„Optimaler Sitzplan im Klassenraum auf Basis evolutionärer Algorithmen“

**Studienarbeit II**

für die Prüfung zum

Bachelor of Science

der Angewandten Informatik

an der Dualen Hochschule Baden-Württemberg - Karlsruhe

von

Michael Sprauer

September 2013

Bearbeitungszeitraum 6. Semester

Matrikelnummer, Kurs 5147809, TAI10B2

Ausbildungsfirma Thales Defence & Security Systems GmbH, Pforzheim

Betreuer Prof. Dr. H. Braun

1. Erklärung

gemäß § 5 (2) der „Studien- und Prüfungsordnung DHBW Technik“ vom 18. Mai 2009:

Ich habe die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Karlsruhe, Mai 2013

MICHAEL SPRAUER

Abstrakt

In der Schule müssen Lehrer regelmäßig einen Sitzplan erstellen, um in der Klasse eine optimale Lernatmosphäre zu gewährleisten. Dabei sind viele – zum Teil widersprüchliche – Anforderungen zu berücksichtigen. Beispielsweise müssen einige Schüler in der ersten Reihe sitzen, da sie eine Sehschwäche haben oder besonderer Betreuung durch den Lehrkörper bedürfen. Außerdem sollen nur die Schüler nebeneinander sitzen, die sich nicht vom Lernen abhalten. Besser wären Sitznachbarn, die das Lernen fördern. Damit insgesamt das Lernen in der Klasse optimal läuft muss auch der Geräuschpegel niedrig sein. Wer sich also zu gut versteht darf auch nicht nebeneinander sitzen. Diese Herausforderungen implizieren bereits, dass es keinen perfekten Sitzplan gibt. Diese Studienarbeit verwendet evolutionäre Algorithmen, um den optimalen Sitzplan mit den gegebenen Einschränkungen und Parametern zu finden.

Abstract

Creating a seating plan is a conventional task for every teacher. A well-defined seating plan is mandatory for an effective learning atmosphere in every class room. The requirements are partly conflicting. For example some of the students have to sit in the front row, because they have debility of sight or need special mentoring from the teacher. Other students can’t be placed next to each other, because they distract each other. In the best case the neighbours support each other. To facilitate the best possible learning environment the noise level should be very low. Chatting students shouldn’t sit side by side. All these challenges already imply that there is no such thing like a perfect seating plan. This thesis uses genetic algorithms to find the best possible solution considering all the given constraints and parameters.

1. Inhaltsverzeichnis

[1 Einleitung und Motivation 1](#_Toc356415598)

[1.1 Zielsetzung 1](#_Toc356415599)

[1.2 Wirtschaftliche Aspekte 1](#_Toc356415600)

[1.3 Gliederung der Arbeit 2](#_Toc356415601)

[2 Grundlagen: Evolutionäre Algorithmen 3](#_Toc356415602)

[2.1 Repräsentation 4](#_Toc356415603)

[2.2 Zielfunktion / Fitnessfunktion 5](#_Toc356415604)

[2.3 Greedy 6](#_Toc356415605)

[2.4 Mutationsoperatoren 7](#_Toc356415606)

[2.4.1 Rekombination 7](#_Toc356415607)

[2.4.2 Mutation 8](#_Toc356415608)

[2.5 Selektion 8](#_Toc356415609)

[2.6 Strategie 9](#_Toc356415610)

[2.7 Haltebedingung 10](#_Toc356415611)

[3 Implementierung 11](#_Toc356415612)

[3.1 Entwicklungsumgebung 11](#_Toc356415613)

[3.2 Software-Architektur 13](#_Toc356415614)

[3.3 Codeausschnitte 14](#_Toc356415615)

[3.3.1 Bewertungsfunktion 15](#_Toc356415616)

[3.3.2 Die Bewertungsfunktion 15](#_Toc356415617)

[3.3.3 Selektion 15](#_Toc356415618)

[3.4 Codequalität 15](#_Toc356415619)

[3.5 Arbeitsaufwand 16](#_Toc356415620)

[4 Bedienung 17](#_Toc356415621)

[4.1 Einstellungen 17](#_Toc356415622)

[4.2 Klassen erzeugen, laden und speichern 18](#_Toc356415623)

[4.3 Konfigurationen erzeugen, laden und speichern 19](#_Toc356415624)

[4.3.1 Generation 20](#_Toc356415625)

[4.3.2 Mutation 21](#_Toc356415626)

[4.3.3 Gewichtung 21](#_Toc356415627)

[4.4 Klassenraum 21](#_Toc356415628)

[4.4.1 Relationen der Schüler 21](#_Toc356415629)

[4.4.2 Relation zur Tafel 23](#_Toc356415630)

[4.5 Fitness-Funktion 23](#_Toc356415631)

[5 Evaluation 24](#_Toc356415632)

[5.1 Mutation durch Swap 24](#_Toc356415633)

[5.2 Mutation durch Inversion 26](#_Toc356415634)

[5.3 Auswahl der Mutationsoperatoren 27](#_Toc356415635)

[5.4 Strategien 31](#_Toc356415636)

[5.5 Selektion 31](#_Toc356415637)

[6 Zusammenfassung und Ausblick 32](#_Toc356415638)

[6.1 Verbesserungsmöglichkeiten 32](#_Toc356415639)

[6.2 Fazit 33](#_Toc356415640)

# Einleitung und Motivation

In der Schule müssen Lehrer regelmäßig einen Sitzplan erstellen, um in der Klasse eine optimale Lernatmosphäre zu gewährleisten. Dabei sind viele – zum Teil widersprüchliche – Anforderungen zu berücksichtigen. Beispielsweise müssen einige Schüler in der Ersten Reihe sitzen, da sie eine Sehschwäche haben oder besonderer Betreuung durch den Lehrkörper bedürfen. Außerdem sollen nur die Schüler nebeneinander sitzen, die sich nicht vom Lernen abhalten. Noch besser sind Sitznachbarn, die das Lernen fördern. Damit insgesamt das Lernen in der Klasse optimal läuft muss auch der Geräuschpegel niedrig sein. Wer sich also zu gut versteht darf auch nicht nebeneinander sitzen. Diese Herausforderungen implizieren bereits, dass es keinen perfekten Sitzplan gibt. Einen brauchbaren Plan zu erstellen, in dem die möglichst viele Aspekte berücksichtigt werden erfordert einen erheblichen Zeitaufwand.

## Zielsetzung

Ziel der Arbeit ist es einen optimalen Sitzplan zu finden, der die gegebenen Rahmenbedingungen und Parameter berücksichtigt. Dabei werden die folgenden Kriterien Berücksichtigt:

* Entfernung zur Tafel
* Beziehung zum direkten Nachbar
* Beziehung zum übernächsten Nachbar
* Beziehung zum Vorder-/Hintermann
* Beziehung zu den Schülern, die diagonal sitzen
* Schüler, die ihre Position nicht verändern dürfen

## Wirtschaftliche Aspekte

Die Zeit zur Erstellung eines Sitzplans bedeutet für den Lehrer in der Regel einen Aufwand von mindestens einer Stunde. Je nach Verhalten der Klasse oder Eignung des Sitzplans ist dieser Vorgang mehrmals pro Schuljahr notwendig. Diese Zeit könnte effektiver zur besseren Vorbereitung des Unterrichts verwendet werden.

## Gliederung der Arbeit

Diese Arbeit ist grob in zwei Teile unterteilt. Das Kapitel 2 Beschäftigt sich mit den theoretischen Grundlagen des evolutionären Algorithmus. Dann folgt in Kapitel 3 die Beschreibung der konkreten Realisierung der gestellten Aufgabe. In Kapitel 4 wird die Bedienung der Anwendung erläutert. Abschließend fasst Kapitel 5 die erzielten Ergebnisse zusammen und gibt einen Ausblick auf Untersuchungsmöglichkeiten um die Ergebnisse weiter zu verbessern.

# Grundlagen: Evolutionäre Algorithmen

Dieses Kapitel soll die Grundlagen der Studienarbeit erläutern. Als Verfahren zur Lösung des gestellten Problems eignen sich die evolutionären Algorithmen besonders. Dieses Verfahren wurde zwischen 1970 und 1980 von verschiedenen Wissenschaftlern unabhängig voneinander als effiziente Weise der Optimierung von numerischen Problemen erkannt. In Deutschland wurde das Thema durch Schwefel [1] und Rechenberg [2] unter dem Namen „*Evolutionsstrategie*“ bekannt. Auf der anderen Seite des Atlantiks veröffentlichten Goldberg und Holland [3] ähnliche Gedanken unter dem Titel „*genetische Algorithmen*“. Insbesondere bei nicht linearen Optimierungsproblemen muss man Abwägen zwischen Rechenaufwand und Nutzen.

Die evolutionären Algorithmen bilden die Evolutionstheorie Darwins nach. Dort unterscheidet man zwischen Genotyp und Phänotyp. Der Phänotyp entspricht dabei dem konkreten Exemplar einer Spezies. Der Genotyp ist die genetisch kodierte Information über alle Merkmale und Eigenschaften dieser Art. Der Genotyp mutiert durch äußere Einflüsse bzw. wird durch die Fortpflanzung rekombiniert. Der neu Entstandene Genotyp bzw. der dazugehörige Phänotyp wird nun einem Fitnesstest unterzogen. Überlebt dieser Genotyp bzw. kann mehr Nachkommen erzeugen als andere Genotypen, spricht dies für seine Fitness. Damit werden nur die Vererbungslinien weiter verfolgt, die eine bessere Fitness haben als Andere.

