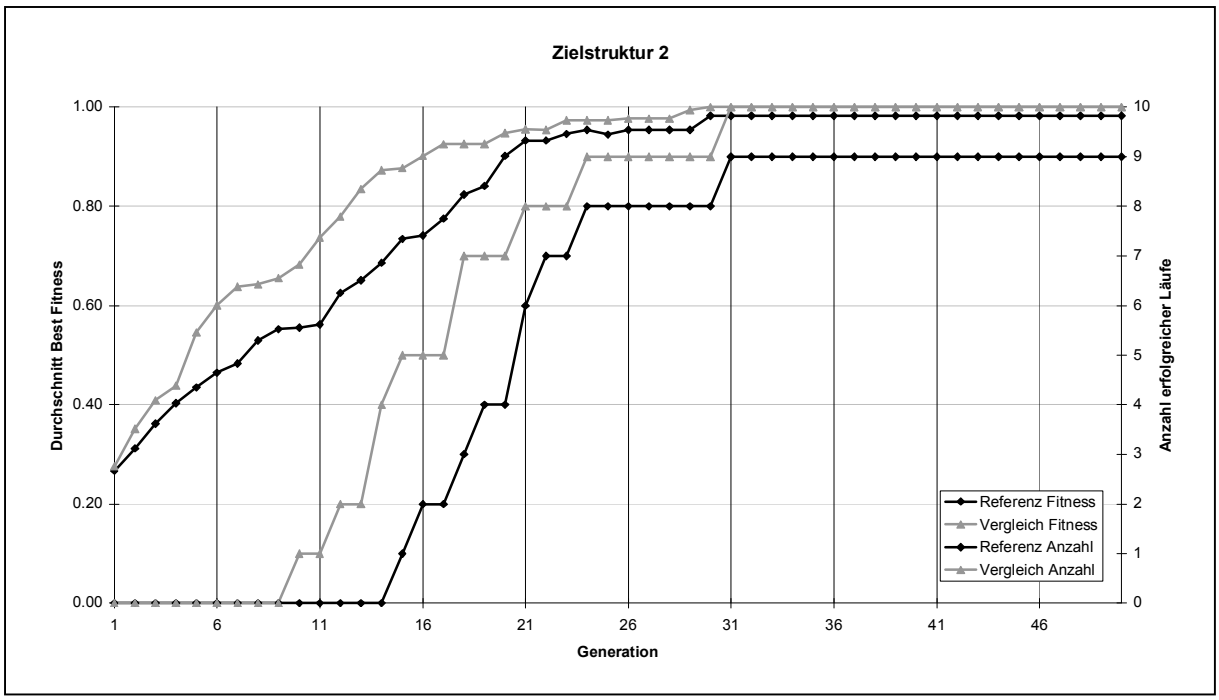




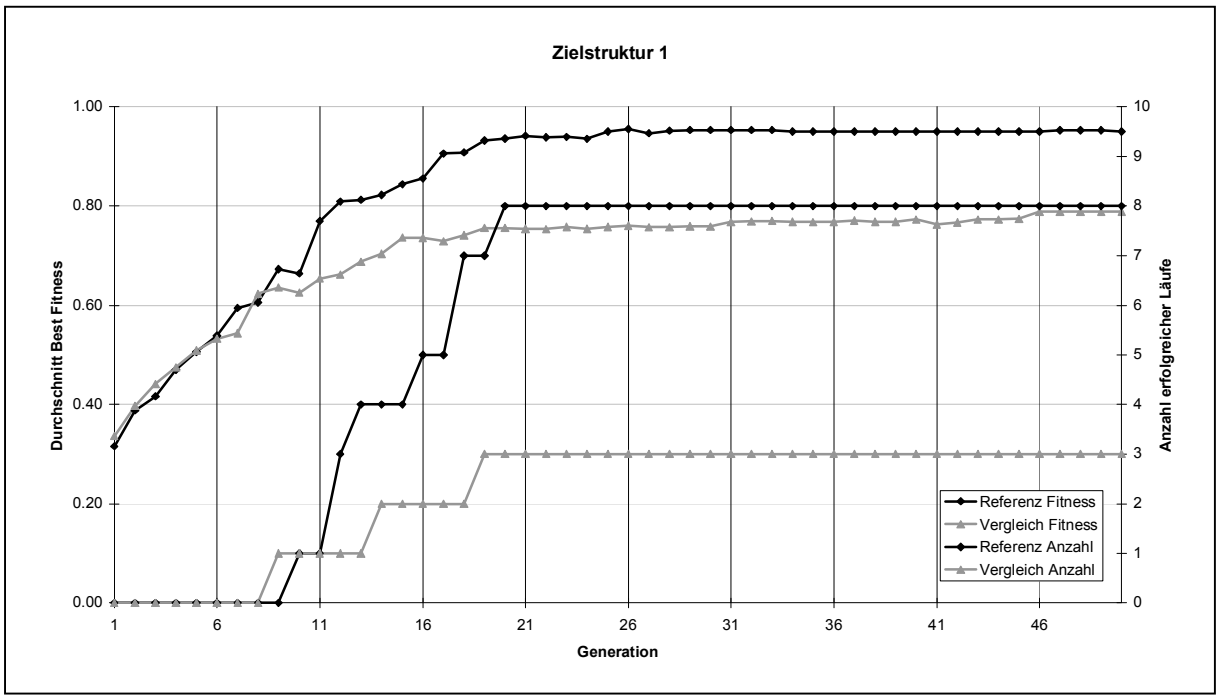
Referenz: CellWeight 1, RatioWeight 1, DimensionWeight 1, ShootWeight 1, TerminalWeight 1
Vergleich: CellWeight 0, RatioWeight 1, DimensionWeight 1, ShootWeight 1, TerminalWeight 1

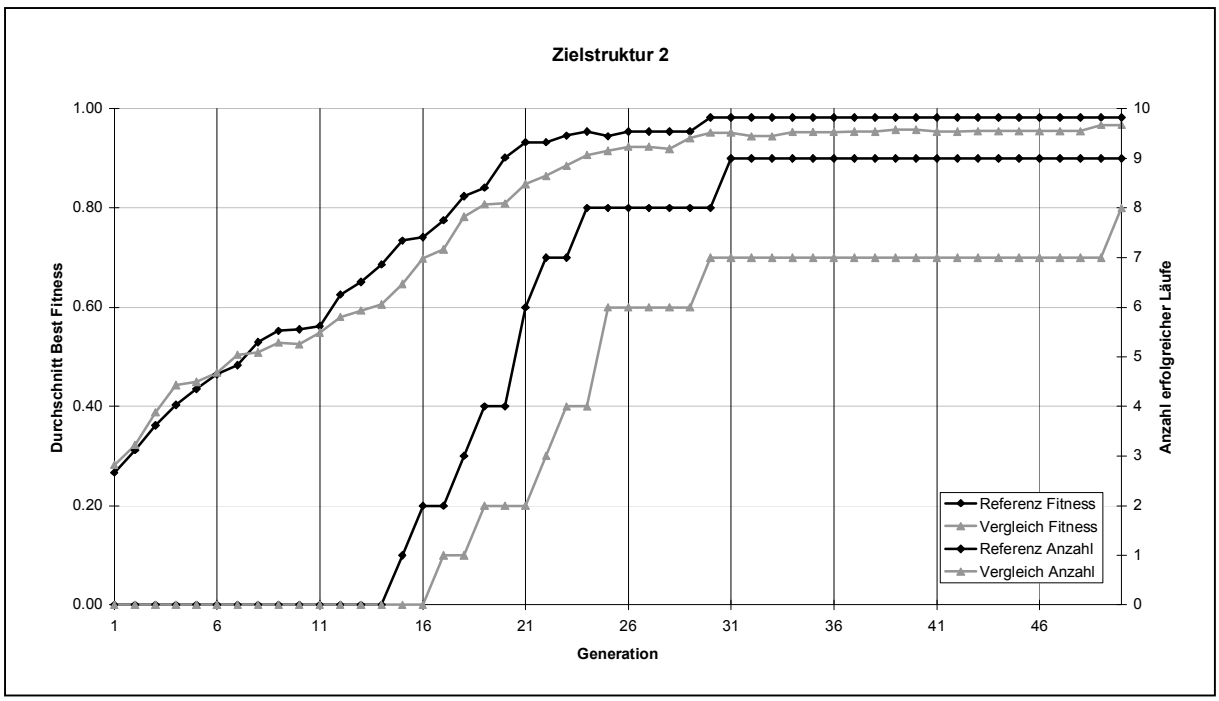
Hierbei wurden manchmal falsche Strukturen erzeugt, da die Zellverteilung der genaueste Indikator für die Morphologie einer Struktur ist.





Referenz: CellWeight 1, RatioWeight 1, DimensionWeight 1, ShootWeight 1, TerminalWeight 1
Vergleich: CellWeight 1, RatioWeight 0, DimensionWeight 1, ShootWeight 1, TerminalWeight 1





10. Tests und Ergebnisse mit den Evolutionsparametern

Nach dem gleichen Verfahren, wie in Kapitel 9 beschrieben, wurden auch einige Tests mit den Parametern, die das Evolutionsverhalten beeinflussen, gemacht.

Bei diesen Tests wurden die Fitnessfunktionen gleichgewichtet und nur einzelne Parameter der Parametereingabe verändert. Nur die veränderten Parameter werden nachfolgend bei den Diagrammen unter – Referenz – und – Vergleich – aufgeführt.

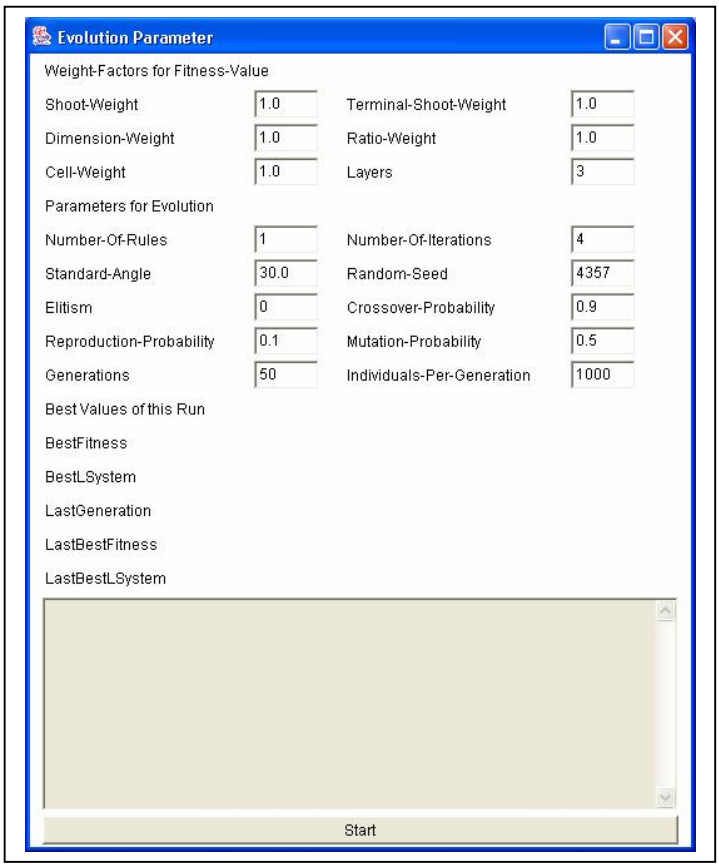
Folgende Parameter für die Evolution wurden getestet:

- Elitismus
- Kreuzungswahrscheinlichkeit
- Reproduktionswahrscheinlichkeit
- Mutationswahrscheinlichkeit
- Größe der Tournament-Selektion

