HTWK LEIPZIG
Fakultät Informatik, Mathematik und
Naturwissenschaften
Evolutionären Algorithmen
Prof. Dr. rer. nat. Karsten Weicker

Projektarbeit

Optimale Lösungen des SameGames

– Erreichen Maximaler Punktzahlen im SameGame mithilfe eines Evolutionären Algorithmus –

vorgelegt von:

Student:	Wolfgang Teuber, B. Sc.
Studiengang:	Medien-Informatik Master
Matrikelnummer:	52693
Geburtsdatum:	07.07.1985
E-Mail:	wolfgang.teuber@gmx.net

Leipzig, den 18.02.2011

Inhalt

1 Einleitung

Das SameGame ist ein populäres Logikspiel für eine Person. Es zählt zu der Klasse von Spielen mit vollständiger Information. Das bedeutet, dass bereits zu Spielbeginn alle Informationen über das Spiel vorhanden sind. Der Spielverlauf wird demzufolge nicht durch zufällige Ereignisse beeinflusst. Das SameGame wurde 1985 von dem Japaner Kuniaki Moribe als „ChainShot!“ erfunden. Das Spielfeld ist rechteckig und in Zeilen und Spalten unterteilt. Jede dieser Zellen hat eine bestimmte Farbe. Zu Spielbeginn hat jede Zelle eine zufällige Farbe. Der Spieler kann in einem Zug eine Gruppe angrenzender Zellen gleicher Farbe entfernen, wobei darüber liegende Zellen nach unten fallen. Eine Gruppe besteht aus mindestens zwei Zellen. Die Regeln des Spiels sind im Vergleich zu anderen Logik- oder Puzzlespielen einfach, trotzdem ist das Entscheidungsproblem „*Leeren des Spielfelds*“ NP-vollständig[1]. Im Rahmen dieser Arbeit wird ein Programm entwickelt, das sowohl das Entscheidungsproblem „*Leeren des Spielfelds*“ als auch das Optimierungsproblem „*Maximale Punktzahl*“ behandelt. Zur Realisierung wird ein Evolutionärer Algorithmus eingesetzt, der in Ruby implementiert ist.

[1] [BIE02],Therese C. Bied et al. (2002): The Complexity of Clickomania, More Games of No Chance - MSRI Publications, http://library.msri.org/books/Book42/files/biedl.pdf [Stand 01.02.2011], S. 4

2 Theorie des SameGames

In diesem Abschnitt werden Begriffe im Zusammenhang dieser Arbeit erklärt. Außerdem werden theoretische Eigenschaften des SameGames erläutert. Die betrachteten Probleme „*Leeren des Spielfelds*“ und „*Maximale Punktzahl*“ werden im Abschnitt 2.2 diskutiert. Im Abschnitt 2.3 wird darauf eingegangen, wie die mathematischen Strukturen im Programm modelliert werden. Im darauffolgenden Abschnitt 2.4 wird auf den Evolutionären Algorithmus eingegangen, der im Rahmen dieser Arbeit entwickelt wird.

2.1 Allgemeines

Wie in der Einleitung erwähnt, handelt es sich beim SameGame um ein Spiel mit vollständiger Information. Diese Informationen betreffen:

- die Anzahl der Spalten *cols* (*cols* > 1),
- die Anzahl der Zeilen *rows* (*rows* > 1),
- die Anzahl der max. möglichen Farben *colors* (*colors* > 1) und
- die Farbe jeder einzelnen Zelle des Spielfelds.

Das folgende Beispiel zeigt ein **Spielfeld**:

```
A = 1 2 2 1 1      B = 3 1 1 3 3
    1 2 1 1 1          3 1 3 3 3
    3 3 1 3 1          2 2 3 1 3
    2 3 3 2 3          1 2 2 1 2
```

Die Spielfelder *A* und *B* haben je 5 Spalten, 4 Zeilen und 3 Farben. Farben werden in dieser Arbeit nicht als „rot“, „grün“, „blau“,... dargestellt, sondern als Zahlen. Die Abbildung 1 zu „rot“, 2 zu „blau“, usw. ist beliebig, denn nicht der Farbwert ist von Interesse, sondern dessen Verteilung auf dem Spielfeld. Die Spielfelder *A* und *B* sind demzufolge gleich, weil die Verteilung der Farbwerte gleich ist. Die Nummerierung der Spalten beginnt links mit 0 und wird nach recht größer. Die Nummerierung der Zeilen beginnt ebenfalls mit 0 und wird nach oben größer. Die Zelle *A*(0,0) hat den Wert 2, *A*(0,2) den Wert 1.

Als **Gruppe** werden Zellen bezeichnet, die die gleiche Farbe haben und im Sinne einer Vierernachbarschaft aneinanderliegen. Die Zelle *A*(*col*, *row*) hat 4 anliegende Zellen: *A*(*col*+1, *row*), *A*(*col*-1, *row*), *A*(*col*, *row*+1), *A*(*col*, *row*-1).

Randzellen haben nur innere Nachbarn. Kantenzellen haben demzufolge 3 Nachbarn und Eckzellen 2. Eine Gruppe besteht aus mindestens zwei (*group* = *area* = 2) Zellen. Eine Gruppe kann vollständig mit einem **Zug** vom Spieler entfernt werden. Der Zustand des Feldes wird dadurch verändert. Alle Zellen „fallen“ zunächst nach unten, falls unter ihnen durch den Zug eine leere Zelle entstanden ist. Danach rutschen alle Spalten nach links, falls leere Spalten entstanden sind. Welche Gruppe der Spieler entfernt, ist ihm überlassen. Das Spiel endet, wenn kein Zug mehr möglich ist. Im folgenden Beispiel wird die Gruppe fett gedruckt, die als nächstes entfernt wird (von links nach rechts).

1 **2** **2** 1 1	1 0 0 1 1	0 0 1 1 0	0 1 1 0 0
1 **2** 1 1 1	1 0 1 1 1	**1** 0 1 1 0	0 1 1 0 0
3 3 1 3 1	**3** **3** 1 3 1	**1** **1** 3 1 0	0 3 1 0 0
2 3 3 2 3	2 **3** **3** 2 3	2 **1** 2 3 0	2 2 3 0 0

Die Zahl 0 steht für ein leeres Feld. Der Spieler kann noch 3 weitere Züge machen, um das Spielfeld zu leeren. Eine **Zugfolge** wird als Folge von Zahlen dargestellt, wobei jede Zahl für den Index der gewählten Gruppe steht, beginnend mit 1. Die Nummerierung beginnt bei Zelle (0,0) und wird erst reihenweise (nach oben) fortgesetzt und dann spaltenweise (nach rechts). Eine 0 bedeutet, dass auf dieser Zelle kein Zug möglich ist. Für das obige Beispiel werden nun die Indexe für alle möglichen Züge dargestellt:

2 **3** **3** 4 4	2 0 0 3 3	0 0 2 2 0	0 2 2 0 0
2 **3** 4 4 4	2 0 3 3 3	**1** 0 2 2 0	0 2 2 0 0
1 1 4 0 4	**1** **1** 3 0 3	**1** **1** 0 2 0	0 0 2 0 0
0 1 1 0 0	0 **1** **1** 0 0	0 **1** 0 0 0	1 1 0 0 0

Der Spieler hat sich für die Zugfolge [3,1,1,...] entschieden. Für jeden Zug bekommt der Spieler Punkte, die auf den Anfangspunktstand von 0 aufaddiert werden. Für die Berechnung der Punkte gilt allgemein: je größer die Gruppe des Zuges, desto mehr Punkte erhält der Spieler für diesen Zug. Für die Betrachtung in dieser Arbeit werden die Punkte wie folgt berechnet: Punkte = (Gruppengröße des Zugs - 2) ^ 2. Für das Entfernen von zwei Zellen bekommt der Spieler also 0 Punkte, für 3 entfernte Zellen 1 Punkt, für 4 4 Punkte, für 5 9 Punkte, usw. Beendet der Spieler das angefangene Beispiel, so hat er am Ende 18 Punkte. Die Problemstellung „*Maximale Punktzahl*“ beschäftigt sich mit der

Frage, ob es eine Lösung gibt, mit der man mehr Punkte erhält und welches die beste (optimale) Lösung ist.