Diese Überlegung wird nun auf das gestellte Problem übertragen. Der Genotyp wird im Folgenden auch als Repräsentation bezeichnet und in Kapitel 2.1 genauer beschrieben. Der Fitnesstest wird durch die Fitnessfunktion bzw. Zielfunktion durchgeführt (Kapitel 2.2). Darauf folg die Erläuterung der initialen Erstellung des ersten Genotyps durch eine sogenannte Greedy-Funktion in Kapitel 2.3. Das Kapitel 2.4 beschreiben die Mutation und Reproduktion. Selektion wird im Kapitel 2.5 behandelt und im Kapitel 2.6 geht es um die Strategie zur Auswahl der Eltern. Abschließend beschreibt der letzte Teil 2.7 das Ende dieses iterativen Prozesses. Alle Abschnitte in diesem Kapitel 2 betrachten die Evolutionären Algorithmen immer im Kontext der gestellten Aufgabe.

## Repräsentation

Die Repräsentation bezeichnet die Kodierung eines konkreten Sitzplans. Das bedeutet, dass in dieser Kodierung alle Informationen über den Sitzplan enthalten sein müssen. Auf diese Repräsentation wird auch die Fitnessfunktion angewendet. Die folgenden Elemente müssen aus der Darstellung hervorgehen:

* Auf welche Position im Klassenraum bezieht sich die Angabe
* Welcher Schüler/Student ist gemeint

Damit ergibt sich für einen rechteckigen Klassenraum mit Tischen die folgende Repräsentation – in diesem Fall eine Permutationscodierung:

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |

Formel 1: Permutationscodierung - ist der Schüler an der Stelle (i,i)

In der Permutationscodierung kommt jeder Schüler genau einmal vor. Bei unterbesetzten Klassenräumen kann es jedoch vorkommen, dass Plätze zwischen den Schülern unbesetzt bleiben können. Diese Eigenschaft kann man sich unter Umständen zu Nutze machen, indem Puffer zwischen Schülern eingebaut werden, um die Fitness zu verbessern. In der Regel werden die freien Plätze durch den Algorithmus jedoch eher in der hinteren Reihe angeordnet.

## Zielfunktion / Fitnessfunktion

Die Zielfunktion beschreibt die Fitness einer gegebenen Repräsentation, gibt also eine Bewertung über die Qualität ab. Man kann sich die Funktion als eine mehrdimensionale Trajektorie vorstellen, die einen Lösungsraum aufspannt. Die verschiedenen Lösungen werden durch die Dimensionen definiert und der Wert an dieser Stelle entspricht der Fitness. Die Zielfunktion besteht aus zwei Teilen, die jeweils gewichtet und summiert werden:

* Abstand zur Tafel
* Abstand zu den Nachbarn

In der folgenden Abbildung 1 sind die Gewichtungsfaktoren jeweils unterschiedlich eingefärbt. Für die Realität ist es – wie in der Bewertung – nicht relevant, ob ein Schüler links oder rechts neben einem anderen Schüler sitzt. Ebenso werden die Schüler vorn und hinter dem Schüler, diagonal zu dem Schüler und zwei Plätze neben dem aktuellen Schüler gleich gewichtet.

**Schüler**

**zwei neben**

**direkt neben**

**zwei neben**

**direkt neben**

**diagonal**

**diagonal**

**diagonal**

**diagonal**

**vor**

**hinter**

**Tafelpräferenz**

Abbildung 1: Gewichtung der Umgebung eines Schülers

- Jeder Farbe kann ein Gewichtungsfaktor zugewiesen werden. Schüler, die direkt nebeneinander sitzen, egal ob rechts oder links, haben den gleichen Gewichtungsfaktor.

Die übrigen Schüler werden zur Bestimmung der Fitness und damit für die Qualität des Sitzplans nicht berücksichtigt.

Der optimale Chromosomensatz kann entweder eine möglichst kleine oder eine möglichst große Fitness haben, abhängig davon wie man das Problem formuliert bzw. die Fitnessfunktion definiert. Hier ist die Fitnessfunktion ist so definiert, dass der beste Sitzplan den nominell höchsten Wert hat. Die Beziehungen werden addiert und mit dem entsprechenden Faktor aus Abbildung 1 multipliziert. Damit ergibt sich eine umso bessere Fitness je mehr positive Beziehungen sich nebeneinander befinden.

## Greedy

Um eine Mutation zu erzeugen oder zwei Gene zu rekombinieren muss zunächst ein initialer Genstring erzeugt werden. Diese Aufgabe wird – in Anlehnung an das Knappsackproblem - auch *Greedy* genannt. Bei dem Knappsack müssen viele Elemente unterschiedlicher Größe in einen begrenzten Speicher (dem knappen Sack) so einsortiert werden, dass möglichst viele hinein passen. Dabei fängt man üblicherweise mit dem größten Element an. Wenn man den Gedankengang auf das vorliegende Problem überträgt, dann erhält man gleich eine recht gute Lösung. Der Algorithmus dazu besteht aus drei Schritten:

1. Alle Schüler, die einen festen Sitzplatz zugewiesen bekommen haben, werden im Raum platziert.
2. Die übrigen Schüler werden nach ihrer Tafelpräferenz sortiert.
3. Der Reihe nach werden die Schüler auf die übrigen Plätze verteilt beginnend vorne links mit dem Schüler mit der höchsten Tafelpräferenz. Bei Schülern mit der gleichen Tafelpräferenz wird zufällig ein Schüler ausgewählt.

Für diesen Genstring wird nun die Fitness bestimmt.

## Mutationsoperatoren

Um nun von diesem initialen Gen zu einem besseren Gen zu kommen, im Sinne der Fitnessfunktion (siehe 0) muss das Gen verändert werden – also Mutation – oder es müssen zwei Elemente ein Neues bilden – also Rekombination. Diese beiden Verfahren sollen nun kurz Erläutert werden.

### Rekombination

Bei der Rekombination kommen zwei mehr oder weniger zufällig ausgewählte Gene zusammen und bilden gemeinsam ein neues Gen. Dabei gibt es verschiedene Varianten, nach denen der neue Genstring gebildet werden kann:

1. Uniform Crossover [4]
2. One Point Crossover
3. Multipoint Crossover [5]
4. Partially Maped Crossover (PMX)
5. Order Crossover
6. Cycle Crossover [6]

Je nach ausgewählter Variante weist das neue Gen mehr oder weniger Ähnlichkeiten zu seinen beiden Eltern auf, kann also hinsichtlich der Bewertung besser oder schlechter sein. Eine neue Bewertung durch die Fitnessfunktion muss also erfolgen. Diese Rekombinationsoperatoren sind sehr anfällig dafür nicht überlebensfähige Nachkommen zu bilden und wurden daher in der aktuellen Version nicht implementiert. Die Nachkommen werden ausschließlich über Mutation erzeugt.

### Mutation

Mutation ist eine Operation die auf ein einzelnes Gen wirkt. Auch hier können verschiedene Varianten eingesetzt werden.

1. Swap
2. Inversion
3. Insert
4. Splice [7]

An dieser Stelle werden nur die ersten beiden Varianten erläutert, die in diesem Projekt auch eingesetzt wurden.

Der **Swap**-Operator wählt zufällig zwei Schüler aus und vertauscht ihre Sitztposition. Die neue Repräsentation unterscheidet sich nicht so stark von der vorherigen. Diese Eigenschaft ist wichtig, um sich der optimalen Sitzordnung langsam zu nähren und nicht immer wieder am Ziel vorbei zu schießen.

Der **Invert**-Operator vertauscht die Reihenfolge einer ganzen Sitzreihe. Dabei verändert sich in der Mitte der Reihe nur wenig, während die äußeren Sitzplätze sehr starken Änderungen unterworfen sind. Das hat zur Folge, dass die neue Bewertung sehr stark von der vorherigen abweichen kann. Für eine effiziente Annäherung an die optimale Sitzordnung sollten nicht zu viele Nachkommen durch Invertierung erzeugt werden, da hier sehr starke Schwankungen zu erwarten sind.

## Selektion

Die so erzeugten Nachkommen werden in den Genpool der aktuellen Generation aufgenommen. Dort werden die Gene miteinander verglichen. Um die Qualität der Nachkommen weiter zu verbessern dürfen nicht alle Elemente im Genpool auch Nachkommen bzw. gleich viele Nachkommen erzeugen. Damit werden Fehlentwicklungen verhindert. Dieser Auswahlprozess wird Selektion genannt. Zur Selektion stehen ebenfalls verschiedene Möglichkeiten zur Verfügung, die nun kurz vorgestellt werden.

1. Nur die n Besten (Best-N)
2. Nach Rang (Rank-Based)
3. Wettbewerb (Tournament)

Die erste Methode (**Best-N**) ist auch gleichzeitig die Einfachste. Hier werden alle Nachkommen nach Absteigender Fitness sortiert und die N Besten daraus zur Mutation ausgewählt.