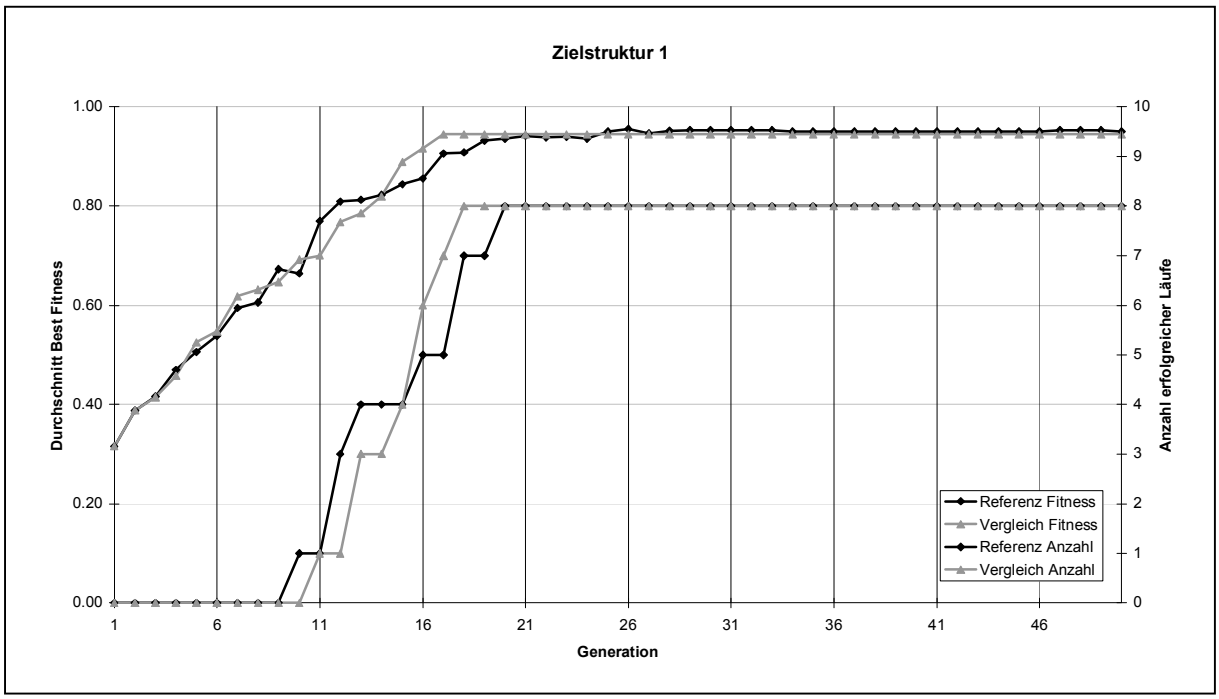
Die folgenden Ergebnisse zeigen, dass nur die Änderung der Mutationswahrscheinlichkeit auf 0.5 eine Verbesserung der gesamten Evolution verursachte.

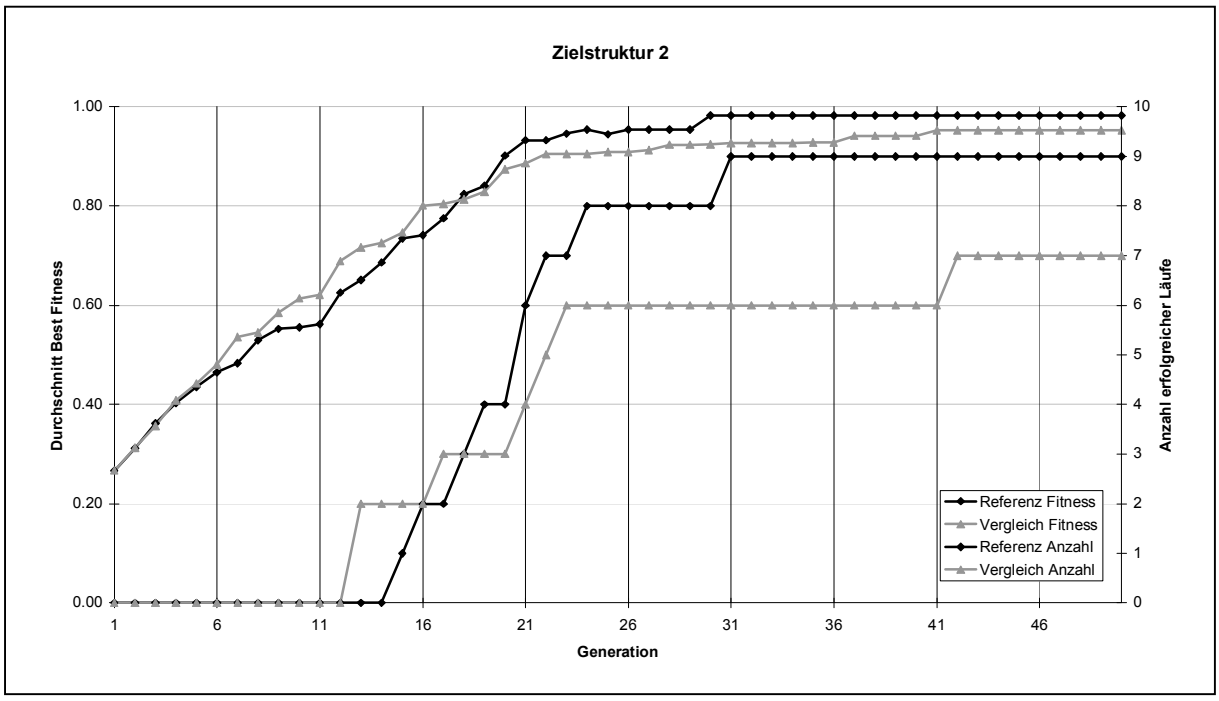
Die daraus resultierende vorerst optimale Konfiguration ist:

- Elitismus 0
- Kreuzungswahrscheinlichkeit 0.9
- Reproduktionswahrscheinlichkeit 0.1
- Mutationswahrscheinlichkeit 0.5