2.2 Probleme des SameGames

Um die betrachteten Probleme zu lösen, wird das Spiel eines Spielers ebenfalls formalisiert. Ein Spieler wählt in jedem Zustand des Spielfelds genau einen der möglichen Züge. Ist kein Zug mehr möglich, endet das Spiel. Der Spieler hat eine bestimmte Punktzahl. Diese Punktzahl zu maximieren ist das Lösungsziel des Optimierungsproblems „Maximale Punktzahl“. Das Entscheidungsproblem „*Leeren des Spielfelds*“ wird immer dann gelöst, wenn das Spielfeld bei Spielende vollständig geleert ist. Beide Probleme können nicht erschöpfend gelöst werden. Ob ein globales Maximum gefunden wurde und ob ein Spielfeld nicht geleert werden kann, ist mit dem Evolutionären Algorithmus nicht berechenbar.

Es gelten folgende theoretische Extrema für das SameGame:

- Die **maximale Punktzahl**: ((*cols* * *rows* - 2) ^ 2
  Dies tritt ein, wenn das gesamte Spielfeld einfarbig ist. Dazu benötigt der Spieler genau einen Zug (Zuglänge 1).
- Die **minimale Punktzahl**: 0
  Kein Zug ist möglich, oder in jedem durchgeführten Zug wurden genau 2 Zellen entfernt.
- Die **längste Zugfolge** hat die Länge: *floor*((*cols* * *rows) / group*)
  Es werden solange Zweiergruppen entfernt, bis weniger als zwei Zellen übrig bleiben.
- Die **kürzeste Zugfolge** hat die Länge: 0
  Von Spielbeginn an ist kein Zug möglich.

Der Zusammenhang zwischen theoretisch maximaler Punktzahl mit Zuglänge 1 und 0 Punkte mit der theoretisch maximalen Zuglänge, lässt eine Annahme entstehen: Unter der Bedingung, dass gleich viele Zellen geleert werden und alle bisherigen Festlegungen gelten, führt eine kürzere Zugfolge zu mehr Punkten als eine längere. Sollte diese Annahme stimmen, so wären die Lösung der Optimierungsprobleme „*Maximale Punktzahl*“ und „*Kürzeste Zugfolge*“ gleich.

Weiterhin gilt:

- Punkte pro Zug: *points*(*size*) = (*size* - 2) ^ 2
- maximal mögliche Punktzahl mit instgesamt *n* entfernten Zellen und *x* Zügen: *max*(*n*, *x*) = *points*(*n* - 2 * (*x* - 1))
- minimal mögliche Punktzahl:
  min(*n*, *x*) = (*n mod x*) * *points*(*ceil*(*n* / *x*) + 1)
  + (*x* – *n mod x*) * *points*(*floor*(n / x))

Wenn die Annahme zutrifft, dann gilt für alle *n* und *x*:

min(*n*, *x*) ≥ *max*(*n*, *x* + *i*), *i* > 0

Theoretisches Gegenbeispiel:

min(12, 2) = 32

max(12, 2 + 1) = 36

Praktisches Gegenbeispiel:

Das 10x10 Zellen große Spielfeld

```
M = 3 1 3 1 3 1 2 1 3 2
    2 3 1 1 1 1 3 3 2 1
    1 2 1 2 2 1 3 3 2 3
    2 2 2 1 3 2 2 3 2 3
    3 1 3 2 3 3 1 1 1 3
    1 1 1 3 2 2 3 3 1 2
    2 1 3 2 3 2 3 3 1 2
    3 2 2 2 1 1 2 1 2 1
    1 3 1 1 1 3 1 1 2 2
    1 1 3 1 1 2 2 3 2 3
```

wird sowohl mit der Zugfolge

z1 = [3,4,11,3,14,6,8,7,6,4,6,5,3,3,3,1,5,4,2,1,2,1,2,2,1] (Länge 25)

als auch mit der Zugfolge

z2 = [12,10,11,13,14,11,1,1,8,8,2,5,4,2,3,2,1,3,1,2,1,1,1] (Länge 23)

geleert. Trotzdem werden mit der längeren Zugfolge *z1* auf dem Spielfeld *M* 244 Punkte erreicht, wohingegen mit *z2* nur 242 Punkte erreicht werden. Die ursprüngliche Annahme ist somit falsch. Die Optimierungsprobleme „*Maximale Punktzahl*“ und „*Kürzeste Zugfolge*“ sind getrennt zu betrachten. Diese Arbeit beschäftigt sich im Folgenden mit nur einem Optimierungsproblem: „*Maximale Punktzahl*“.

Das Entscheidungsproblem „*Leeren des Spielfelds*“ beschäftigt sich mit der Aussage über die Möglichkeit ein Spielfeld zu leeren. Die Aussage ist entweder positiv: Es gibt mindestens eine Zugfolge, die das Spielfeld leert, oder negativ: Es gibt keine Zugfolge, die das Spielfeld leert. Das heißt, dass ein Spielfeld nicht geleert werden kann, wenn:

- Mindestens eine Farbe zu Spielbeginn weniger als *group* mal vor kommt.
- Zu Spielbeginn kein Zug möglich ist
- Bei allen möglichen Zugfolgen mindestens eine Zelle nicht geleert ist

Das Entscheidungsproblem wird vom Evolutionären Algorithmus nur teilweise gelöst. Nämlich in dem Fall, wenn eine Lösung gefunden wurde, die das Spielfeld leert. Die begründete Aussage, dass ein Spielfeld nicht geleert werden kann, kann nicht getroffen werden.

2.3 Modellierung

In diesem Abschnitt wird beschrieben, wie die in den Abschnitten 2.1 und 2.2 eingeführten Größen modelliert bzw. kodiert werden. Dazu werden deren Datenstrukturen und Semantik beschrieben. Ziel der Modellierung ist es, diese Größen in der Programmiersprache Ruby auszudrücken, damit der Evolutionäre Algorithmus darauf arbeiten kann.

Das Spielfeld ist eine Matrix und wird als zweidimensionales Feld (*Array)* von Zahlen (*Numeric*) beschrieben. Beispielsweise steht [[1,2,3],[4,5,6],[7,8,9], [10,11,12]] für die Matrix:

```
3  6  9  12
2  5  8  11
1  4  7  10
```

Dabei ist darauf zu achten, dass die Orientierung des Spielfelds spaltenweise von links nach recht und zeilenweise von unten nach oben ist. Die Position (0,0) beschreibt demzufolge die linke untere Zelle. Diese Matrix wird durch eine Instanzvariable der Klasse **SameGame** repräsentiert. Die Verwendung der Klasse SameGame wird im Abschnitt 3.1.4 genauer beschrieben.

Eine Folge von Zügen ist ebenfalls ein Feld (*Array*) von Zahlen (*Numeric*), dieses Feld ist eindimensional. Jede Zugfolge wird mit der Länge der theoretisch längsten Zugfolge *n = floor*((*cols* * *rows) / group*) initialisiert. Mit

der Zugfolge [hit_1,hit_2,...,hit_n] wird zuerst der Zug mit dem Index hit_1 ausgeführt. Danach wird der Zug mit dem Index hit_2 usw. Der Index bezieht sich auf die Anzahl der möglichen Züge im aktuellen Zustand des Spielfelds. Sollte vor *n* Zügen ein Zustand erreicht werden, in dem kein Zug mehr möglich ist, so werden die restlichen Zugindexe ignoriert. Eine Zugfolge wird in einer Instanzvariable der Klasse **Individual** (Individuum) gespeichert. Individuen werden mit computergenerierten Zufallswerten initialisiert. Deswegen wird nachträglich sichergestellt, dass der angegebene Zug ausgeführt werden kann. Im Abschnitt 2.4 wird darauf näher eingegangen. Die Zugfolge stellt den Genotyp eines Individuums dar. Die anderen Eigenschaften eines Individuums ergeben sich zur Laufzeit und hängen von seinem Genotyp und dem Spielfeld ab. Dazu gehören dessen Güte (Fitness), Phänotyp, reparierter Genotyp. Die Güte ist eine Zahl (*Numeric*). Der Phänotyp ist ein zweidimensionales Feld (*Array*) von Zahlen (*Numeric*). Jeder Zug der Zugfolge eines Individuums ergibt eine Positionsangabe (*col*, *row*) für den Phänotyp: [[*col*(hit_1),*row*(hit_1)], [*col*(hit_2),*row*(hit_2)],...,[*col*(hit_n),*row*(hit_n)]].