Auch bei der **Rank-Based**-Selection werden die Nachkommen sortiert und die N besten ausgewählt. Im Unterschied zur Best-n-Selection darf aber nicht jeder gleich viele Nachkommen erzeugen. Die Erzeugung der Nachkommen erfolgt absteigend. Der Beste darf die meisten Nachkommen erzeugen. Der Zweitbeste einen weniger, und so weiter. Je nachdem wie viele Nachkommen erzeugt werden sollen, kann es so passieren, dass die letzten Eltern der N Besten keine Mutationen bilden dürfen.

Die Methode **Tournament** wählt aus allen Elementen zufällig eine Bestimmte Anzahl aus. Davon darf sich dann das beste Element Reproduzieren. Damit kann man recht zuverlässig lokale Optima verlassen, allerdings kommt man nicht so leicht in ein lokales oder globales Optimum. Die Fitness ist am Ende also nicht so gut, wie sie sein könnte.

## Strategie

Nun stellt sich noch die Frage welche Chromosomen für eine Mutation herangezogen können. Ausgehend von dem ersten Element, das durch den Greedy-Algorithmus (Kapitel 2.3) erzeugt wurde, werden anschließend die vorgegebene Anzahl Nachkommen mutiert (Kapitel 2.4) und daraus die Eltern selektiert (Kapitel 2.5). Für den Genpool der nächsten Generation können nun zwei Mengen gebildet werden:

1. Nachkommen mit (+) ihren Elterngeneration
2. Nachkommen ohne (,) ihre Elterngeneration

Die Nachkommen werden in der Regel mit gekennzeichnet und die Elterngeneration mit . Üblicherweise ist die Schreibweise folgende:

Für die nächste Mutationsphase steht jetzt eine neue Menge zur Verfügung, die ebenfalls die Eltern enthalten kann. Diese beiden Optionen wurden auch im Programm implementiert.

Wenn man die Eltern nicht mit in die neue Generation aufnimmt, dann besteht die Möglichkeit, dass alle mutierten Nachkommen nicht besser sind als ihre Eltern und damit die neue Generation insgesamt schlechter abschneidet als die Vorherige.

## Haltebedingung

Zum Schluss muss auch noch eine Abbruchbedingung formuliert werden. Während es bei einer möglichst kleinen Fitness sogar die Möglichkeit gibt, den perfekten Wert „0“ zu erreichen, ist diese Grenze bei steigender Fitness nicht gegeben. Das führt zu einer potentiell endlosen Schleife, in der immer weiter versucht wird ein noch besseres Ergebnis zu erzielen. Man muss also nach einer definierten Anzahl von Generationen abbrechen.

# Implementierung

Zur Implementierung wurde Java verwendet, da es heute eine sehr große Verbreitung hat, durch viele Werkzeuge unterstützt wird und sich gut zur Umsetzung dieser Aufgabe eignet. Die Entwicklungsumgebung dafür wird im Kapitel 3.1 beschrieben. Anschließend folgt die Beschreibung der Softwarearchitektur (Kapitel 3.2) und es werden einige interessante Abschnitte aus dem Quellcode herausgenommen und besprochen (Kapitel 3.3).

## Entwicklungsumgebung

Als IDE (Integrated Development Environment) kommt Eclipse zum Einsatz. Eclipse lässt sich durch Plugins erweitern und an die individuellen Bedürfnisse anpassen. Für dieses Projekt kamen die folgenden Plugins zum Einsatz:

|  |  |
| --- | --- |
| **EclEmma** Java Code Coverage | Zeigt die Testabdeckung direkt in Eclipse an. So lässt sich einfach erkennen, welche Stellen von Tests abgedeckt sind und wo der Code evtl. noch nicht die erforderliche Codequalität aufweist. |
| **Maven** Integration for Eclipse | Integration von dem Application-Lifecycle-Management-Tool Maven. Mit Maven können Abhängigkeiten verwaltet werden und der Buildprozess kann parametrisiert und automatisiert werden. |
| **Sonar** | Sonar zeigt verschiedene Metriken zur Codequalität direkt in der IDE an. Dadurch werden Probleme an ihrer Quelle sichtbar und können umgehend behoben werden. |
| **EGit** | Mit EGit kann das Versionierungswerkzeug git direkt in Eclipse eingebunden werden. Geänderte Elemente werden angezeigt und lassen sich committen oder rückgängig machen. Außerdem kann man sich die Historie einer Datei anzeigen lassen uvm. |
| **WindowBuilder** | Der WindowBuilder erleichtert die Erstellung von grafischen Oberflächen mit Swing und SWT. In dem WYSIWYG-Editor können die grafischen Elemente direkt auf einer Form platziert und parametrisiert werden. |

Tabelle 1: Verwendete Eclipse Plugins

Neben diese Plugins wurden auch zwei Bibliotheken verwendet:

* **JUnit** als Unit-Testing-Framework
* **JFreeChart** zur Darstellung des Diagramms

Zur Nachverfolgung der Änderungen im Quelltext und an der Thesis wird git eingesetzt. Der folgende Screenshot (Abbildung 2) zeigt den Verlauf der Entwicklung. Jede Änderung wird in einem sogenannten Commit zusammengefasst und mit einer Bezeichnung versehen. So lässt sich der Projektfortschritt leicht erkennen und fehlerhafte Änderungen können Rückgängig gemacht werden. Außerdem dient diese Methode als Backup für korrumpierte Dateien oder fehlerhafte Änderungen.

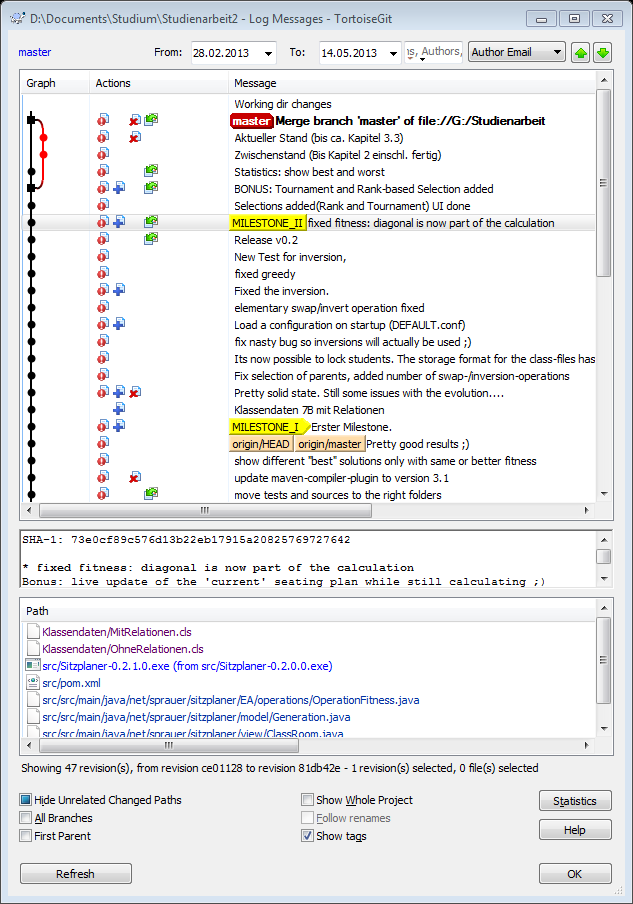


Abbildung 2: Git Versionslog

- Die Änderungen werden zusammengefasst und mit einer kurzen Beschreibung in die Historie eingefügt.

Die gelben Markierungen sind Tags. Dadurch werden Milestones in dem Projekt markiert, um auf ältere Auslieferungsversionen zurück zu kommen und dort ggf. Fehler zu reproduzieren.

## Software-Architektur

Als grobe, grundlegende Architektur kommt für dieses Projekt das MVC-Muster zum Einsatz.

Abbildung 3: MVC-Muster

- Dieses Muster ist eine übliche Architektur für Applikationen. Datenhaltung, Geschäftslogik und Darstellung sind in eigenen Modulen.

Das Model besteht aus einer In-Memory-Datenbank, die eine Anzahl von Studenten und die Abmessungen des Klassenraums, sowie weitere Metadaten speichert.

## Codeausschnitte

In diesem Kapitel sollen einzelne Codesequenzen besprochen werden, die beispielhaft für die Lösung der Aufgabe stehen. Den Anfang macht die Bewertungsfunktion (Kapitel 3.3.1), dann folgt Ausschnitt aus der Selektion (Kapitel 3.3.3).

### Bewertungsfunktion

### Die Bewertungsfunktion

### Selektion

## Codequalität

Wie bereits im Kapitel 3.1 erwähnt, wird in diesem Projekt JUnit eingesetzt. Das Schreiben von Tests dient dazu die Codequalität des Programmcodes zu verbessern. In der folgenden Abbildung 4 sind die einzelnen Module des Systems aufgelistet und nach ihrer Abdeckung sortiert. Das bedeutet: Je mehr ein Modul durch Tests abgesichert ist, desto höher ist die Abdeckung und desto weiter oben erscheint es in der Liste.