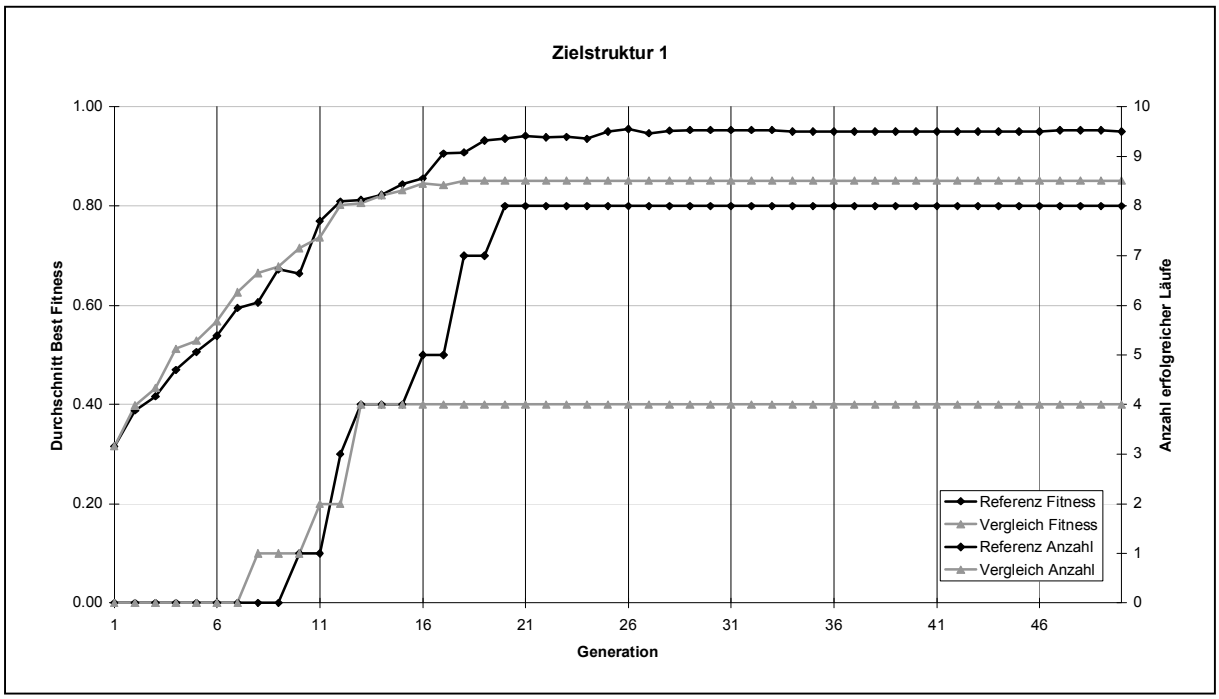


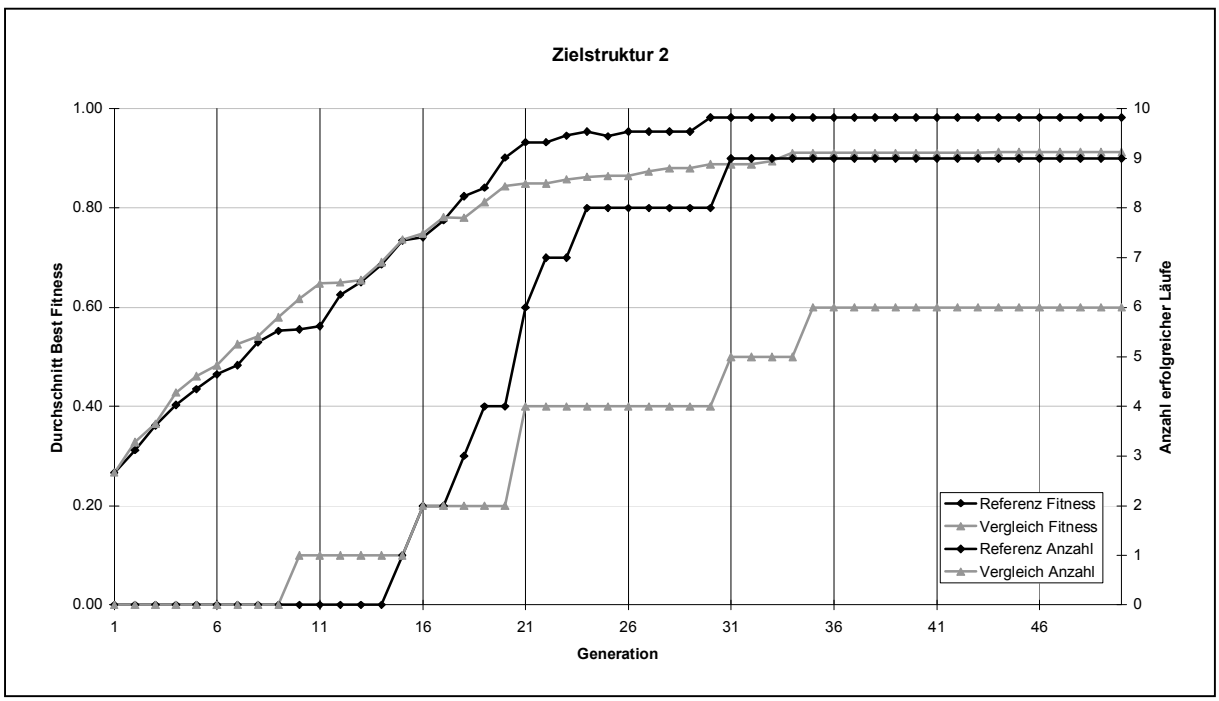
Referenz: Elite 0
Vergleich: Elite 10



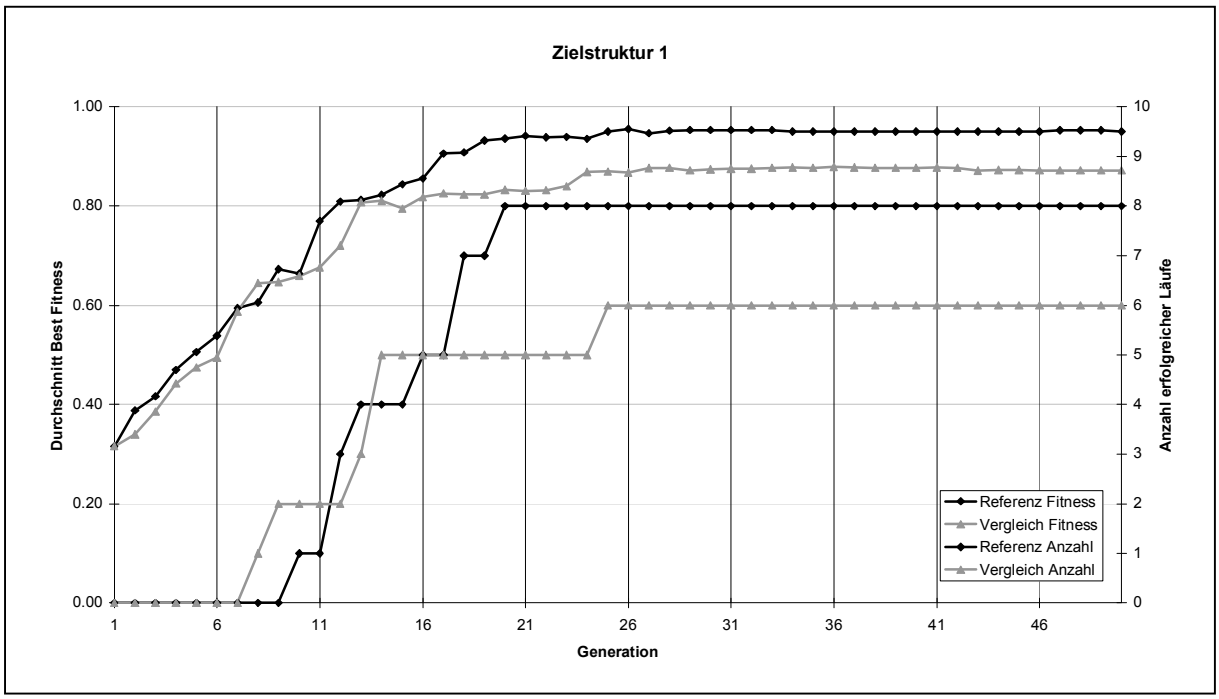


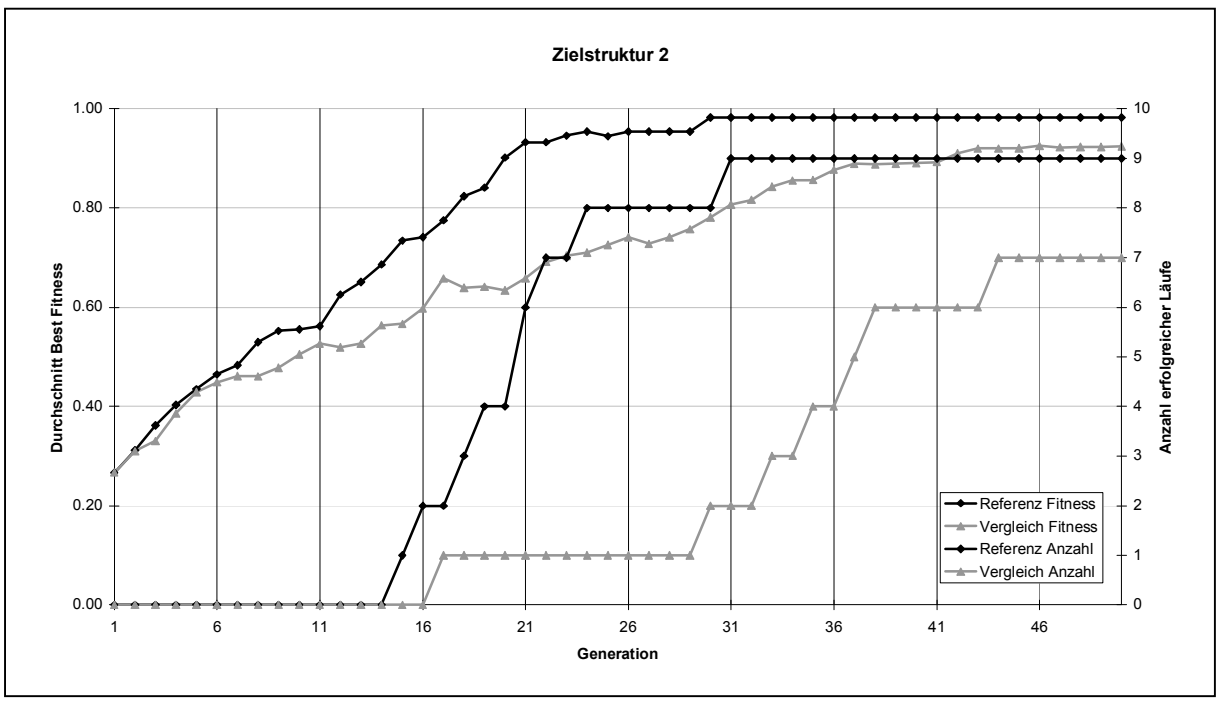
Referenz: TournamentSize 7
Vergleich: TournamentSize 10



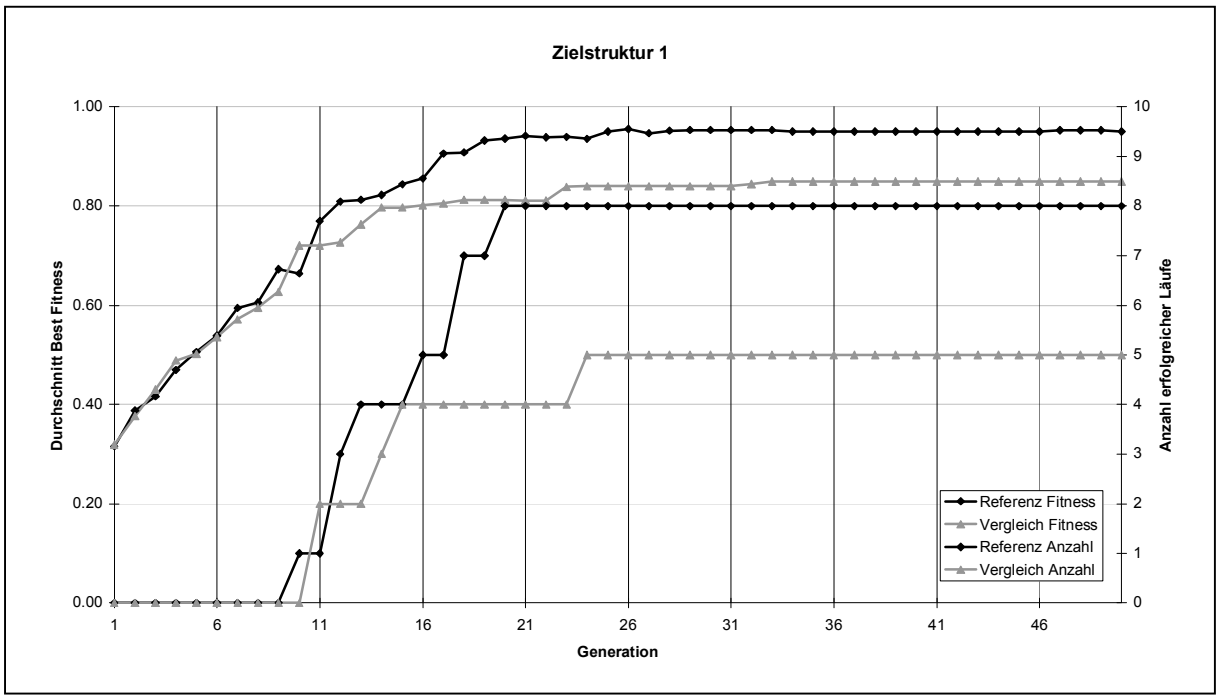


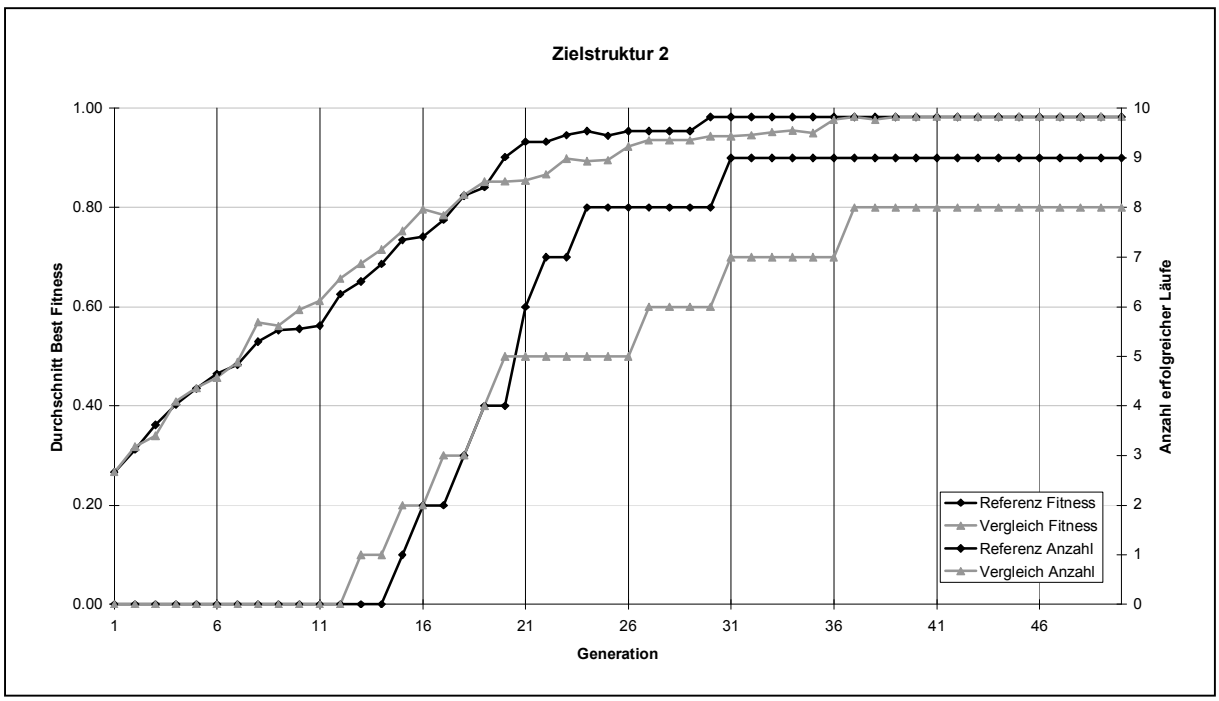
Referenz: TournamentSize 7
Vergleich: TournamentSize 5