Individuen werden wiederum in einem Feld (*Array*) gespeichert. Dieses Feld von Individuen (*Individual*) wird als Instanzvariable der Klasse **Population** gespeichert: [ind_1,ind_2,...,ind_p]. Die Populationsgröße *p* (Anzahl der Individuen in einer Population) wird vom Nutzer festgelegt.

2.4 Der Evolutionäre Algorithmus

In diesem Abschnitt wird beschrieben wie die Suche nach optimalen Lösungen für eine „*Maximale Punktzahl*“ umgesetzt ist. Dazu wird davon ausgegangen, dass die grundlegenden Konzepte von Evolutionären Algorithmen bekannt sind. In dieser Arbeit wird ein Evolutionärer Algorithmus eingesetzt, der mit dem Populationskonzept arbeitet. Dem verwendeten Algorithmus liegt ein Hillclimber zugrunde. Jedes Individuum liefert ein lokales Optimum. Bei der gewählten Kodierung stehen die Güte zweier Individuen und die Ähnlichkeit ihres Genotyps nicht im Verhältnis. Aus diesem Grund wird der Hillclimber so angepasst, dass auch schwächere Individuen überleben bzw. neu erzeugt werden.

Folgende Größen werden vom Nutzer festgelegt:

- Populationsgröße

- max. Generationsanzahl
- Mutationswahrscheinlichkeit
- Rekombinationswahrscheinlichkeit
- Bewertungskriterium „Geleertes Spielfeld“
- Abbruchkriterium „Geleertes Spielfeld“

Für die Initialisierung ist vor allem die Populationsgröße von Bedeutung. Sie gibt die Anzahl der initial erzeugten zufälligen Individuen an. Pro Generation wird zunächst die Güte der Eltern berechnet. Für die Berechnung der **Güte** wird der Genotyp eines Individuums repariert. Die **Reparatur** findet statt, nachdem die Anzahl der möglichen Züge in aktuellen Zustand ermittelt wurde. Sollte der Genotyp einen unmöglichen Zug beschreiben, so wird dieser mittels modulo bezüglich der Anzahl möglicher Züge repariert. Demzufolge wird der Genotyp 12 bei nur 9 möglichen Zügen zu $(12 \textit{ mod } 9 + 1) = 4$ korrigiert. Da die Indexzählung bei 1 beginnt, wird 1 addiert. Auf diese Weise lassen sich beliebige Individuen anhand ihrer Genotypen bewerten. Die Bewertung eines Individuums bricht ab, wenn ein finaler Zustand erreicht ist. Ein Abbruch der Güteberechnung vor dem erreichen eines finalen Zustandes ist nicht zielführend, insbesondere weil in der Regel optimale Lösungen die punktereichsten Züge gegen Spielende ausführen. Eine Elternselektion hat sich nicht als nützlich herausgestellt und wird aus diesem Grund nicht ausgeführt. Die Kinder werden durch **Rekombination** und **Mutation** in Abhängigkeit von den jeweiligen Wahrscheinlichkeiten erzeugt. Als Rekombinationsoperation wurde der 1-Punkt-Crossover gewählt. Dafür werden zwei zufällige Individuen *A* und *B* gewählt. Die Genotypen von *A* und *B* werden so verbunden, dass der Genotyp von A vor dem erreichen eines finalen Zustands abgeschnitten wird und der Genotyp von B ab diesem Punkt angefügt wird. Da die Längen der Genotypen identisch sind, entsteht ein bewertbares neues Individuum A'. Ist die Rekombinationswahrscheinlichkeit 80% so werden im Mittel 8 von 10 Individuen rekombiniert. Bei der Rekombination wird sichergestellt, dass ein Individuum nicht mit sich selbst rekombiniert wird. Als Mutationsoperation wird die vollständige Neuerzeugung eines zufälligen Genotyps gewählt. Im Lauf der Entwicklung des praktischen Teils dieser Arbeit hat sich diese Methode als zuverlässig herausgestellt. Für die Population gilt bei einer Mutationswahrscheinlichkeit von 80%, dass im Mittel 8 von 10 Individuen

mutiert werden. Nachdem die Güte der Kind-Population berechnet ist, findet die **Umweltselektion** statt. Dazu werden die besten 10% (mindestens ein Individuum) beider Populationen übernommen. Hat der Nutzer das Bewertungskriterium „Geleertes Spielfeld“ angegeben, dann werden die besten 10% vorrangig nach „Leeren des Spielfelds“ sortiert und nachrangig nach ihrer erreichten Punktzahl. Eine Mehrzieloptimierung im allgemeinen Sinn ist an dieser Stelle nicht angebracht, da das „Leeren des Spielfelds“ ein Entscheidungsproblem ist und kein Optimierungsproblem. Alle weiteren Individuen werden zufällig aus allen verbliebenen Eltern und Kinder gewählt, bis die Populationsgröße erreicht ist. Die neu entstandene Population ist die Eltern-Population der nächsten Generation. Der Nutzer kann Einfluss auf die Diversität der Population nehmen, indem er die Populationsgröße, Mutationswahrscheinlichkeit und Rekombinationswahrscheinlichkeit variiert. Der Evolutionäre Algorithmus bricht ab, wenn die max. Anzahl von Generationen erreicht ist. Falls der Nutzer das Abbruchkriterium „Geleertes Spielfeld“ angibt, bricht er bereits beim ersten gefundenen Individuum, dass das Spielfeld leert. Die Operationswahl lässt ein vollständiges Konvergieren der Population nicht zu. Die konkrete Implementierung wird im Abschnitt 3 beschrieben.

3 Implementierung

In diesem Abschnitt wird auf entscheidende Implementierungsdetails eingegangen. Dazu werden zunächst die beteiligten Klassen im Abschnitt 3.1 erläutert. Weiterhin werden Operationen näher erklärt, die unmittelbar Bestandteil des Evolutionären Algorithmus sind. Das Programm ist in Ruby geschrieben. Alle Elemente der Programmierung haben englische Bezeichner und werden nach Bedarf beim ersten Erwähnen erklärt. Grundkenntnisse der Programmiersprache Ruby werden hierbei vorausgesetzt.

3.1 Klassen

Die Anwendung ist so in Klassen unterteilt, dass inhaltlich Zusammenhängende Eigenschaften und Funktionen gruppiert werden. Die Klassen sind die Implementierung des in Abschnitt 2.4 beschriebenen Evolutionären Algorithmus. Sie verwenden die im Abschnitt 2.3 beschriebenen Datenstrukturen. Der modulare Aufbau der Anwendung dient der Übersichtlichkeit, erleichtert die Wartbarkeit und bietet individuelle Anpassungsmöglichkeiten. Die unabhängige Verwendung einzelner Klassen ist ebenfalls möglich. Die Anwendung besteht aus 6 Klassen. Sie werden einzeln in den folgenden Abschnitten beschrieben.

3.1.1 Setup

Die Klasse **Setup** beinhaltet alle initialen Daten, die für die Ausführung der Anwendung nötig sind. Dabei prüft die Klasse zuerst, ob die Nutzereingaben das richtige Format haben. Danach werden alle übergebenen Werte verwendbar für die anderen Klassen gespeichert. Die Attribute der Klasse werden im Abschnitt 3.2 näher erklärt. **Setup** hat genau eine Methode, die Initialisierungsmethode *initialize*. Im Programmablauf wird die Instanz der Klasse einmal erzeugt und dient von diesem Zeitpunkt an als Datenobjekt.