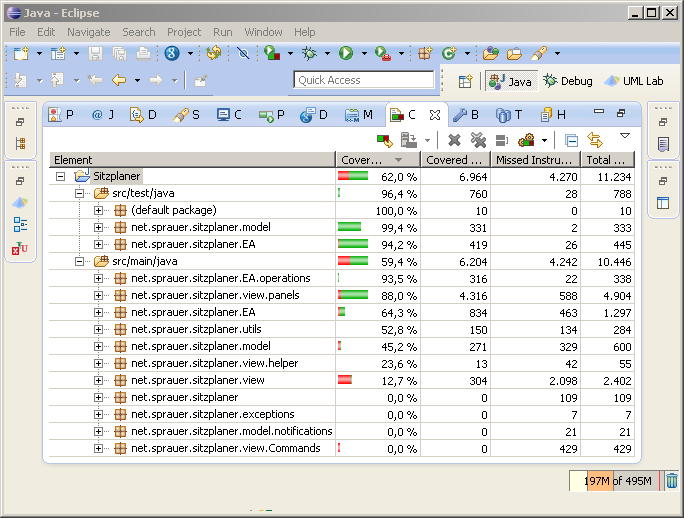


Abbildung 4: Abdeckung des Codes durch Unit-Tests

- Der Balken zeigt die absteigende Abdeckung je Modul in Prozent an. GUI Module haben typischerweise eine sehr niedrigen Grad der Codeabdeckung.

Insgesamt erreicht das Projekt eine Abdeckung von 62 Prozent. Für ein sehr GUI-lastiges Projekt ist das ein recht guter Wert. Es ist deutlich erkennbar, dass die View-Module die geringste Abdeckung haben. Das ist dadurch erklärbar, dass Oberflächen nur schwer automatisiert getestet werden können. Das Ergebnis einer Berechnung kann direkt mit dem Soll-Wert verglichen werden und daher sehr einfach Verifiziert werden. Bei einer Oberfläche können Anzeigeelemente nicht so einfach verifiziert werden. Das Diagramm Beispielsweise wird in ca. 50 Zeilen parametrisiert und bekommt die Werte zugewiesen. Das Ergebnis dieser Operation lässt sich aber nicht kontrollieren.

Die höchste Abdeckung haben die wichtigsten Bereiche: die EA Module, die Hilfsklassen (utils) und das Datenmodel (model).

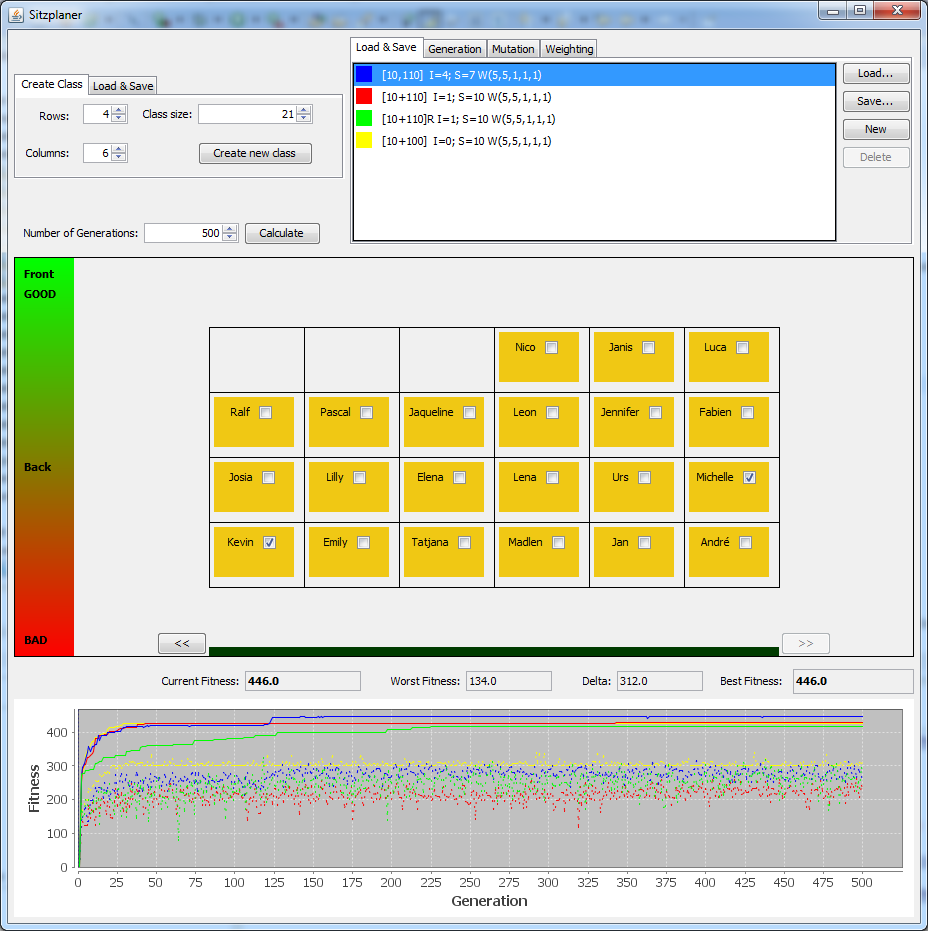
## Arbeitsaufwand

Was ebenfalls in dieser Grafik (Abbildung 4) sichtbar wird, ist die Verteilung der Komplexität bzw. des Aufwands. Der fachliche Anteil (das EA-Modul) fällt mit ca. 1500 Instruktionen recht moderat aus. Davon sind über 1100 Instruktionen über Test abgesichert. DerDerr Anteil für die GUI-Logik mit ist knapp 7500 Instruktionen das größte Modul. Ähnlich verteilt sich auch der geschätzte Arbeitsaufwand:

* GUI-Logik: ca. 70% Arbeitsaufwand
* Fachlogik: ca. 15% Aufwand
* Rest: Datenhaltung und Verwaltung, Import/Export

# Bedienung

Das Programm ist grob in drei Bereiche gegliedert, die in der folgenden Abbildung 5 farbig hervorgehoben sind.



**Klassenraum**

**Einstellungen**

**Fitness-Funktion**

Abbildung 5: Die drei Bereiche des Sitzplaners

## Einstellungen

Die Einstellungen zur Bedienung des Programms können in diesem ersten Abschnitt gesetzt werden.



Abbildung 6: Einstellungen des Programms

- Links der Bereich zum Erzeugen der Klasse und rechts die Parametrisierung des Evolutionären Algorithmus.

## Klassen erzeugen, laden und speichern

In der folgenden Tabelle 2 werden die Einstellungsmöglichkeiten und Funktionen zum Erzeugen einer Klasse und zum Berechnen des Sitzplanes erläutert. Diese befinden sich in Abbildung 6 auf der linken Seite.

|  |  |
| --- | --- |
| Create Class | Eine neue Klasse erzeugen. Die Abmessungen des Klassenraums kann in Anzahl der Zeilen (**Rows**) und Anzahl der Spalten (**Columns**) angegeben werden. Die Klassengröße – also die Anzahl der Schüler in der Klass – wird in **Class Size** eingestellt. Mit Klick auf **„Create new class“** wird die Klasse im Raum verteilt. Das Ergebnis ist im Klassenraum sichtbar. |
| Number of Generations | Anzahl der Generationen, die bei Klick auf **„Calculate“** erzeugt werden. Während der Berechnung wird über diesem Button ein Fortschrittsbalken angezeigt, der die Anzahl der berechneten Ergebnisse darstellt. Außerdem verändert sich der Text von **„Calculate“** zu **„Cancel“**. Damit kann die laufende Berechnung abgebrochen werden. |
| Calculate | Erzeugt die notwendige Anzahl an Generationen und Chromosomen und errechnet die Fitness-Werte für jeden Chromosomensatz. Das beste Ergebnis wird im Klassenraum angezeigt. |

Tabelle 2: Einstellungsmöglichkeiten Erzeugen und Berechnen

- zur Erzeugung von Klassen, Einstellung der Anzahl der Generationen und Berechnung des Sitzplans

|  |  |
| --- | --- |
| Load & Save | Über den Button **„Load“** kann eine gespeicherte Klasse in das Programm geladen werden. **„Save“** speichert eine Klasse. In der Datei werden nur die Beziehungen zueinander und nicht die Position der Schüler gespeichert. Ein neues Berechnen und Verändern ab dem gespeicherten Stand ist möglich. |

Tabelle 3: Einstellungsmöglichkeiten Laden und Speichern

## Konfigurationen erzeugen, laden und speichern

Die Liste auf der rechten Seite in Abbildung 6 zeigt alle geladenen Konfigurationen. Zwischen diesen Konfigurationen kann per Mausklick hin- und her geschaltet werden. Wird eine Konfiguration selektiert, so werden ihre Fitnesswerte über dem Diagramm dargestellt, der zugehörige Chromosomensatz wird in dem Klassenraum dargestellt und die Parameter werden in den Reitern **„Generation“**, **„Mutation“** und **„Weighting“** übernommen. In der Liste sind die wichtigsten Einstellungen in Textform kodiert. Diese Einstellung gibt an, dass aus einer Population von 880 die besten 80 als Eltern ausgewählt werden und aus jedem Elternchromosom dann ein Nachkomme durch Inversion und 10 Nachkommen durch Swap erstellt werden. Außerdem ist die Priorität mit 5 angegeben, die Gewichtung für den direkten Nachbarn beträgt 5 und der Reihe nach wird der übernächste, diagonale und darunter Sitzende Nachbar auf eins gesetzt.