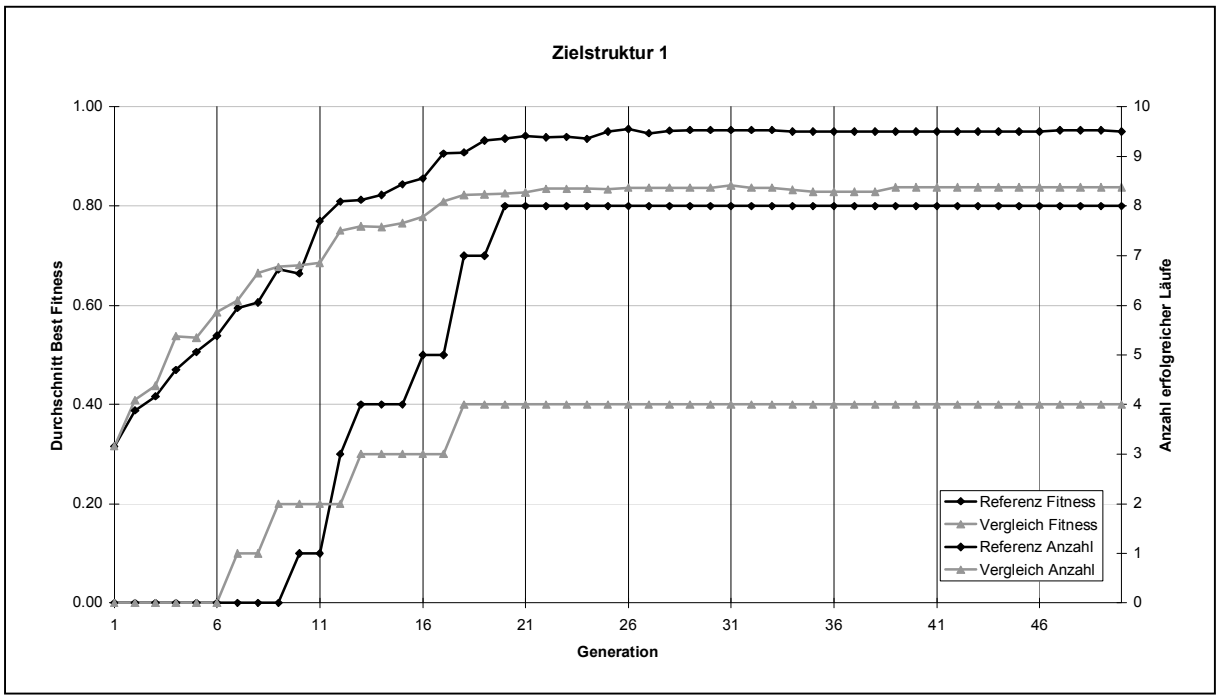


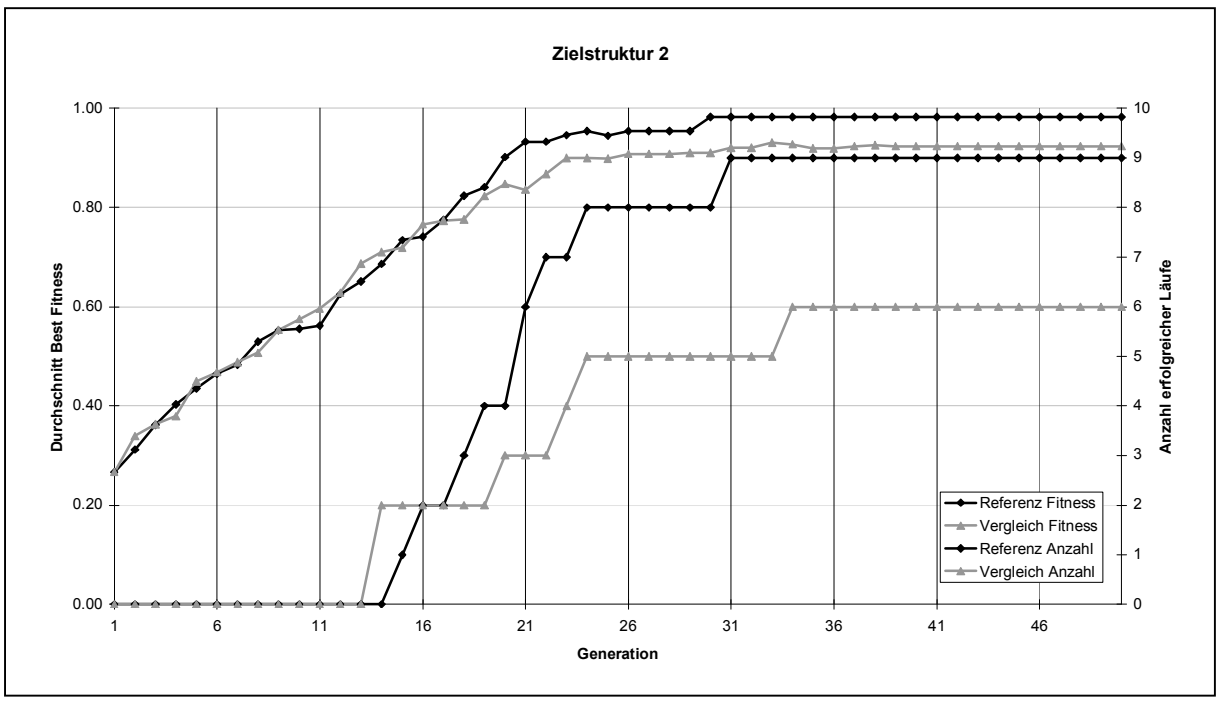
Referenz: CrossOverProb 0.9, ReproductionProb 0.1
Vergleich: CrossOverProb 0.8, ReproductionProb 0.2



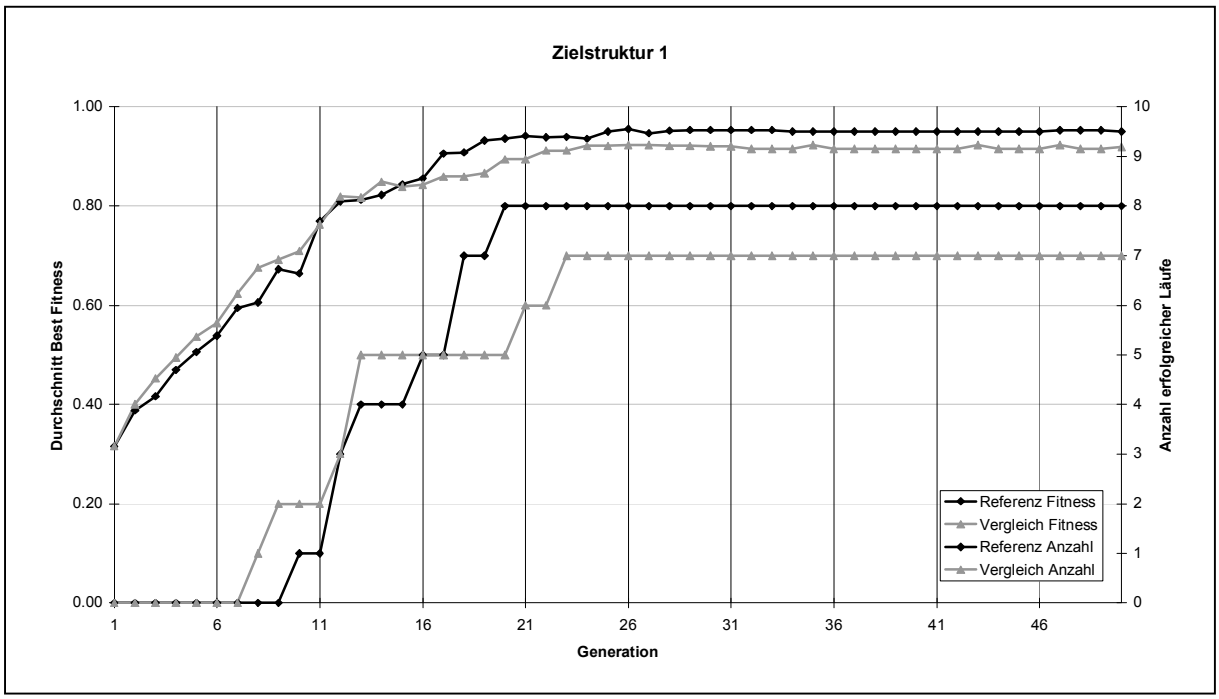


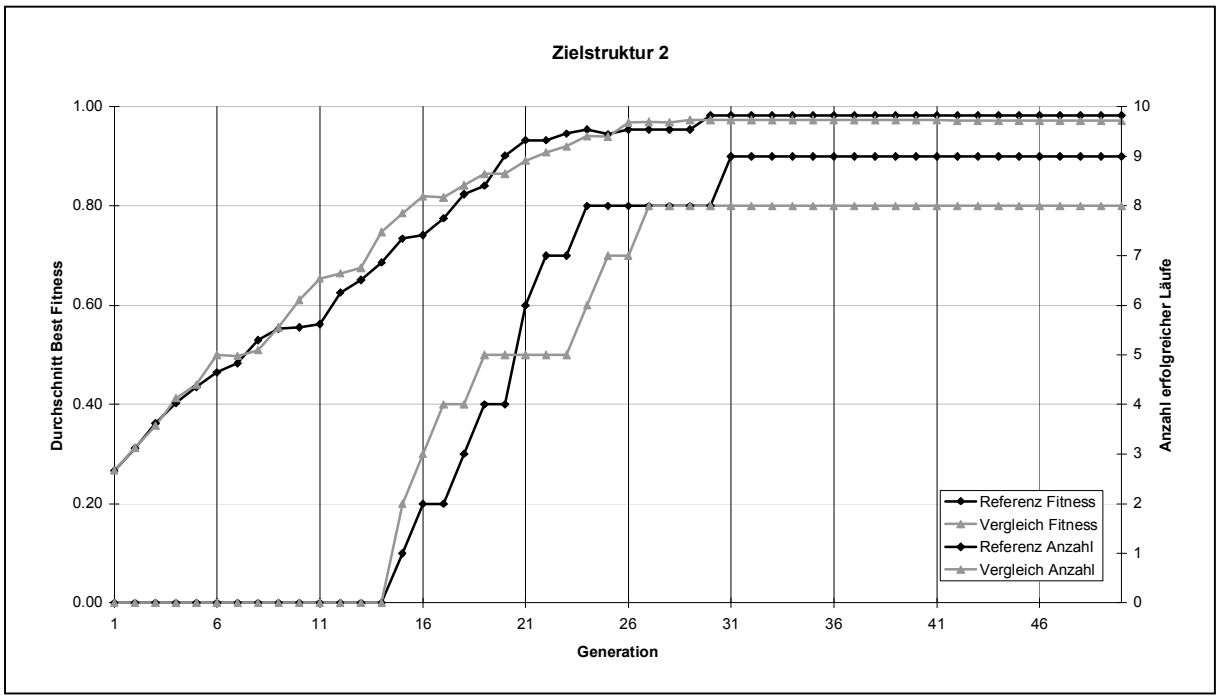
Referenz: CrossOverProb 0.9, ReproductionProb 0.1
Vergleich: CrossOverProb 0.95, ReproductionProb 0.05