3.1.2 Population

Diese Klasse repräsentiert einer Population für den in dieser Arbeit entwickelten Evolutionären Algorithmus. Wird eine Instanz der Klasse erzeugt,

so initialisiert sie zufällige Individuen der Klasse **Individual** je nach vorgegebener Populationsgröße. Individuen einer Population werden im Attribut *individuals* gespeichert. Das Beispiel *Quelltext 1* zeigt die Methoden *best_individual* (bestes Individuum), *next_generation* (nächste Generation) und *mutate!* (mutieren).

```
..
 3 class Population
 4   attr_accessor :individuals
..
24   def best_individual
25     @individuals.sort{|a,b| b.F <=> a.F }.first
26   end
27
..
52   def next_generation
53     res = Marshal.load(Marshal.dump(self))
54     res.recombine!
55     res.mutate!
56     res
57   end
..
70   def mutate!
71     @individuals.each do |individual|
72       if ($setup.mutate_prob == 1.0)||(rand <= $setup.mutate_prob)
73         individual.mutate!
74       end
75     end
76   end
..
```

Quelltext 1: population.rb – Die Klasse Population

In der Zeile 25 wird auf das Attribut *F* von Individuen zugegriffen. Dieses liefert einen numerischen Wert, der der Fitness des entsprechenden Individuums entspricht. Die Methoden *sort* und *first* sind Standardmethoden von Ruby. Ab der Zeile 52 wird die Kind-Population berechnet. Dazu wird die gesamte Population kopiert und auf der Kopie die Methoden *recombine!* und *mutate!* aufgerufen. Entsprechend der Ruby-Konventionen verändern Methoden die mit einem Ausrufungszeichen enden das Objekt auf dem sie arbeiten. Die Implementierung von *mutate!* ist ab der Zeile 70 zu sehen. Mit der Mutationswahrscheinlichkeit von *$setup.mutate_prob* (Gleitkommazahl zwischen 0.0 und 1.0) wird jedes Individuum der Population mutiert. Für den Fall, dass *$setup.mutate_prob* den Standardwert 1 annimmt, wird aus Gründen der Performance die Berechnung einer Zufallszahl eingespart. Jedes Individuum wird dann mutiert. Die Klasse **Population** enthält außerdem die für

den Evolutionären Algorithmus entscheidende Methode *merge*. Sie führt die Umweltselektion wie im Abschnitt 2.4 beschrieben aus.

3.1.3 Individual

Die Klasse **Individual** ist die Repräsentation der Individuen. Die Modellierung und Kodierung von Individuen für den Evolutionären Algorithmus wird im Abschnitt 2.3 beschrieben. Der *Quelltext 2* zeigt einen Ausschnitt aus dieser Klasse.

```
..
 3 class Individual
..
20   def ==(individual)
21     comp_idx = [@max_hit_idx, individual.max_hit_idx].max
22     @G[0..comp_idx] == individual.G[0..comp_idx]
23   end
24
25   def mutate!
..
31     @G = 1.upto(@max_hit_count).map{rand(@max_hit_count)}
32     @clears = false
33     @max_hit_idx = @max_hit_count
34     self
35   end
36
37   def one_point_crossover!(individual, idx = 0)
38     @clears = false
39     @max_hit_idx = @max_hit_count
40     @G = @G[0..idx] + individual.G[(idx+1)..@max_hit_count]
41     self
42   end
43
..
```

Quelltext 2: individual.rb – Die Klasse Individual

Die Methode *mutate!* bewirkt eine zufällige Neuerzeugung des Genotyps des entsprechenden Individuums (Zeile 31). Die Eigenschaften *clears* und *max_hit_idx* werden dabei auf ihren Ursprungswert zurückgesetzt. *max_hit_idx* beschreibt die Anzahl der Züge eine Genotyps, bis ein finaler Zustand erreicht ist. Im Laufe der Entwicklung hat sich herausgestellt, dass insbesondere für Durchläufe mit mehr als 20 Generationen die vollständige Neuerzeugung im Durchschnitt bessere Ergebnisse liefert als eine Punktmutation oder vertauschende Mutation. Die Rekombination in Form eines 1-Punkt-Crossovers ist im *Quelltext 2* (Zeile 37) zu sehen. Der Methode *one_point_crossover!* wird dazu das Individuum übergeben, dessen Genotyp ab

dem übergebenen Index übernommen wird. Der Crossover lässt sich mit Rubys Array-Objekten, wie in der Zeile 40 zu sehen, ausdrücken. Damit Individuen untereinander vergleichbar sind, werden die Methoden <=> und == überladen. Die Implementierung von == ist ab der Zeile 20 zu gezeigt. Zwei Individuen gelten als gleich, wenn ihre Genotypen bis zum größeren der beiden *max_hit_idx* gleich sind. Die Implementierung von <=> ist analog dazu. Gleichheit von Individuen drückt aus, dass sie gleiche Wirkung auf ein gegebenes Spielfeld haben. Alle anderen Eigenschaften des Individuums hängen direkt davon ab. Wie im Abschnitt 3.1.2 beschrieben, wird ein Array von Instanzen der Klasse **Individual** im Attribute *individuals* einer Population gespeichert.

3.1.4 SameGame

Die Klasse **SameGame** enthält alle nötigen Daten über den aktuellen Zustand des Spielfelds. Ihre Hauptaufgabe ist es, die Fitness eines Individuums zu berechnen. Die SameGame-spezifischen Methoden sind dazu in dieser Klasse implementiert. Pro Ausführung der Anwendung wird genau eine Instanz der Klasse erzeugt. Die für den Ablauf des Evolutionären Algorithmus entscheidenden Methoden werden im Quelltext 3 gezeigt.

```
...
  4 class SameGame
  5   attr_accessor :matrix
...
 57   def fitness(&block)
 58     self.class.send(:define_method, 'fitness', &block)
 59   end
...
 68   def fitness_of(individual)
 69     reset!
 70     fit_val = 0
 71     individual.G.each_with_index do |hit_index, idx|
 72       if @history
...
100       else
101         possible_hits = possible_hits_count
102         if possible_hits == 0
103           individual.clears = true if finished?
104           individual.max_hit_idx = idx - 1
105           break
106         end
107         hit_index = (hit_index % possible_hits) + 1
108       end
109       fit_val += fitness(hit(hit_index))
110       individual.clears = true if finished?
```

```
110     end
111     fit_val
112   end
...
240   def hittable?(params)
...
251     @hittable_checked[col][row] = 1
252     @hittable += 1
253
254     return true if @hittable >= @area
255     check_east  = hittable?(params.merge! :col => col+1) if
((col+1) < @col_count) && (!@hittable_checked[col+1][row]) &&
(@matrix[col][row] == @matrix[col+1][row])
256     return true if check_east
257     check_south = hittable?(params.merge! :row => row+1)...
258     return true if check_south
259     check_west  = hittable?(params.merge! :col => col-1)...
260     return true if check_west
261     check_north = hittable?(params.merge! :row => row-1)...
262     return true if check_north
263     return false
264   end
...
277   def mark_involed_positions(col, row, hit_index)
278     @hit_checked[col][row] = hit_index
279     mark_involed_positions(col+1, row, hit_index) if
((col+1) < @col_count) && (!@hit_checked[col+1][row]) &&
(@matrix[col][row] == @matrix[col+1][row])
280     mark_involed_positions(col, row+1, hit_index)...
281     mark_involed_positions(col-1, row, hit_index)...
282     mark_involed_positions(col, row-1, hit_index)...
283     nil
284   end
285
286   def finished?
287     (@matrix[0][0] == 0)
288   end
289
290   def possible_hits_count
291     hit_index = 0
...
293     @matrix.each_with_index do |rows, col|
294       rows.each_with_index do |val, row|
295         if (!@hit_checked[col][row]) && hittable?(:col =>
col, :row => row)
296           mark_involed_positions(col, row, (hit_index+=1))
...
298         end
299       end
300     end
301     hit_index
302   end
...
```