Die Einstellungen zum Laden, Speichern und Erzeugen von Konfigurationen finden sich in Abbildung 6 auf der rechten Seite und werden in der folgenden Tabelle 4 beschrieben.

|  |  |
| --- | --- |
| Load & Save | Über die Buttons **„Load“** und **„Save“** können vordefinierte Parameter des Evolutionären Algorithmus geladen und gespeichert werden. Die Liste links von diesen Buttons zeigt alle geladenen Einstellungen. |
| New & Delete | Durch einen Klick auf **„New“** wird die aktuell selektierte Konfiguration dupliziert und erscheint als neuer Eintrag mit neuer Farbe in der Liste. Für jede Einstellung werden zwei Kurven im Statistik-Panel angezeigt. Die Farben werden automatisch der Reihenfolge nach zugewiesen. Der Button **„Delete“** löscht die selektierte Konfiguration. Es muss jedoch immer mindestens eine Konfiguration geladen sein. |

Tabelle 4: Bearbeiten der Konfigurationen Liste

- Alle Einstellungen, die die Berechnung des Sitzplans beeinflussen wie z.B. die Gewichtungsfaktoren und die EA-Parameter können in eine Datei gespeichert werden.

### Generation

Der Evolutionäre Algorithmus wird hauptsächlich durch die Definition der Generation und der Mutation parametrisieren. Die Generation legt fest aus welchen Elementen die aktuelle Population besteht.

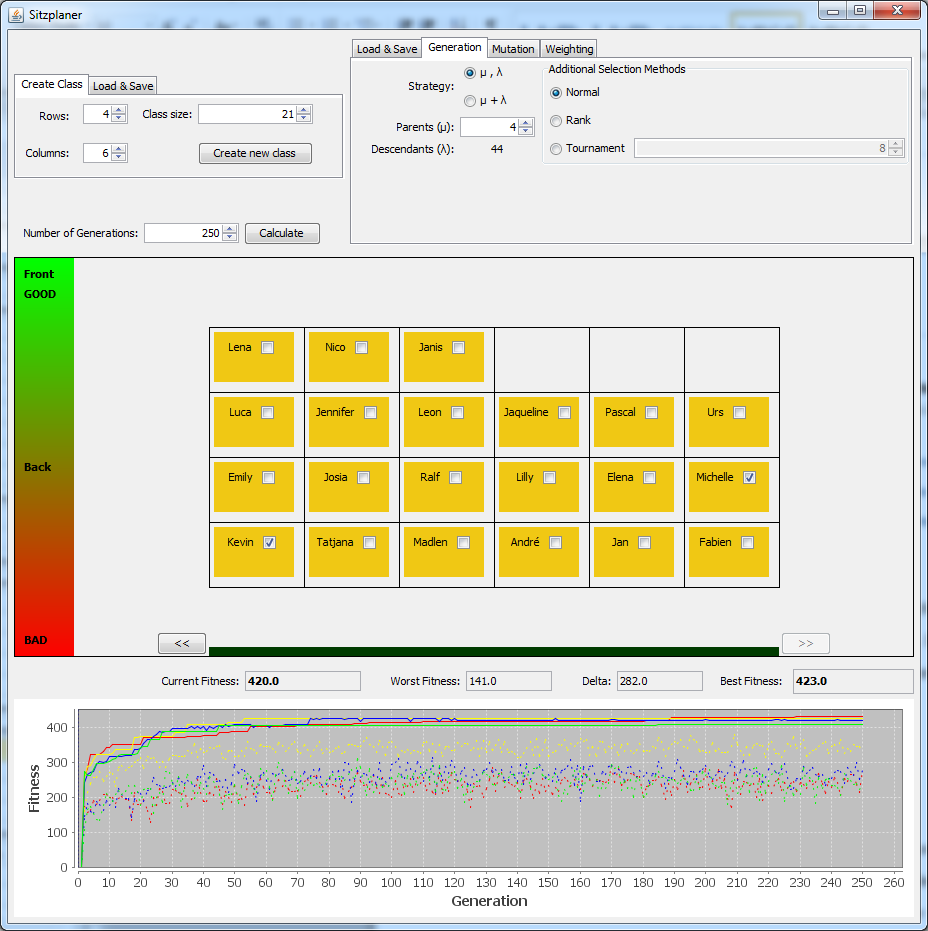


Abbildung 7: Einstellungen zur Bildung der Generationen

Die Abbildung 7 zeigt die möglichen Parameter und die zugehörige Tabelle 5 erläutert die einzelnen Optionen.

|  |  |
| --- | --- |
| Strategy | Bei der -Strategie kommen nur die Nachfahren der vorhergehenden Population in betracht. Mit dieser Strategie kann man lokale Optima besser wieder verlassen, die Fitness kann sich dadurch aber zwischendurch verschlechtern. Wählt man dagegen die -Strategie werden auch die Eltern der vorherigen Population mit einbezogen. Alle nachfolgenden Generationen können nie schlechter als das Elternelement werden. Damit verbessert man die Fitness kontinuierlich auf ein (lokales) Optimum hin. In Kapitel 2.6 findet sich die Theorie dazu. |
| Parents | Dieser Wert bestimmt den µ-Parameter – also die Anzahl der Chromosomen, die Nachkommen erzeugen dürfen. Je nach eingstellter Selektionsmethode werden beispielsweise nur die Besten 4 der vorherigen Generation zur Reproduktion bzw. Mutation zugelassen. |
| Normal | Jedes Element der Population darf gleich viele Nachkommen erzeugen. |
| Rank | Das beste Element der Elterngeneration darf am meisten Nachkommen erzeugen. Der zweit Beste einen weniger, und so weiter. Damit ergibt sich eine linear absteigende Funktion. Es kann vorkommen, dass ein die letzten Eltern keine Nachkommen erzeugen dürfen (siehe Kapitel 2.5). |
| Tournament | Bei dieser Methode wird das eine beste Chromosom aus 8 zufällig ausgewählten Chromosomen als Elternchromosom ausgewählt. Eine nähere Beschreibung findet sich in Kapitel 2.5. |

Tabelle 5: Einstellungen zur Bildung der Generationen

### Mutation

### Gewichtung

## Klassenraum

Der Klassenraum ist in einem Quadrat angeordnet. Der gelbe Bereich markiert einen Tisch. An jedem Tisch kann ein Schüler sitzen. Die Tafel ist unten im Bild zu sehen. Auf der linken Seite ist eine Legende eingezeichnet. Wenn man mit der Maus über die Schüler fährt, wird für jeden Schüler die Beziehung zu allen anderen Schülern angezeigt. Dabei entspricht grün einer guten Beziehung (diese Schüler sollten nebeneinander sitzen) und rot entspricht einer schlechten Beziehung (wenn diese Schüler nebeneinander sitze, wird der Unterricht gestört oder die Beiden lenken sich gegenseitig ab).

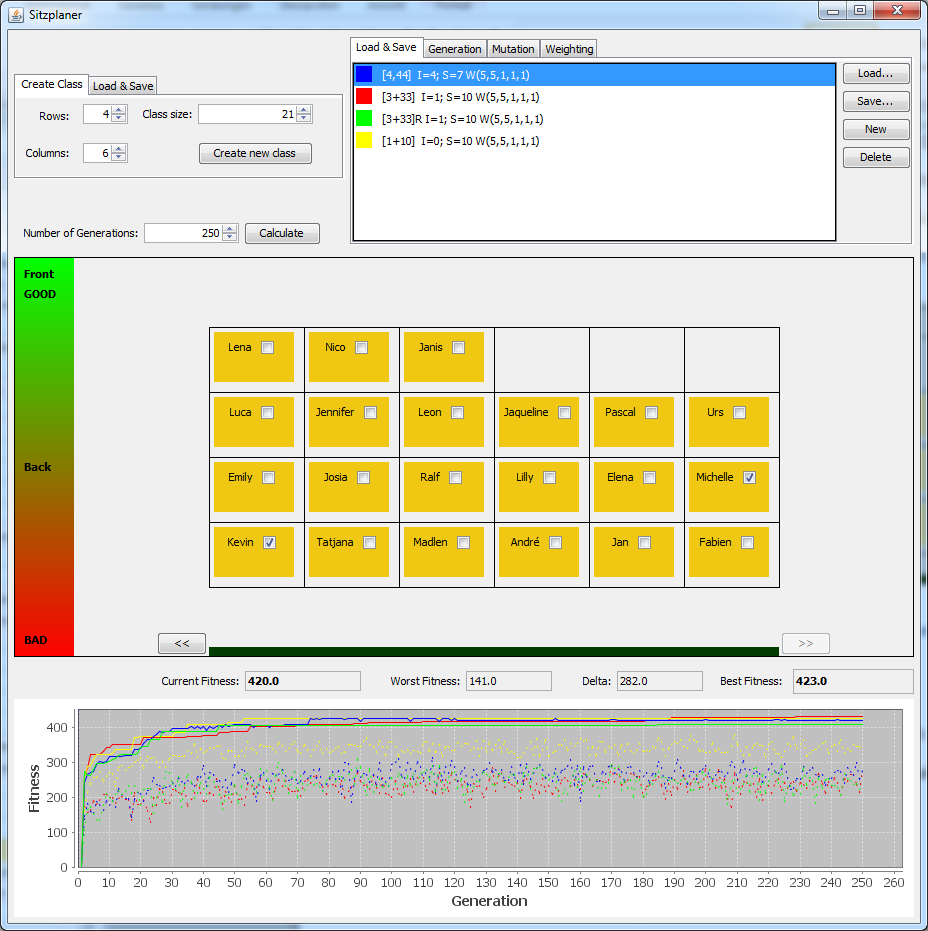


Abbildung 8: Klassenraum mit Tafel, Schülern und einer Legende

### Relationen der Schüler

Alle Eigenschaften der Schüler lassen sich in der Klassenraumansicht anzeigen und festlegen. Dabei wird zunächst zwischen zwei verschiedenen Relationsarten unterschieden:

1. Beziehung zwischen zwei Schülern
2. Beziehung des Schülers zur Tafel

Wenn die Maus über einem Schüler schwebt, werden seine Beziehungen angezeigt. In der Abbildung 8 ist die Maus über Lilly und ihre Relationen werden sichtbar. Hier ist beispielsweise die Relation zur Tafel=2. Das bedeutet, Lilly sollte nach Möglichkeit näher an der Tafel sitzen. Dieser Wert lässt sich nicht nur an der Linie zur Tafel ablesen, sondern ist auch als weißer Balken in der Legende eingezeichnet.

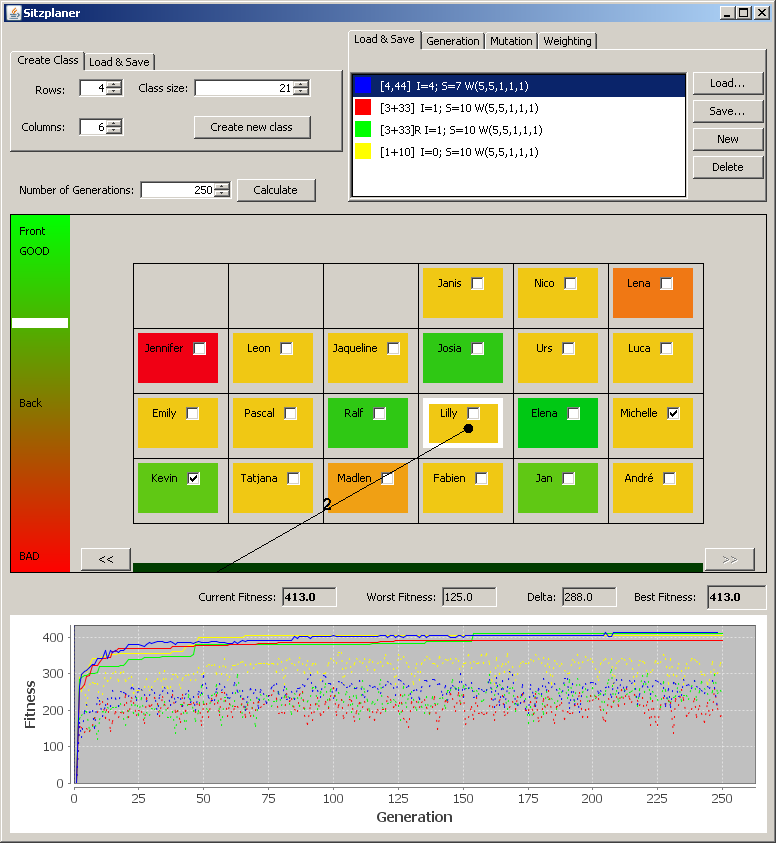


Abbildung 9: Die Relationen einer Schülerin werden angezeigt.

Außerdem werden die Relationen zu verschiedenen Mitschülern wie zum Beispiel Jennifer(-5=Rot) und Ralf (4=Grün) angezeigt. Diese Relationen lassen sich auch verändern. Wenn Lilly beispielsweise nicht neben Ralf sitzen soll, dann kann Lilly angeklickt werden, mit gedrückter linker Maustaste auf Ralf gezogen werden und dort die linke Taste losgelassen werden. Über das Mausrad kann nun die Relation verändert werden. Bei einer Drehung des Rads vom Benutzer weg, wird dieser Wert größer (Richtung grün). In die andere Richtung wird der Wert kleiner (Richtung rot). Der Wert wird an der Kante zwischen Lilly und Ralf angezeigt, ist in der Legende über den weißen Balken sichtbar und zusätzlich wird der Tisch von Jan in der entsprechenden Farbe eingefärbt. Ist der gewünschte Wert eingestellt, dann muss die Maus den Tisch von Jan verlassen. Damit wird die Bearbeitung abgeschlossen und der Wert übernommen. Die Beziehungen zwischen zwei Schülern ist immer dieselbe. Wenn Lilly einen Wert von 4 zu Ralf hat, dann hat Ralf auch einen Wert von 4 zu Lilly. Auf diese Weise lassen sich alle Beziehungen innerhalb der Klasse eintragen.

### Relation zur Tafel

Wenn die Relation zur Tafel geändert werden soll, dann muss der entsprechende Schüler auf die Tafel gezogen werden. Auch hier ist nun die Bestimmung des Wertes über das Mausrad möglich. Beendet und übernommen wird die Eingabe durch einen Klick mit der linken Maustaste. Der Wertebereich für diese Relation liegt zwischen 0 und 5. Ein negativer Wert kann also nicht vergeben werden.

In Abbildung 8 ist die Relation von Lilli zur Tafel an der Kante zwischen ihrem Sitzplatz und der Tafel zu sehen. Für sie wurde ein Wert von 2 vergeben. Sie sollte also nicht ganz hinten sitzen. Aber wenn in der ersten Reihe kein Platz mehr ist, verliert die Gesamtbewertung (je nach Gewichtung) nicht so viel.

## Fitness-Funktion

Der letzte Abschnitt der Oberfläche zeigt die Fitness-Funktion und damit die historische Entwicklung der Qualität der Sitzordnung.



Abbildung 10: Verlauf der Fitness und Fitness der aktuellen Repräsentation

In diesem Abschnitt ist auch der numerische Fitnesswert oben rechts eingeblendet. Wie die Abbildung 9 zeigt, steigt die Fitness in den ersten 25 Generationen stark an. Ab der 50. Generation verbessert sich die Fitness nur noch minimal. Aktuell wird ein Klassensatz angezeigt mit einer Fitness von 435. Der beste Wert ist mit 449 angegeben. Das macht ein Delta von 367 zum Schlechtesten, der 82 entspricht.

# Evaluation

In diesem Abschnitt werden die verschieden Parameter an einigen Beispielen untersucht. Damit sollte sich ein Parametersatz ergeben, der unter möglichst geringer Rechenzeit optimale Ergebnisse liefert. Als Datensatz für die Berechnung wurde ein 4 mal 6 großer Klassenraum erstellt und mit 21 Schülern besetzt. Damit bleiben 3 Plätze frei. Außerdem wurden einige Relationen gesetzt und zwei Schüler wurden in ihrer Platzwahl auf ihren aktuellen Platz beschränkt. Diese Datei ist auf dem Datenträger im Ordner „Klassendaten“ zu finden.

Zunächst sollen die beiden Mutationsoperatoren Swap und Inversion untersucht werden.

## Mutation durch Swap

Für den Swap-Operator kann man einstellen, wie viele Swaps je Nachkommen durchgeführt werden sollen. Dieser Wert sollte nicht zu hoch sein, da sonst die Rechenzeit steigt und am Ende eventuell zwei bereits getauschte Schüler wieder zurück getauscht werden. Er darf aber auch nicht zu niedrig sein, sonst müssen zu viele Generationen erzeugt werden um vom Ausgangspunkt zum Zielpunkt zu kommen. Daher soll nun Untersucht werden welcher Wert optimal ist. Dazu werden verschiedene Werte eingestellt und die Ergebnisse verglichen. Die Einstellungen finden sich in Tabelle 5.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Konfiguration | Parameter | Blau | Rot | Grün | Gelb | Schwarz |
| Swap | Strategy | + | + | + | + | + |
| Selection | Normal | Normal | Normal | Normal | Normal |
| Parents | 10 | 10 | 10 | 10 | 10 |
| Swaps | 10 | 10 | 10 | 10 | 10 |
| Swap operations | **1** | **2** | **3** | **4** | **5** |
| Inversion | 0 | 0 | 0 | 0 | 0 |
| Inversion operations | 0 | 0 | 0 | 0 | 0 |
|  | Descendants | 110 | 110 | 110 | 110 | 110 |

Tabelle 6: Evaluation – Parameter: Wie viele Swaps je Nachkommen?