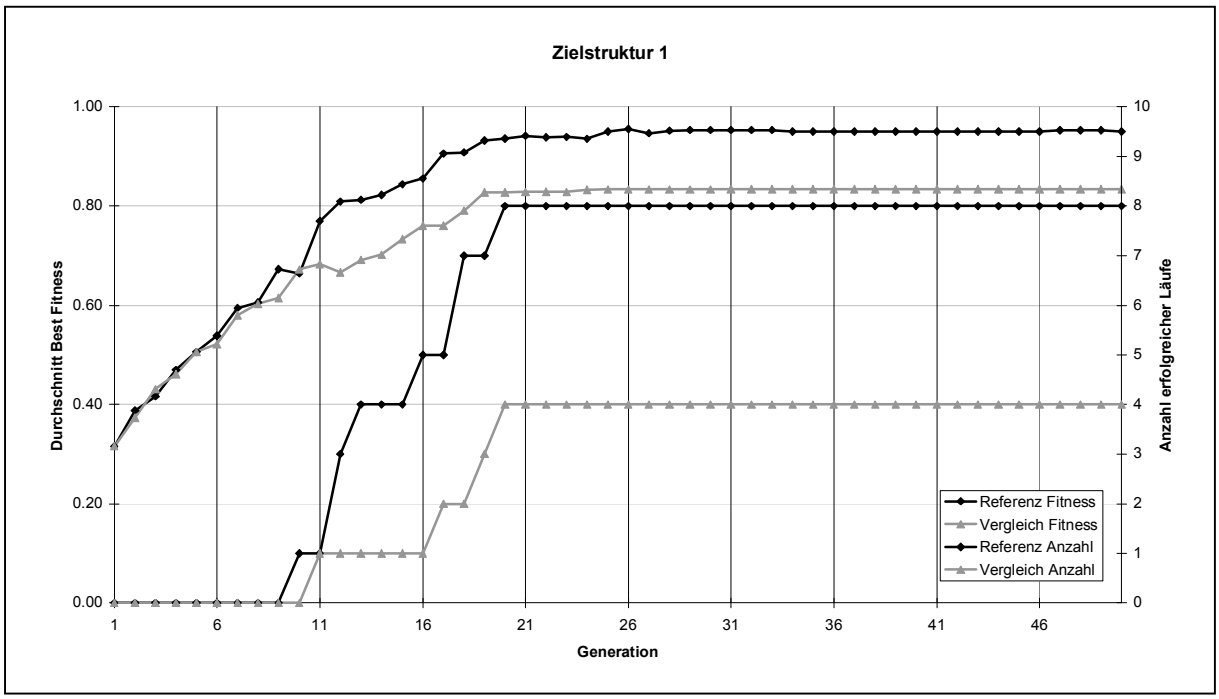


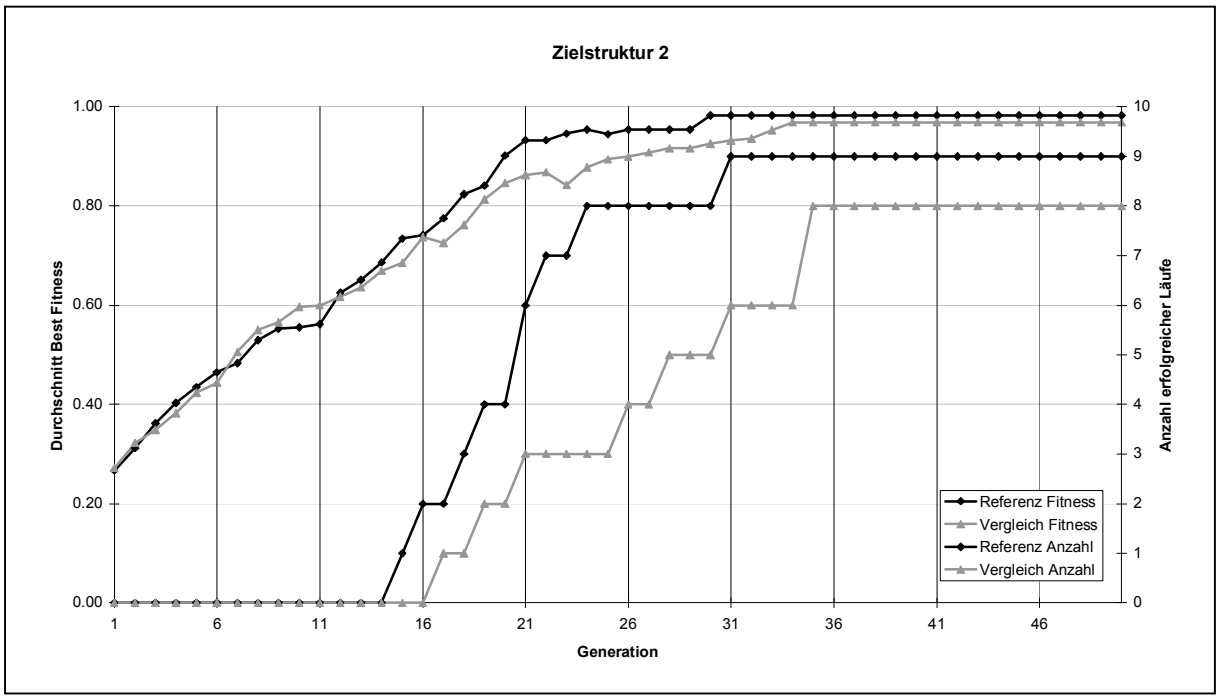
Referenz: MutationProb 0.0
Vergleich: MutationProb 0.01



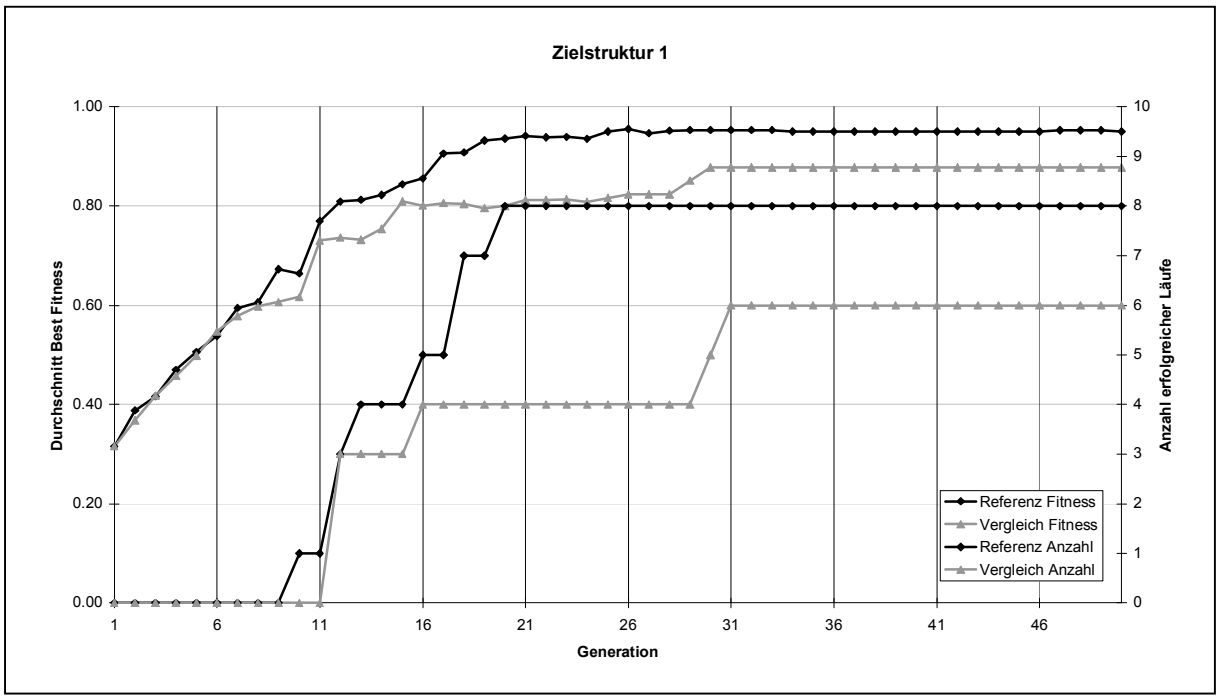


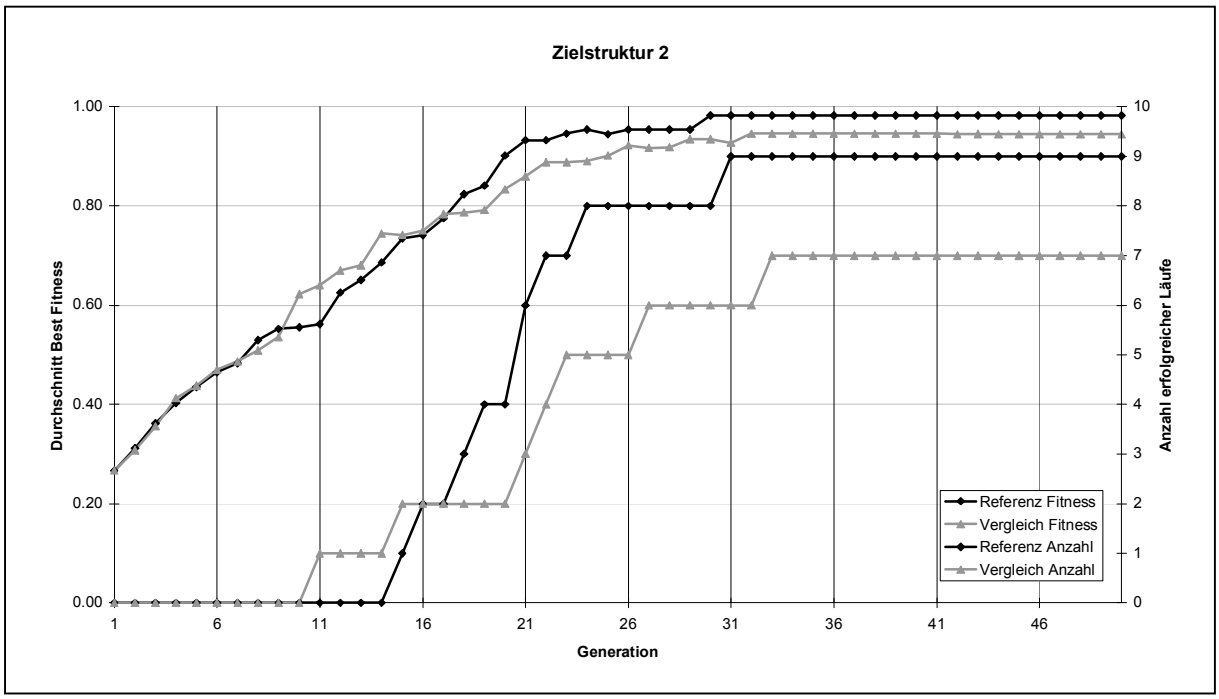
Referenz: MutationProb 0.0
Vergleich: MutationProb 0.05



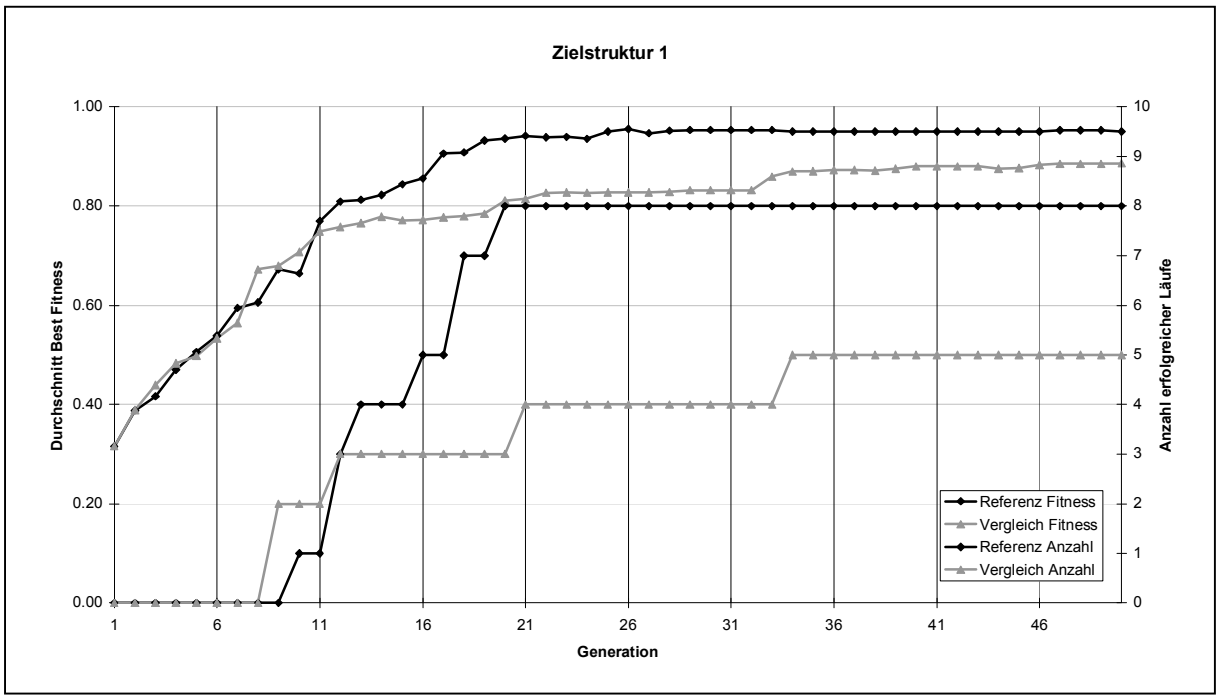


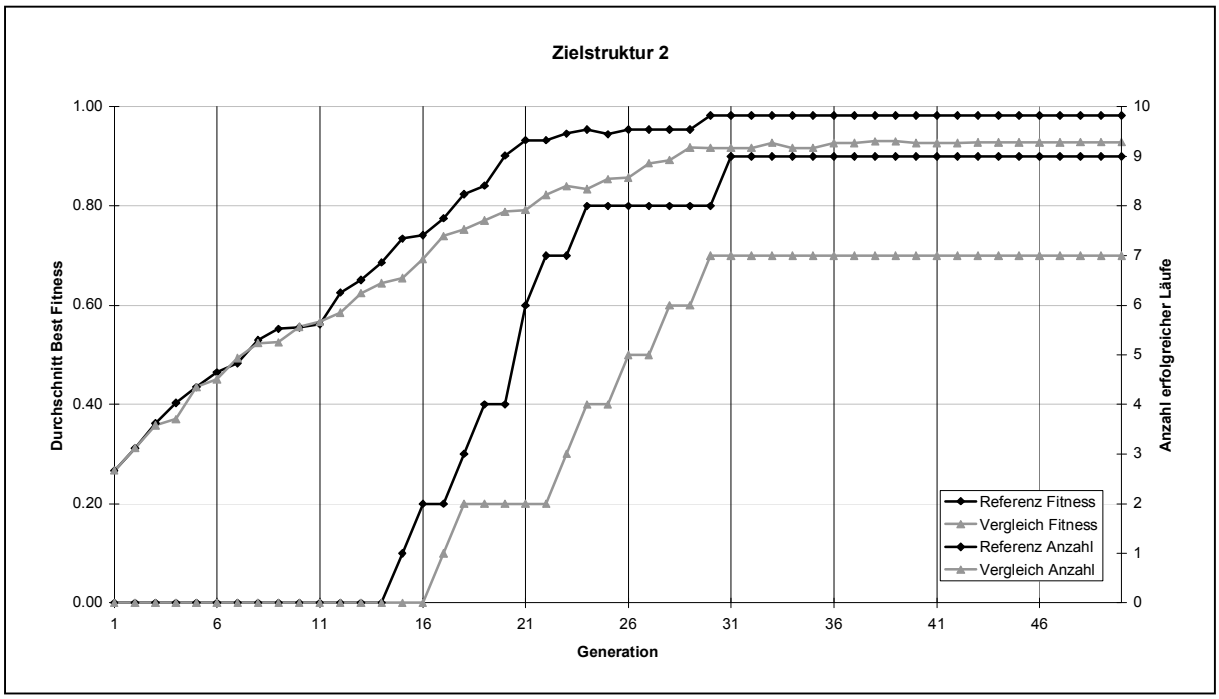
Referenz: MutationProb 0.0
Vergleich: MutationProb 0.1