Quelltext 3: same_game.rb – Die Klasse SameGame

Die Methode *hittable?* (Zeilen 240ff.) liefert einen Wahrheitswert, der ausdrückt ob ein Zug auf einer bestimmten Zelle (*col*, *row*) ausgeführt werden kann (*true*) oder nicht (*false*). Die Zeilen 257, 259 und 261 sind analog zur Zeile 255. Sie werden aus Gründen der Übersichtlichkeit abgekürzt. *hittable?* wird solange rekursiv aufgerufen, bis die Summe der betrachteten anliegenden Felder gleicher Farbe die Wertvorgabe aus *area* (*group* = *area*) erreicht hat. Dass ein Zug auf einer bestimmten Zelle zulässig ist, ist in Zeile 295 die Voraussetzung für die Berechnung der Matrix *hit_checked.* Sie beinhaltet nach dem Aufruf der Methode *possible_hits_count* alle möglichen Züge in Form ihrer Indexe an allen Positionen der Zellgruppe, die zum entsprechenden Zug gehören. Pro möglichen Zug wird dafür die Methode *mark_involed_positions* aufgerufen (Zeile 296). Auch an dieser Stelle werden die Zeilen 280-282 verkürzt dargestellt, da sie analog zur Zeile 279 strukturiert sind. Das folgende Beispiel zeigt die Wirkung der Methode *possible_hits_count* auf das Attribut *hit_checked* in Abhängigkeit von *matrix* und *area*.

matrix:	*hit_checked* (*area*=2):	*hit_checked* (*area*=3):
1 1 2 3 1	2 2 F F F	F F F F F
2 3 3 2 2	F 4 4 6 6	F 1 1 3 3
1 3 3 2 2	1 4 4 6 6	F 1 1 3 3
1 2 1 3 2	1 3 5 F 6	F F 2 F 3
3 2 1 1 1	F 3 5 5 5	F F 2 2 2

Der Rückgabewert von *possible_hits_count* ist dabei der größte vorkommende Index der Matrix *hit_checked*. Es sind also 6 Züge möglich, wenn die Zellgruppenmindestgröße 2 ist und nur 3 Züge, wenn sie 3 ist. Zellen, die mit einer Zahl besetzt sind, repräsentieren einen möglichen zulässigen Zug. Alle anderen Zellen sind mit F (*false*) gekennzeichnet. Die Matrix *hit_checked* wird bereits in diesem Schritt berechnet, damit die Methode *hit* im weiteren Programmverlauf effizient ausgeführt wird. Die Implementierung von *hit* wird hier nicht gezeigt. *hit* setzt alle Zellen auf 0, die den übergebenen Zug-Index haben und stellt den korrekten Nachfolgezustand des Spielfelds *matrix* her. Der Rückgabewert von *hit* ist die Anzahl der beteiligten Zellen an diesem Zug. Da die Orientierung des Spielfelds links unten ist, kann mit der Abfrage, ob die Zelle (0, 0) dem Wert 0 entspricht, geprüft werden, ob das Feld geleert wurde. Dies geschieht in der Methode *finished?* (Zeile 286). Die beschriebenen Methoden werden für die Berechnung der Fitness eines Individuums verwendet. Die Methode *fitness_of* (Zeile 68) bekommt ein Individuum als

Argument, dessen Genotyp die Reihenfolge der Zugindexe definiert. Die in der Zeile 91 verwendete Methode *fitness* wird zuvor definiert. *fitness* überschreibt sich beim ersten Aufruf selbst. Eine mögliche Definition zeigt die folgende Quelltextzeile:

```
same_game.fitness{|positions| positions * positions}
```

In diesem Beispiel wird die Punktzahl für einen Zug berechnet, indem die Anzahl aller beteiligten Zellen mit sich selbst multipliziert wird. Die Zeilen 73 bis 99 beschäftigen sich mit dem Fall, dass die **History** aktiviert ist. Darauf wird im Abschnitt 3.1.6 näher eingegangen.

3.1.5 SameGameHelper

Die Klasse **SameGameHelper** besitzt zwei Klassenmethoden. Beide dienen dem Generieren von Spielfeldern für das SameGame. Die Methode *generate_matrix* erzeugt ein zufälliges Spielfeld in Abhängigkeit von den Argumenten *cols* (Spaltenanzahl), *rows* (Zeilenanzahl) und *colors* (Anzahl der max. vorkommenden Farben). Das Spielfeld wird als Matrix (bzw. zweidimensionales Array) zurückgegeben. Die folgende Quelltextzeile zeigt die Generierung einer solcher Matrix:

```
1.upto(cols).map{1.upto(rows).map{rand(colors)+1}}
```

generate_matrix_string funktioniert auf die gleiche Weise, mit dem Unterschied, dass das Ergebnis als Zeichenkette (*String*) zurückgegeben wird. Je nach Nutzereingaben ist die Erzeugung eines zufälligen Spielfelds zu Beginn der Programmausführung nötig. Übergibt der Nutzer ein Spielfeld, so wird die Klasse **SameGameHelper** in diesen Durchlauf nicht verwendet.

3.1.6 History

Die Klasse **History** hat die Aufgabe berechnete Zwischenwerte zu speichern um später im Programmablauf wieder darauf zugreifen zu können. Wie im Abschnitt 3.1.4 erwähnt, wird die vergleichsweise aufwendigen Methoden *fitness_of* der Klasse **SameGame** dadurch beschleunigt. Im Quelltext 4 wird die Verwendung von **History** gezeigt.

```
...
 72  if @history
 73    possible_hits = @history.possible_hits(@matrix)
 74    if !possible_hits.nil? # known from history
...
 80      hit_index = (hit_index % possible_hits) + 1
```

```
 81      hit_checked = @history.hit_checked(@matrix, hit_index)
 82      if !hit_checked.nil? # known from history
 83        @hit_checked = @history.hit_checked(@matrix,
hit_index)
 84      else
 85        possible_hits_count # to set @hit_checked
 86      end
 87    else
 88      possible_hits = possible_hits_count
 89      @history.add_possible_hits :matrix => @matrix,
:possible_hits => possible_hits
...
 97      hit_index = (hit_index % possible_hits) + 1
 98      @history.add_hit_checked :matrix => @matrix,
:hit_index => hit_index ,:hit_checked => @hit_checked
 99    end
100  else
...
```

Quelltext 4: same_game.rb – Verwendung von History

Das Attribut *history* wird bei der Instanziierung der Klasse **SameGame** initialisiert. Zu diesem Zeitpunkt sind keine Daten in *history* vorhanden. Da die Methode *possible_hits* in diesem Fall *nil* liefert, wird der in Zeile 87 beginnende else-Zweig ausgeführt. Die Werte von *possible_hits* und *hit_checked* werden im **History**-Objekt, wie in den Zeilen 89 und 98 zu sehen, gespeichert. Sobald im weiteren Verlauf die Anzahl der möglichen Züge für ein Spielfeld im gleichen Zustand gebraucht wird, wird ausschließlich auf den bereits berechneten Wert zugegriffen. Weitere Aufrufe der Methode *possible_hits_count* werden somit gespart. Für *hit_checked* gilt das gleiche. Die Implementierung der Klasse **History** wird im Qelltext 5 gezeigt.

```
  1 class History
...
 14   def add_possible_hits(params)
...
 17     @history_of_possible_hits[matrix_to_i(matrix)] =
possible_hits
 18     @history_of_possible_hits =
@history_of_possible_hits[1..@history_of_possible_hits.length]
if @history_of_possible_hits.length > @limit
 19     self
 20   end
 21
 22   def possible_hits(matrix)
 23     @history_of_possible_hits[matrix_to_i(matrix)]
 24   end
...
 39   def matrix_to_i(matrix)
 40     if @color_count+1 <= 10
```

```
 41        res =
Integer(matrix.reverse.transpose.reverse.to_s.gsub(/[ ,\
[\]]/,''),(@color_count+1))
 42      else
 43        res = matrix
 44      end
 45      res
 46    end
...
```