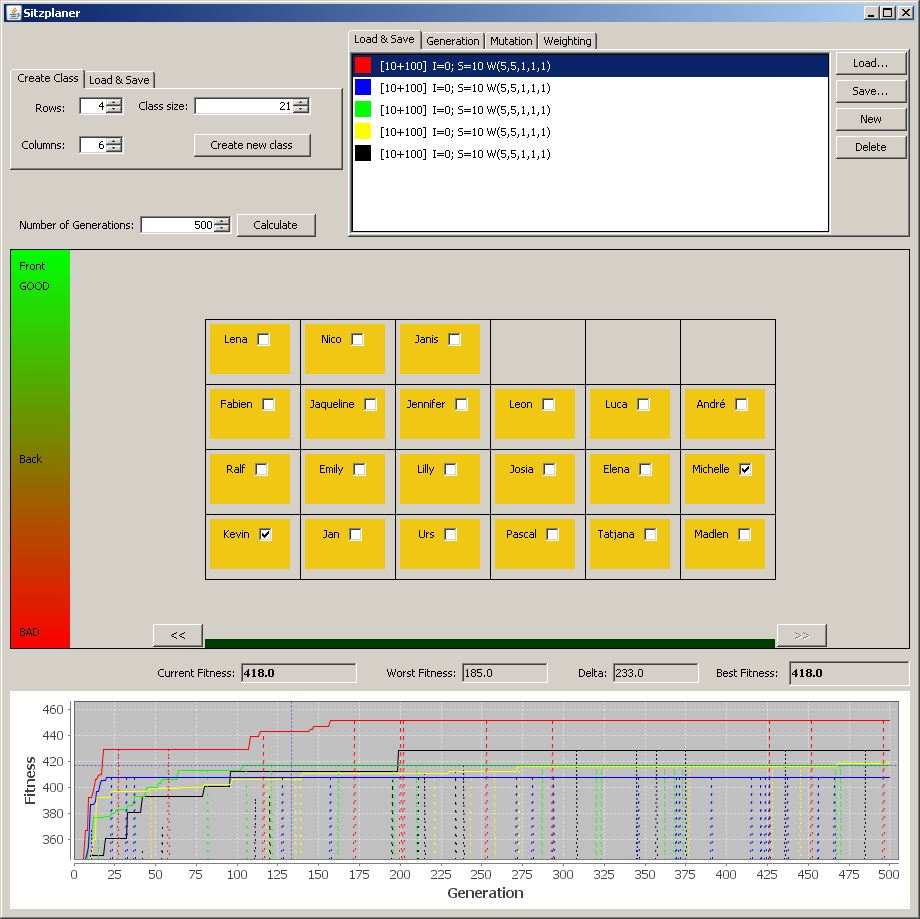


Abbildung 11: Wie viele Swaps je Nachkommen

- Den besten Wert mit 451 erreicht die rote Kurve mit 2 Swaps je Nachkommen. Deutlich wird: Je mehr Swaps desto langsamer steigt die Fitness.

Die Antwort ist nicht so eindeutig, wie der Screenshot es vermuten lässt, denn es gibt auch Fälle in denen mit 4 oder 5 Swaps die besten Ergebnisse erzielt wurden. Deutlicher ist dagegen der Zusammenhang zwischen Anfangssteigung und Swap-Anzahl. Je mehr Swaps eingestellt werden, desto langsamer steigt die Fitness.

Mit 2 bis 3 Swaps je Nachkommen kommt man schnell zu guten Ergebnissen. Wenn man ausreichend viele Generationen erzeugt (hier über 1.000), kann man auch mit 6 oder mehr Swaps den besten Sitzplan erzeugen, allerdings nicht so zuverlässig. Mit nur einem Swap je Nachkommen stagniert die Fitness bereits nach ca. 50 Generationen bei einer durchschnittlichen Fitness.

## Mutation durch Inversion

Dieselbe Evaluation wird nun für die Inversion durchgeführt. Bei der Inversion werden eine oder mehrere zufällig ausgewählte Reihen invertiert.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Konfiguration | Parameter | Blau | Rot | Grün | Gelb | Schwarz |
| Inversion | Strategy | + | + | + | + | + |
| Selection | Normal | Normal | Normal | Normal | Normal |
| Parents | 10 | 10 | 10 | 10 | 10 |
| Swaps | 0 | 0 | 0 | 0 | 0 |
| Swap operations | 0 | 0 | 0 | 0 | 0 |
| Inversion | 10 | 10 | 10 | 10 | 10 |
| Inversion operations | **1** | **2** | **3** | **4** | **5** |
|  | Descendants | 110 | 110 | 110 | 110 | 110 |

Tabelle 7: Evaluation – Parameter: Wie viele Inversionen je Nachkommen?

Hier stellt sich bereits nach drei Generationen für alle Varianten ein konstanter Wert ein, der für eine gerade Anzahl an Inversionsvorgängen bei 239 liegt und für eine ungerade Anzahl bei 240. Diese Werte ergeben sich aus der sehr geringen Anzahl an Reihen in der Beispielklasse. Jede Inversion kann nur aus 4 verschiedenen Reihen wählen. Die Wahrscheinlichkeit bei geraden Inversionsvorgängen, die eben vorgenommene Inversion rückgängig zu machen ist sehr hoch.

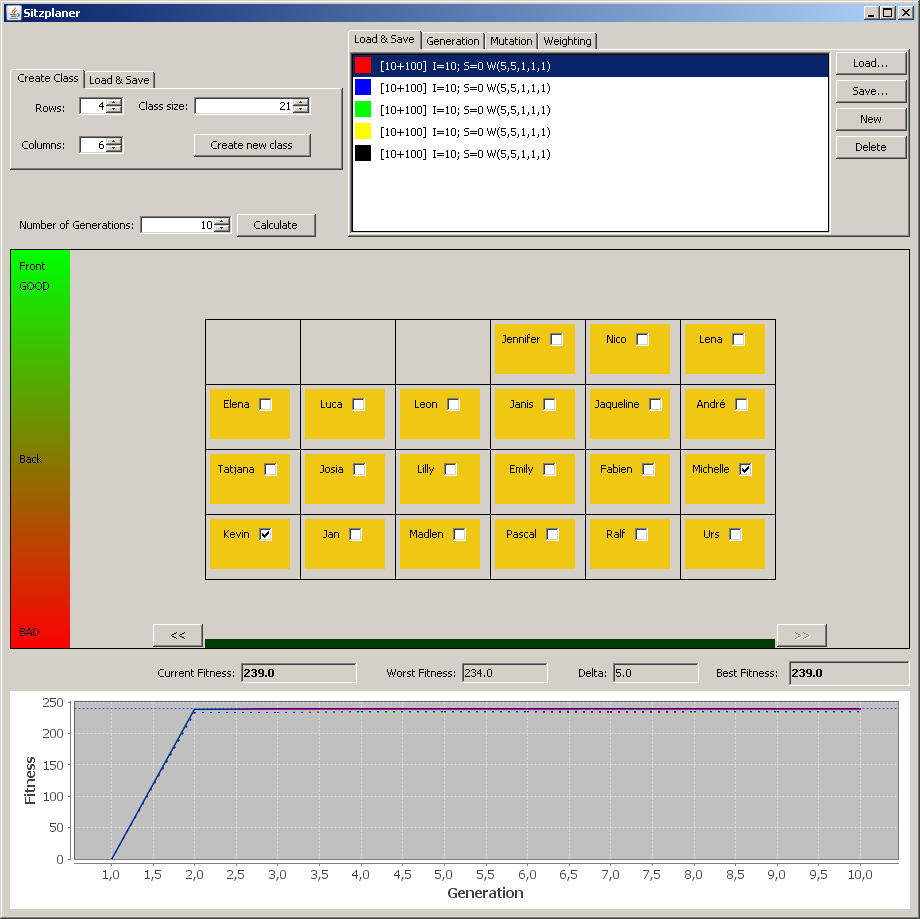


Abbildung 12: Wie viele Inversionen je Nachkommen?

Da die Ergebnisse für eine Inversion fast identisch mit den Ergebnissen für 2 oder mehr Inversionen sind, empfiehlt es sich diesen Wert aus Effizienz Gründen auf 1 zu setzen.

## Auswahl der Mutationsoperatoren

Nachdem gezeigt wurde, welche Parameter für die Mutation durch Swap bzw. durch Inversion optimal sind, soll nun untersucht werden, welcher der beiden Operatoren besser ist. Als Konfiguration werden die Werte wie in Tabelle 7 eingestellt.

|  |  |  |  |
| --- | --- | --- | --- |
| Konfiguration | Parameter | Blau | Rot |
| 11\*Inversion (Blau)  verglichen mit  10\*Inversion + 1\*Swap  (Rot) | Strategy | + | + |
| Selection | Normal | Normal |
| Parents | 10 | 10 |
| Swaps | **0** | **1** |
| Swap operations | N/A | 2 |
| Inversion | **11** | **10** |
| Inversion operations | 1 | 1 |
|  | Descendants | 110 | 110 |

Tabelle 8: Evaluation – Parameter: Wird Swap benötigt?

Mit dieser Parametrierung kommen beide Konfigurationen auf 110 Nachkommen und hätten damit die gleiche Chance den besten Nachkommen zu bilden. In Abbildung 12 wird jedoch deutlich, dass bereits nach der 2. Generation in der blauen Konfiguration keine Verbesserung mehr eintritt. Die rote Konfiguration, bei der je Generation ein Nachkomme durch Swap erzeugt wurde, stagniert nach ca. 100 Generationen auf einer Fitness zwischen 400 bis 420. Ohne Swaps kommt die Fitness nicht über 240 hinaus.