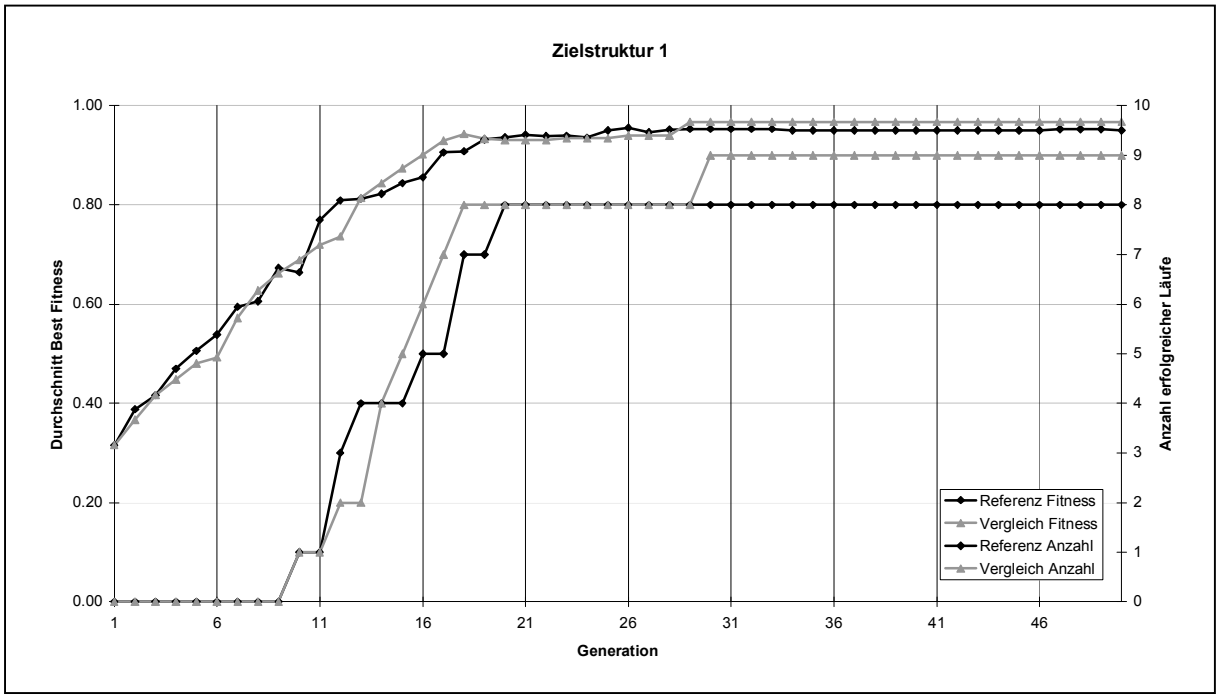


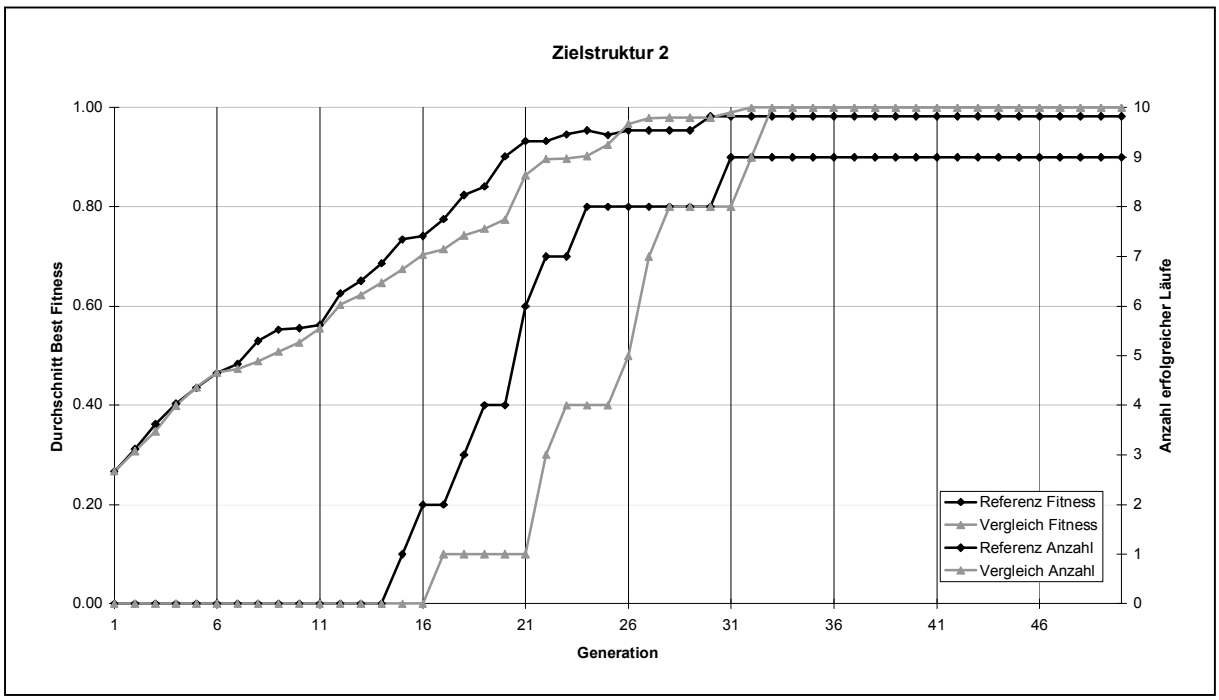
Referenz: MutationProb 0.0
Vergleich: MutationProb 0.2



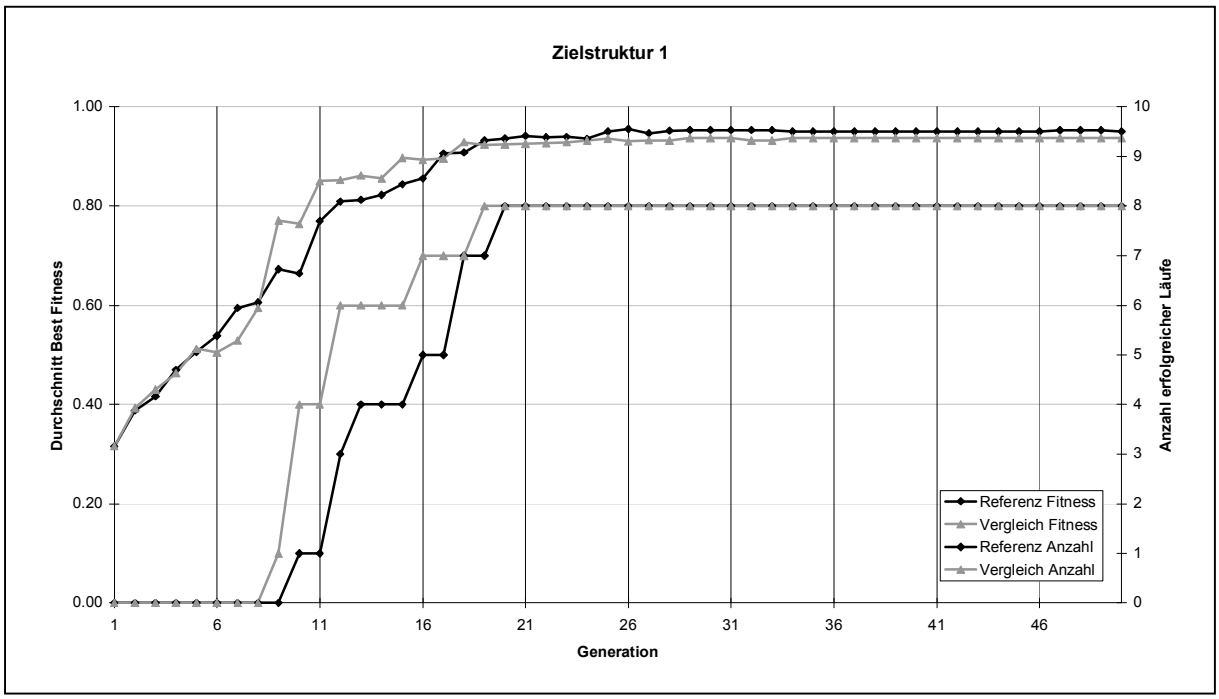


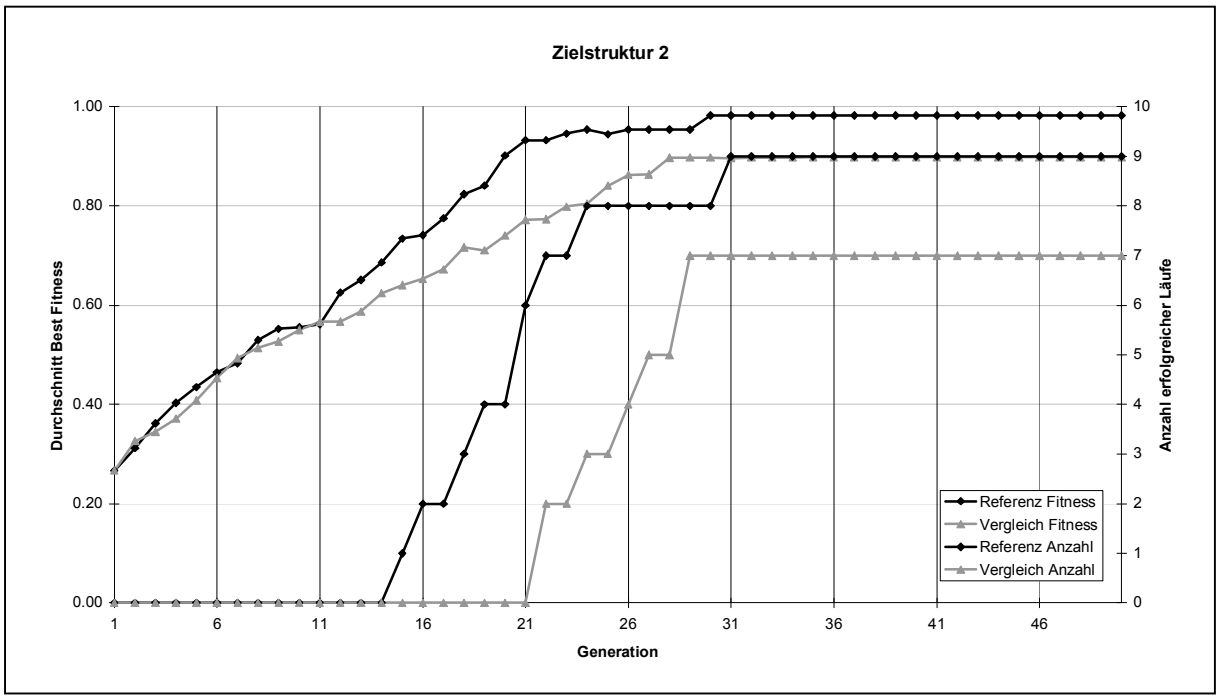
Referenz: MutationProb 0.0
Vergleich: MutationProb 0.5