Quelltext 5: history.rb – Die Klasse History

Die Variablen *history_of_possible_hits* und *history_of_hit_checked* haben den Typ **Hash**. Mit der Methode *add_possible_hits* (Zeilen 14ff.) wird ein Schlüssel-Wert-Paar in *history_of_possible_hits* eingefügt. Der Schlüssel ist die in eine Zahl transformierte Matrix des Spielfelds. Der Wert ist die Anzahl der möglichen Züge im aktuellen Zustand des Spielfelds. Mit *possible_hits* wird der Wert aus dem Hash ausgelesen und zurück geliefert. Analog dazu funktionieren *add_hit_checked* und *hit_checked*. Deren Quelltext ist hier nicht abgebildet. Der Unterschied zu den *possible_hits*-Methoden ist, dass der Schlüssel aus den beiden Komponenten Spielfeld (*matrix*) und Index des Zugs (*hit_index*) zusammengesetzt ist. Der Rückgabewert von *hit_checked* ist die Matrix, auf der die möglichen Züge durch ihren Index markiert sind. Die Transformation einer Spielfeldmatrix in eine Zahl ist ab Zeile 39 definierte. Dazu transformiert die Methode *matrix_to_i* die gegebene Matrix zunächst so, dass sie in ihrer Zeichenkettendarstellung möglichst viele führende Nullen hat. Danach wird diese Zeichenkette als Zahl der Basis (*color_count* + 1) aufgefasst und schließlich in einen numerischen Wert umgewandelt. Dafür ist entscheidend, dass die Orientierung des Spielfelds, sowie dessen Dimension und die Anzahl der Farben konstant ist. Für ein Spielfeld der Größe 15 mal 15 mit 4 Farben und einer Mindestgruppengröße von 2 werden in der ersten Generation bereits über 1000 Einträge für je *history_of_possible_hits* und *history_of_hit_checked* erzeugt. Um den Arbeitsspeicher effizient zu nutzen, erfolgt deswegen die Transformation mittels *matrix_to_i*. Das folgende Beispiel demonstriert den Sachverhalt (5 Spalten, 5 Zeilen, 4 Farben):

```
A = 2 4 2 4 2     B = 0 0 0 0 0     C = 0 0 0 0 0
    1 2 1 2 2         0 0 0 0 0         0 0 0 0 0
    2 4 2 2 3         2 4 0 2 0         0 0 0 0 0
    3 3 1 1 3         3 3 1 1 0         0 0 0 0 0
    1 4 2 3 2         1 4 2 3 2         1 4 0 0 0
```

A, *B* und *C* benötigen je 64 Bytes Speicher.
matrix_to_i(A): `17380635691068294 6` (17 Bytes)
matrix_to_i(B): `2656776696` (9 Bytes)
matrix_to_i(C): `0` (4 Bytes)

Auf diese Weise wird dafür gesorgt, dass mehr Einträge im vorhandenen Speicher abgelegt werden können als ohne diese Transformation. Damit bei Laufzeiten, in denen der Speicher in denen mehr Speicher gebraucht wird als vorhanden ist, wird ein Limit (*limit*) festgelegt. Sobald die Anzahl der Einträge über diesen Wert hinaus geht, wird das älteste Element aus dem entsprechenden Hash entfernt. Das ist in der Zeile 18 zu sehen. Beim Entwurf wurde darauf geachtet, dass die Funktion *matrix_to_i* eineindeutig ist. Die Umkehrfunktion *restore_matrix* transformiert eine Zahl zurück in die ursprüngliche Matrix. Die Verwendung von **History** ist optional, aber für Durchläufe mit Spielfeldern größer als 10x10 und mehr als 5 Generationen wird die Verwendung empfohlen. Die Untersuchungen zur Ressourcenschonung finden nicht im Rahmen dieser Arbeit statt.

3.2 Programmaufruf

In diesem Abschnitt wird die Verwendung des Programms erklärt. Der Ablauf des Evolutionären Algorithmus wurde bereits im Abschnitt 2.4 beschrieben. Das Programm **samegame** ist eine Anwendung, die auf Computern ausgeführt werden kann, die Ruby unterstützen. Alle vom Nutzer eingegebenen Argumente werden in **Setup**-Objekt gespeichert. Das Programm wird in der Kommandozeile aufgerufen. Möglich Parameter sind:

-c, --clip	Das Spielfeld aus der Windows-Zwischenablage kopieren.
-g, --generate	Ein zufälliges Spielfeld erzeugen
--cols=<colums>	Anzahl der Spalten des zu erzeugenden Spielfelds
--rows=<rows>	Anzahl der Zeilen des zu erzeugenden Spielfelds
--colors=<colors>	Anzahl der Farben des zu erzeugenden Spielfelds
--area=<size>	Gruppengröße
--finished	Bewertungskriterium „Leeren des Spielfelds“
--first	Abbruchkriterium „Leeren des Spielfelds“
--pop=<population>	Populationsgröße

--gen=<generations>	Anzahl der Generationen
--mprob=<percent>	Mutationswahrscheinlichkeit
--rprob=<percent>	Rekombinationswahrscheinlichkeit
-h, --history	History verwenden
-o, --output	Zwischenergebnisse ausgeben
-s, --string	Spielfeldausgabe als Zeichen anstelle von UNIX-farbkodierter Ausgabe
-p, --play	Lösung auf Spielfeld ausgeben
<game_field>	Spielfeld, z.B. [[1,3,2,1],[3,1,1,2],[1,3,2,3],[1,2,3,2]]

Bei der Angabe des Spielfelds ist darauf zu achten, das dies spaltenweise geschieht. Das angegebene Beispiel ist die Kodierung des folgenden Spielfelds:

```
2 2 3 2 2
2 1 2 3 3
3 1 3 2 2
1 3 1 1 1
```

Beispielaufrufe sind:

```
samegame -g -o -p -s
samegame -g -o --gen=30 --pop=10 --cols=20 --rows=15 -h
samegame -o [[1,3,2,1],[3,1,1,2],[1,3,2,3],[1,2,3,2]]
```

Je nach Systemeinstellungen kann es auf Unix-ähnlichen Systemen nötig sein **`./samegame...`** anstelle von **`samegame...`** aufzurufen. Weiterhin steht die Windows-Zwischenablage nicht zur Verfügung. In einer Windows-Umgebung kann es nötig sein **`ruby samegame...`** zu verwenden. Da Windows die farbkodierte Ausgabe nicht interpretieren kann, sollte stets die Option **`-s`** verwendet werden.