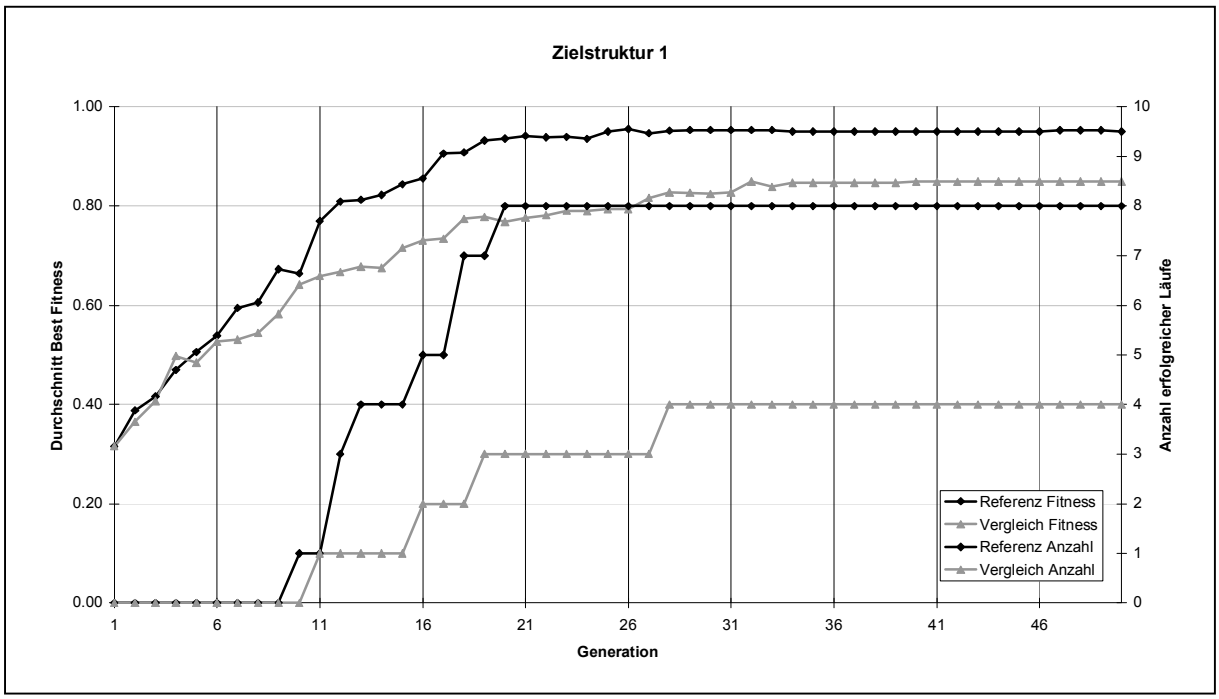


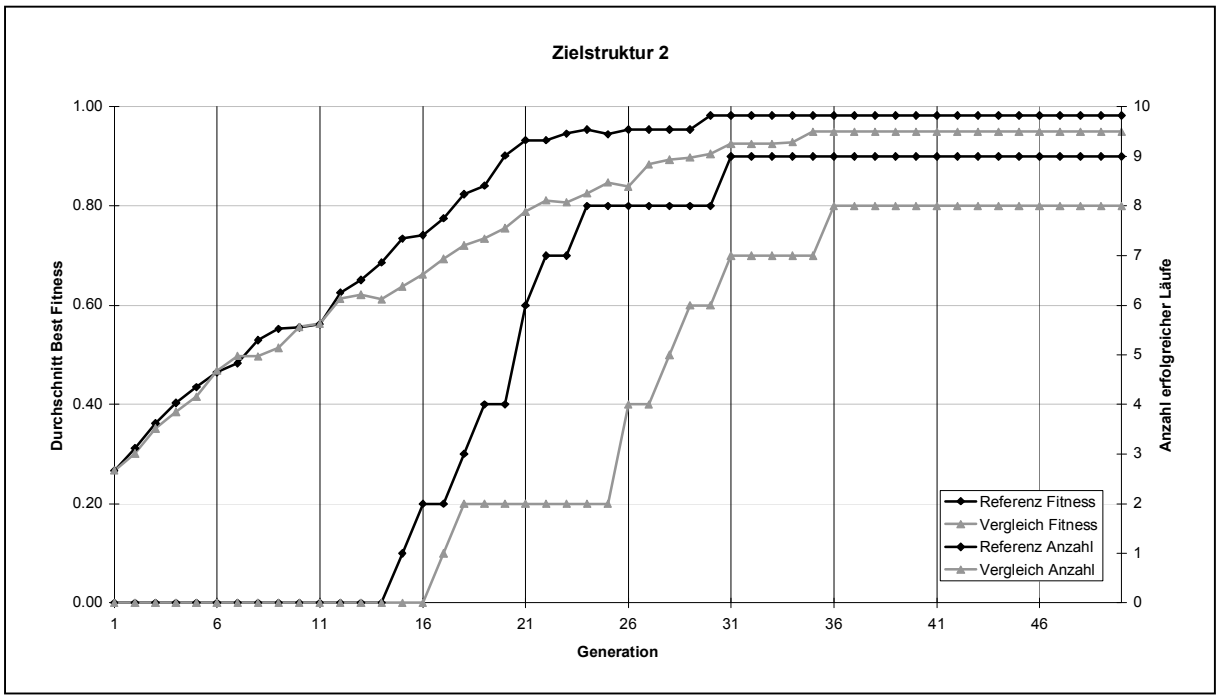
Referenz: MutationProb 0.0
Vergleich: MutationProb 0.9



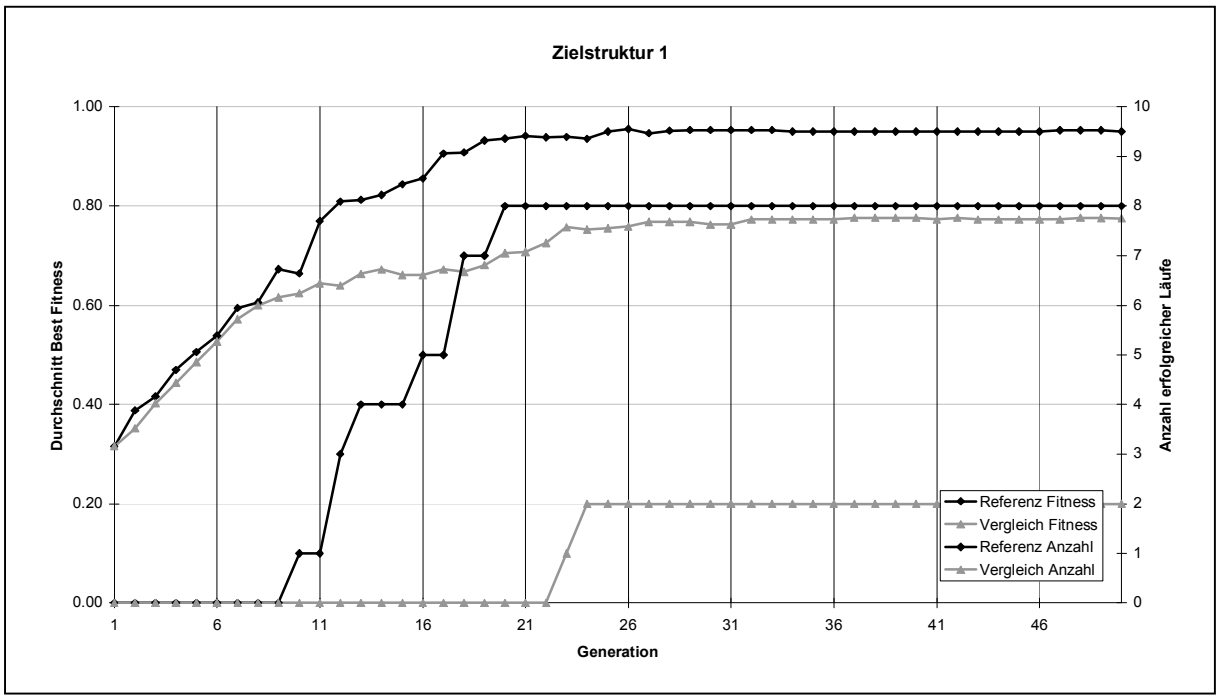


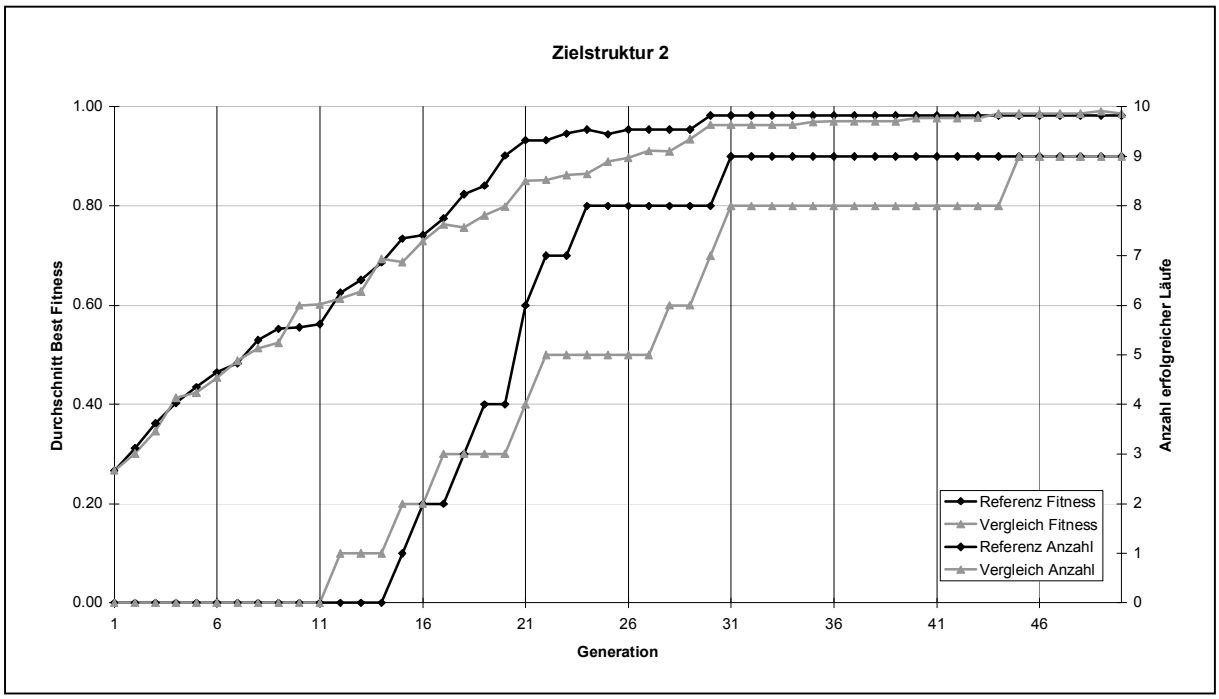
Referenz: MutationProb 0.0
Vergleich: MutationProb 0.7



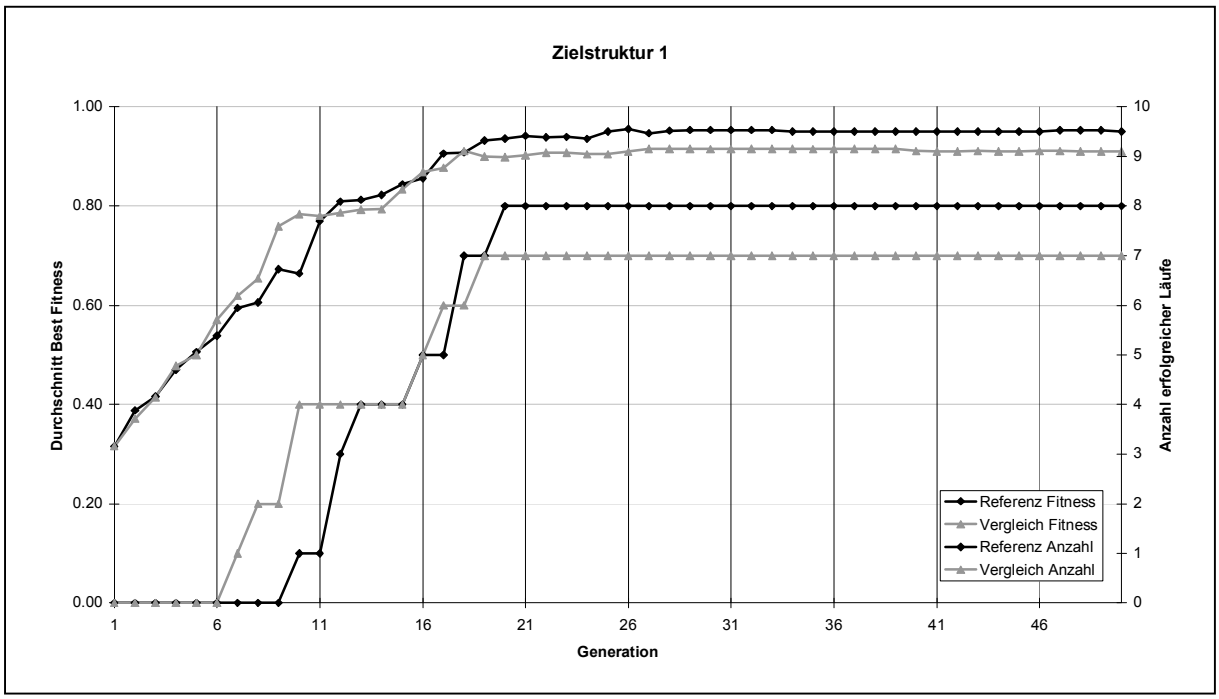


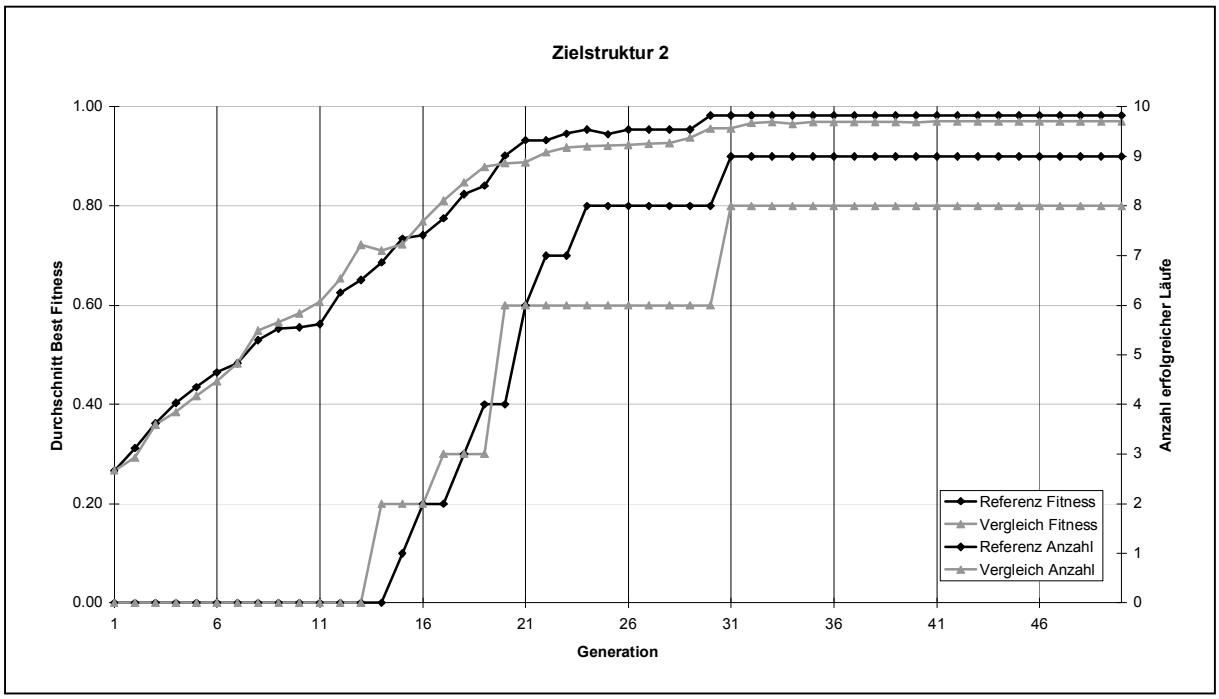
Referenz: MutationProb 0.0
Vergleich: MutationProb 0.6



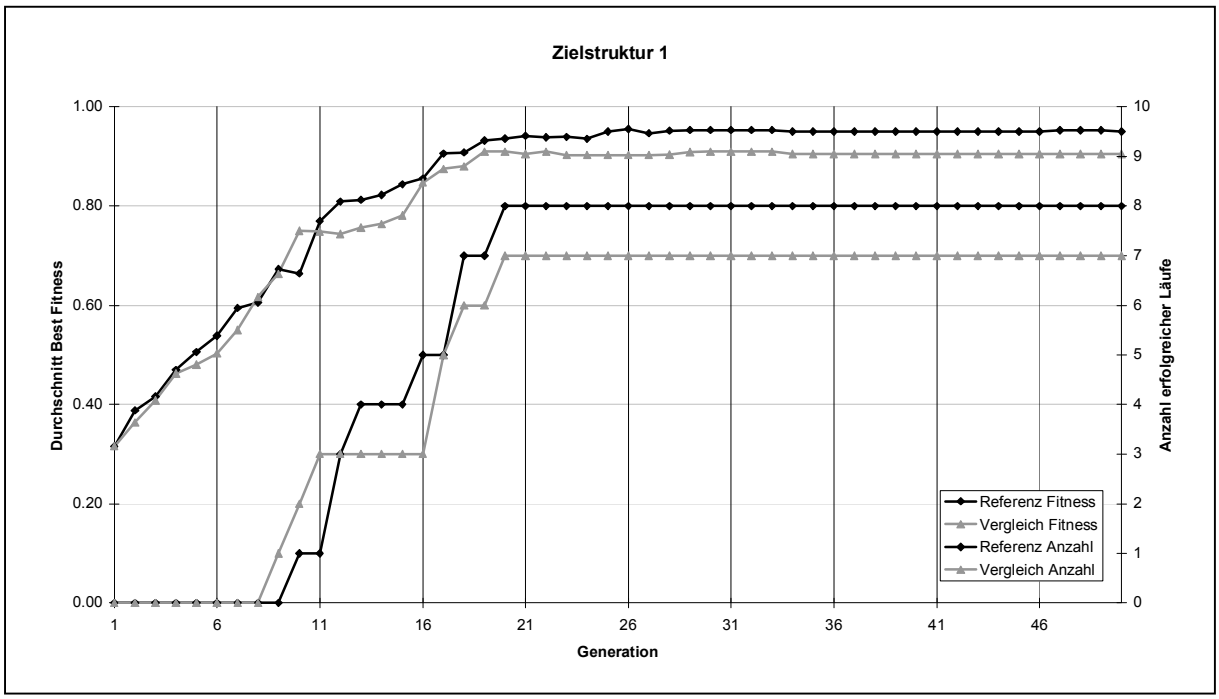


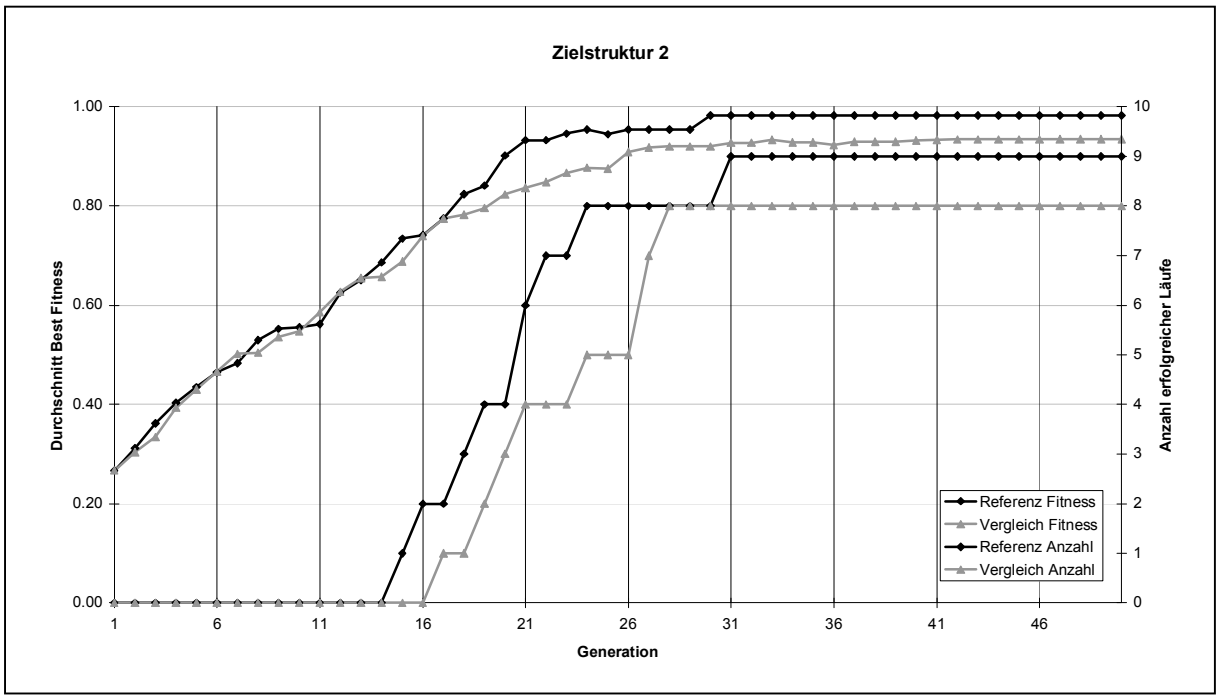
Referenz: MutationProb 0.0
Vergleich: MutationProb 0.4





Referenz: MutationProb 0.0
Vergleich: MutationProb 0.3





11. Literaturverzeichnis

M. Ebner, „Evolution and Growth of Virtual Plants“, „Advances in Artificial Life – Proceedings of the 7th European Conference on Artificial Life (ECAL)“, „Springer-Verlag“, Berlin, 2003, Seiten 228-237, http://www2.informatik.uni-wuerzburg.de/staff/ebner/welcome.html

W. Kurth, „Die Simulation der Baumarchitektur mit Wachstumsgrammatiken. Stochastische, sensitive L-Systeme als formale Basis für dynamische, morphologische Modelle der Verzweigungsstruktur von Gehölzen“, „Wissenschaftlicher Verlag Berlin“, Berlin, 1999, http://www-gs.informatik.tu-cottbus.de/~wwwgs/deutsch/publ.html

C. Jacob, „Genetic L-System Programming“, „PPSN III – Parallel Problem Solving from Nature, International Conference on Evolutionary Computation, Lecture Notes in Computer Science 866“, „Springer Verlag“, Berlin, 1994, Seiten 334-343, http://www2.informatik.uni-erlangen.de/IMMD-II/Persons/jacob

C. Jacob, „Genetic L-System Programming: Breeding and Evolving Artificial Flowers with Mathematica“, „IMS '95, Proc. First International Mathematica Symposium“, „Computational Mechanics Publications“, Southampton, 1995, Seiten 215-222, http://www2.informatik.uni-erlangen.de/IMMD-II/Persons/jacob

C. Jacob, „Evolution Programs Evolved“, „Proc. PPSN – IV, Parallel Problem Solving from Nature, Lecture Notes in Computer Science 1141“, „Springer-Verlag“, Berlin, 1996, Seiten 42-51, http://www2.informatik.uni-erlangen.de/IMMD-II/Persons/jacob

K. J. Mock, „Wildwood: The Evolution of L-System Plants for Virtual Environments“, „International Conference on Evolutionary Computing (ICEC '98)“, Anchorage Alaska, 1998, http://www.math.uaa.alaska.edu/~afkjm/

A. Ortega, A. A. Dalhoum, M. Alfonseca, „Grammatical evolution to design fractal curves with a given dimension“, „IBM Journal of Research and Development, Vol. 47, No. 4, 2003“, 2003, Seiten 483-493, http://www.research.ibm.com/journal/rd47-4.html

M. O’Neill, C. Ryan, „Grammar based function definition in Grammatical Evolution“, „Proceedings of GECCO 2000, the Genetic and Evolutionary Computation Conference“, 2000, Seiten 485-490, http://shine.csis.ul.ie/