4 Untersuchungen

In diesem Abschnitt werden Messungen diskutiert, die den implementierten Evolutionären Algorithmus charakterisieren. Es wird auf einem gegebenen 10x10 Zellen großes Spielfeld untersucht, wie sich die Mutationswahrscheinlichkeit und Rekombinationswahrscheinlichkeit auf die Güteentwicklung bester Individuen auswirken. Damit Messungen mit „Ausreißern“ nicht ins Gewicht fallen, wird jede Messung 10-fach mit den gleichen Parametern ausgeführt. Die Ergebnisse werden in den Abbildungen 1 bis 9 gezeigt. Die Matrix des untersuchten Spielfelds ist:

```
M = 1 3 1 1 3 3 1 3 3 1
    3 2 1 3 2 2 3 3 2 2
    3 2 1 1 2 1 3 3 3 1
    2 1 2 3 2 1 2 1 2 2
    3 3 1 2 3 3 1 3 1 3
    1 3 3 3 1 1 2 1 1 3
    1 2 3 2 3 3 2 1 2 2
```

Jede Messung wird mit einer Population von 10 Individuen durchgeführt. Für alle Versuche werden 20 Generationen berechnet. Es werden verschiedene Kombinationen von Mutationswahrscheinlichkeit (*mprob*) und Rekombinationswahrscheinlichkeit (*rprob*) verwendet (Angaben in Prozent):

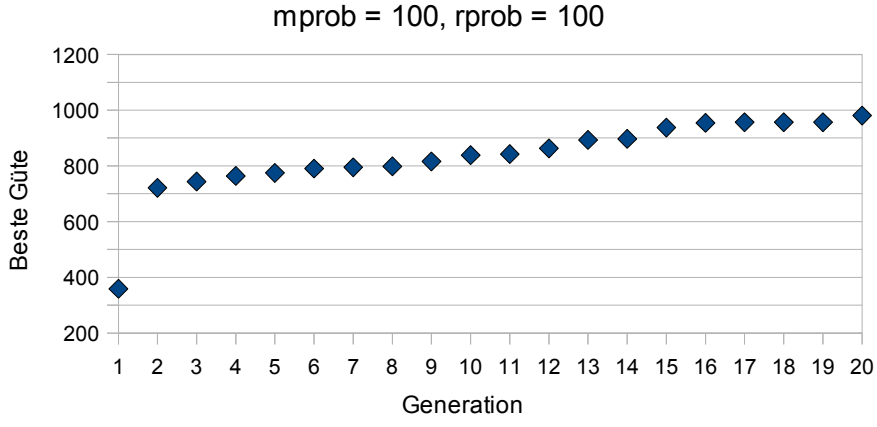

Abbildung 1: Beste Güte bei mprob=100 und rprob=100

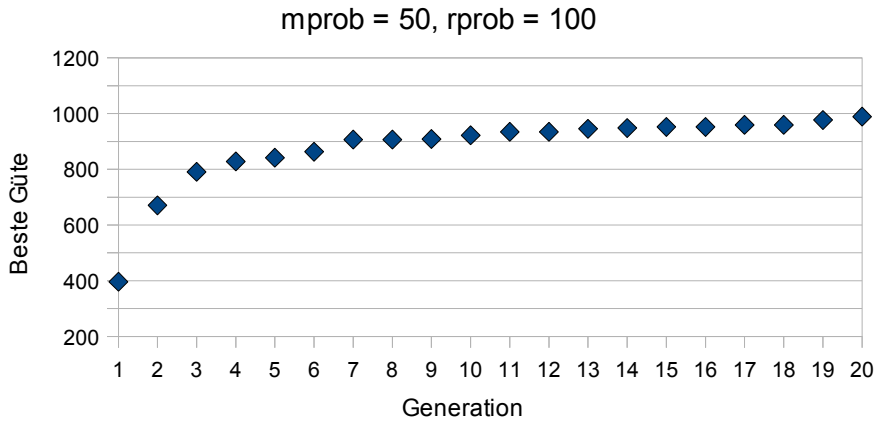

Abbildung 2: Beste Güte bei mprob=50 und rprob=100

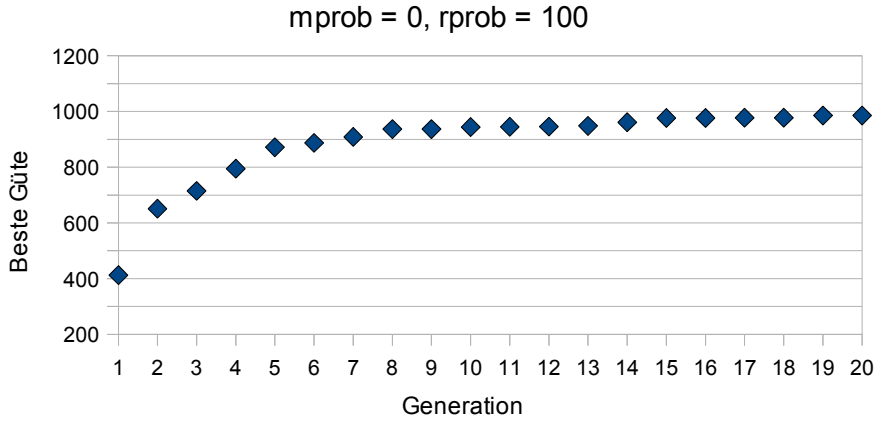

Abbildung 3: Beste Güte bei mprob=0 und rprob=100

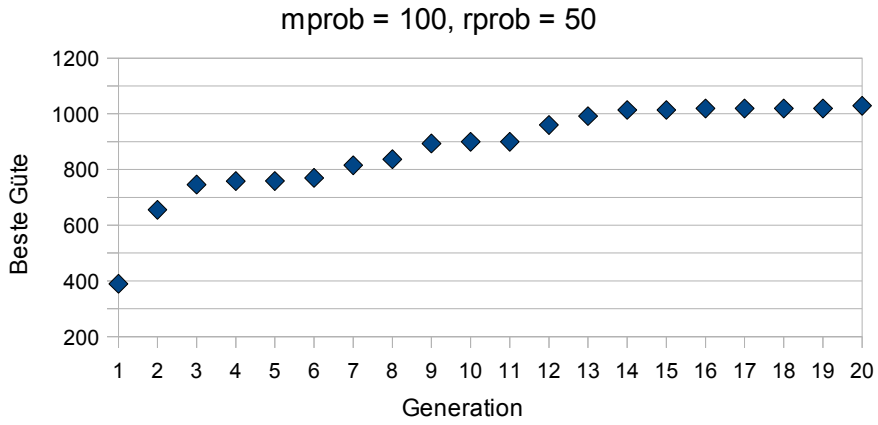

Abbildung 4: Beste Güte bei mprob=100 und rprob=50

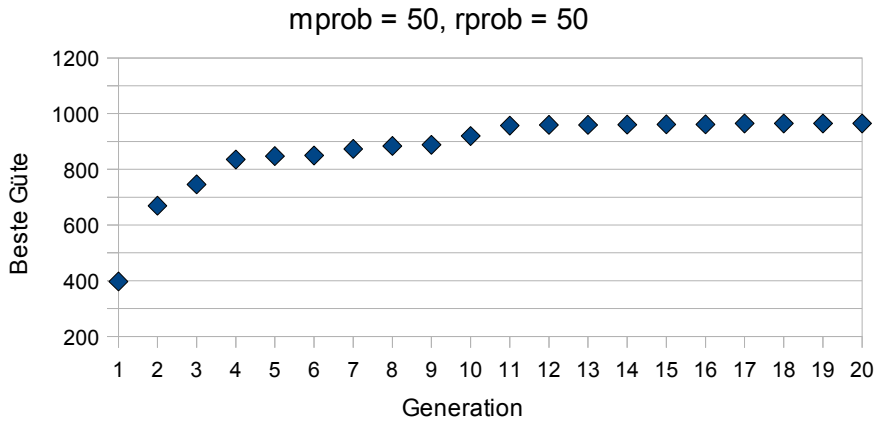

Abbildung 5: Beste Güte bei mprob=50 und rprob=50

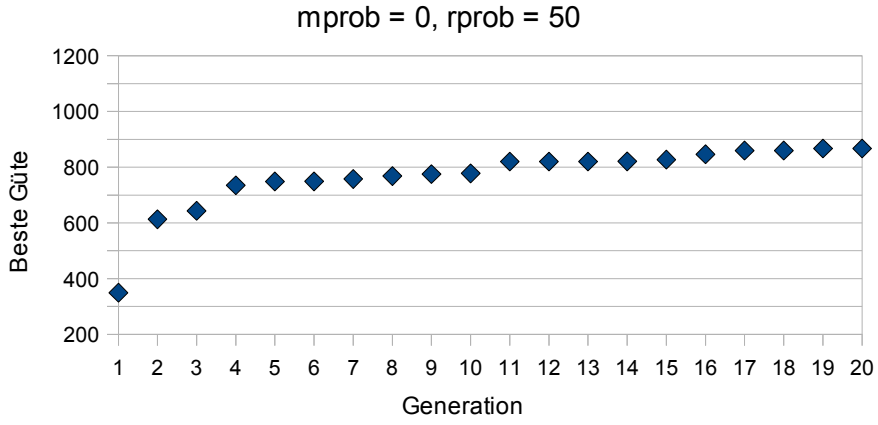

Abbildung 6: Beste Güte bei mprob=0 und rprob=50

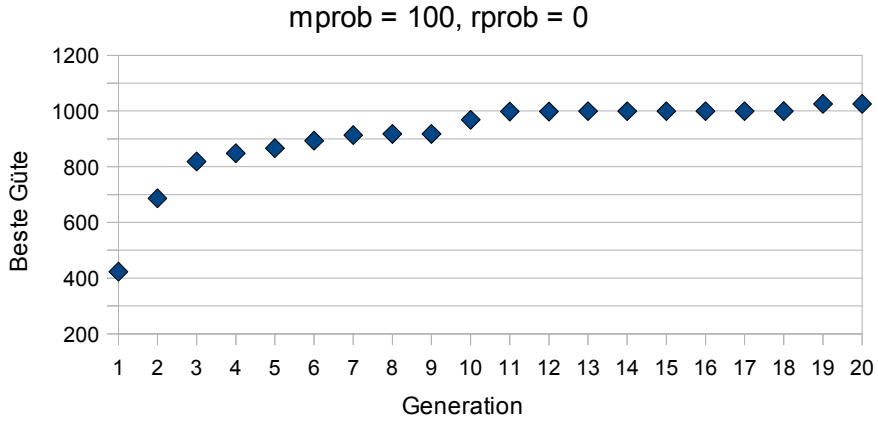

Abbildung 7: Beste Güte bei mprob=100 und rprob=0

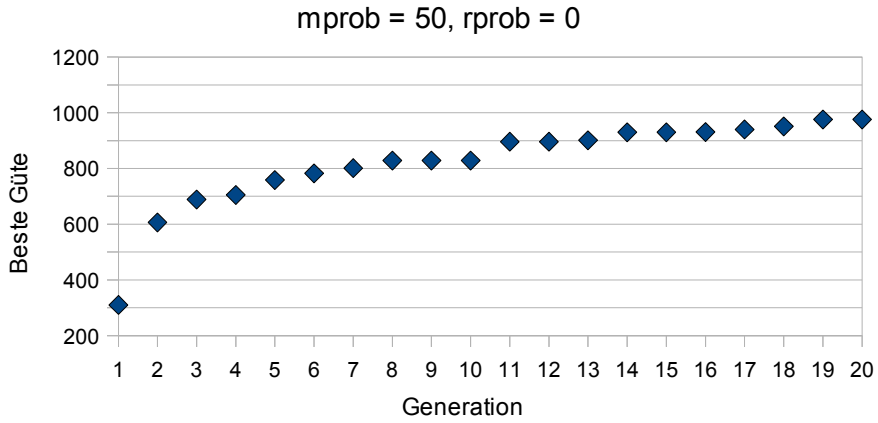

Abbildung 8: Beste Güte bei mprob=50 und rprob=0

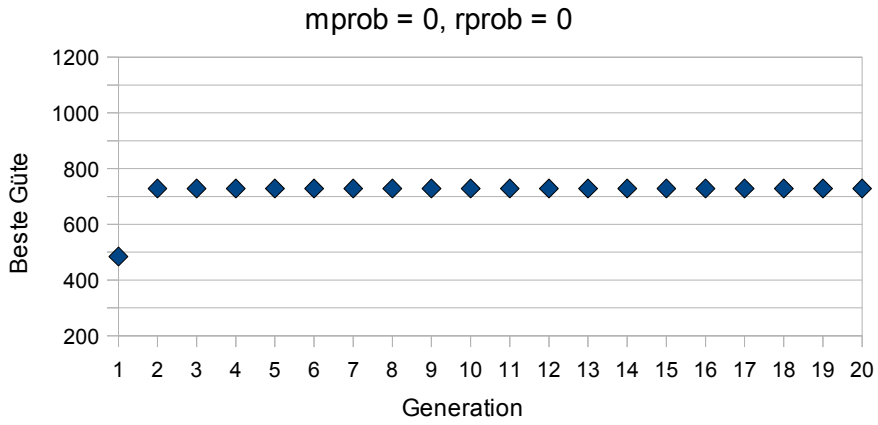

Abbildung 9: Beste Güte bei mprob=0 und rprob=0

Die Abbildungen 1 bis 9 zeigen die beschriebenen Untersuchungsergebnisse. Bei der Interpretation der Graphen ist zu beachten, dass sich die beste Güte auf die vorhergehende Generation bezieht. Der bei Generation 1 eingetragene Wert ist ein Zufallswert. Alle weiteren bilden die Güte des besten Individuums der vorigen Generation ab. Die Durchläufe mit einer hohen Mutations-wahrscheinlichkeit haben im Durchschnitt bessere Ergebnisse nach 20 Generationen als die mit niedrigeren. Die Rekombinationswahrscheinlichkeit hat einen Einfluss auf die Form des Graphen. Je kleiner die Rekombinationswahrscheinlichkeit ist, desto größer und seltener sind die Sprünge die eine Verbesserung der besten Güte repräsentieren. Dieses

Verhalten ergibt sich aus der Mutationsoperation, die einen völlig neuen Genotyp erzeugt. Rekombination ohne Mutation führt bei 10 Individuen zu ähnlich guten Resultaten wie etwa eine Verteilung *mprob*=50 und rprob=50. Untersuchungen mit 2 bis 5 Individuen bestätigen, dass reine Rekombination früher in einem lokalen Maximum endet als Untersuchungen mit Mutation. Ohne Mutation oder Rekombination bleibt die beste Güte von Beginn an konstant.

5 Ausblick

Im Vergleich zu einem brute-force-Ansatz bietet der hier beschriebene Evolutionäre Algorithmus Möglichkeit, Lösungen für ein SameGame zu finden. Das Entscheidungsproblem „Leeren des Spielfelds“ kann jedoch von einem Algorithmus, der von Zufallsgrößen abhängt, nicht vollständig gelöst werden. Die positive Aussage lässt sich einerseits treffen, sobald ein Individuum das Spielfeld leer. Wird keine derartige Lösung gefunden, lässt sich andererseits keine zuverlässige Aussage über die Lösbarkeit des Problems machen. Um dieses Problem vollständig zu lösen empfiehlt sich der Ansatz einen Evolutionären Algorithmus nicht. Ähnlich verhält es sich mit der Aussage über das globale Maximum. Ob ein globales Maximum gefunden wurde, lässt sich also nicht sagen. Es ist außerdem nicht ausgeschlossen, dass es mehrere globale Maxima gibt. Aussagen über die Qualität der vorgestellten Mutationsoperation und Rekombinationsoperation gilt es in weiteren Arbeiten zu machen. Der gewählte Genotyp ist universell einsetzbar für die meisten Implementierungen des SameGames, stellt aber einen Nachteil für das Laufzeitverhalten des Evolutionären Algorithmus dar. Auch der Ansatz, der mit History verfolgt wird, lässt sich weiter ausbauen. So können beste Lösungen oder Teile von besten Lösungen wiederverwendet werden. Dazu wird die History in einen persistenten Datenspeicher ausgelagert. Die entwickelte Software ist zudem nur beschränkt praxistauglich, wenn es um die Automatisierung von SameGame-Spielen geht. Die Entwicklung von Screenparser und Genotypinterpreter werden nötig, um eine Automatisierung zu realisieren. Auch Performance-orientierte Änderungen auf Kosten den Universalität sind zu untersuchen.

6 Verzeichnisse

Literaturverzeichnis

[BIE02]: Therese C. Bied et al. (2002), The Complexity of Clickomania, More Games of No Chance, http://library.msri.org/books/Book42/files/biedl.pdf

Abbildungsverzeichnis

Quelltextverzeichnis

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

Leipzig, den 18.02.2011

_____

Wolfgang Teuber, B. Sc.