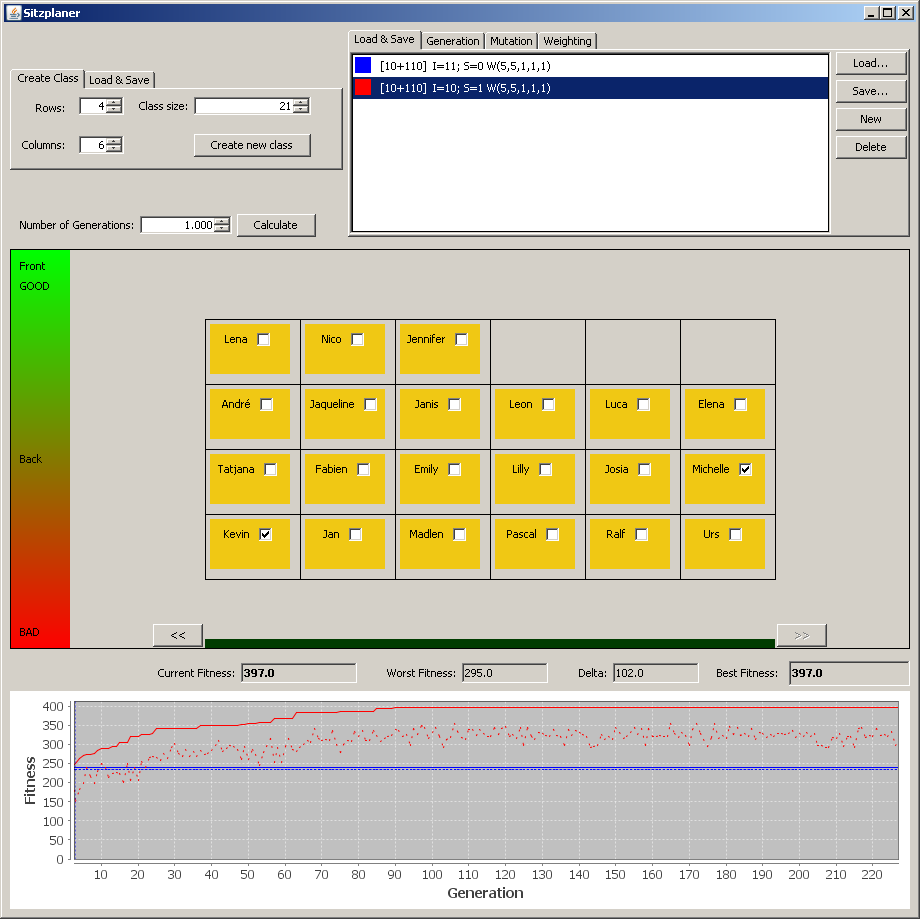


Abbildung 13: Wird Swap benötigt?

- Die blaue Linie wird ausschließlich mit durch Inversion erzeugten Nachkommen gebildet. Der rote Verlauf entsteht durch eine Kombination von 10 Inversions-Nachkommen und einem Swap-Nachkommen.

Dieses Ergebnis überrascht nicht wirklich, denn allein durch invertieren einer Sitzreihe kommt nicht genug Variabilität in Sitzordnung. Insgesamt gibt es nur verschiedene Möglichkeiten die Reihen zu invertieren. Bei 10 Nachkommen ist die Wahrscheinlichkeit nach 2 Generationen bereits bei 83%, dass das optimale Ergebnis darunter ist.

Nun könnte man vermuten, die Inversion ist überflüssig. Diese These wird in der zweiten Evaluation untersucht. Dabei werden nur die Vorgaben für Swap und Inversion vertauscht. Die Parametrierung enthält Tabelle 8.

|  |  |  |  |
| --- | --- | --- | --- |
| Konfiguration | Parameter | Blau | Rot |
| 11\*Swap (Blau)  verglichen mit  10\*Swap + 1\*Inversion  (Rot) | Strategy | + | + |
| Selection | Normal | Normal |
| Parents | 10 | 10 |
| Swaps | **11** | **1** |
| Swap operations | 2 | 2 |
| Inversion | **0** | **10** |
| Inversion operations | N/A | 1 |
|  | Descendants | 110 | 110 |

Tabelle 9: Evaluation – Parameter: Wird Inversion benötigt?

Hier fällt das Ergebnis deutlich knapper aus. Die Abbildung 13 zeigt die Verläufe der Fitness mit diesen Parametern.

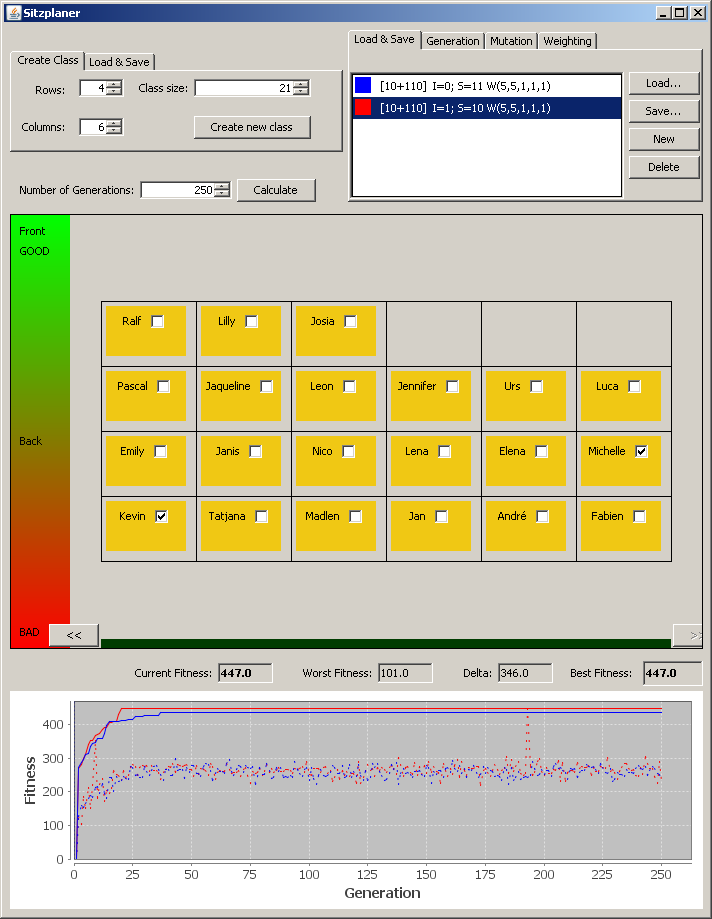


Abbildung 14: Wird Inversion benötigt?

- Nach etwa 50 Generationen erreichen beide Konfigurationen ihr Maximum. Die Kombination Swap+Inversion liegt sie jedoch etwa 19 Punkte über der Variante ohne Inversion.

In mehreren Messungen wurden die Ergebnisse überprüft. Die folgende Tabelle 9 zeigt die Ergebnisse nach mehreren Durchläufen.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Durchlauf | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |  |
| Rot | 431 | 442 | 441 | 432 | 428 | 413 | 419 | 449 | 426 | 449 | 433 |
| Blau | 421 | 437 | 425 | 417 | 440 | 429 | 419 | 428 | 406 | 414 | 423,6 |

Tabelle 10: Vergleich Swap mit und ohne Inversion nach 10 Durchläufen

- Im Mittel sind die Swapmutationen in Kombination mit der Inversion ca. 5 Punkte besser. Es gibt jedoch auch Fälle in denen das Ergebnis ohne Inversion besser ist.

Offensichtlich kommt man nur mit der Swapmutation ohne die Verwendung der Inversion auf annährend gleich gute Ergebnisse. Im Mittel liegt jedoch die Kombination Swap + Inversion 5 Punkte über den Ergebnissen ohne Inversion. Daraus kann man schließen, dass die optimale Konfiguration einen deutlich größeren Anteil an Swaps als Inversionen enthalten sollte.

## Strategien

## Selektion

# Zusammenfassung und Ausblick

Dieses Kapitel soll aufzeigen welche Teilaspekte des Projekts im Sinne der Anforderungen, oder auch darüber hinaus, nicht optimal gelöst werden konnten.

Das Ziel des Projekts war es Evolutionäre Algorithmen zur Erstellung eines Sitzplans zu verwendet. In den Kapiteln 2 und 3 wurden die Grundlagen und die Implementierung erläutert. Kapitel 4 befasste sich mit der Beschreibung des fertigen Programms. Der Abschnitt 5 zeigt, dass die gestellte Aufgabe voll erfüllt wurde. Es ist mit Hilfe des Programms möglich einen Klassenraum in Rasterform zu modellieren und mit Schülern zu füllen. Außerdem können die geforderten Einschränkungen und Beziehungen modelliert werden und zum Schluss ein optimaler Sitzplan errechnet werden. Diese Daten lassen sich über eine File-Schnittstelle aus dem Programm exportieren und wieder in das Programm laden.

Zusätzlich dazu sind verschiedene Einstellungen möglich um die Erzeugung bzw. die Effizienz der Erzeugung zu beeinflussen und die Konfiguration zur Erzeugung zu speichern. Um einen schnellen Überblick über die Qualität des Sitzplans zu bekommen wurde ein Diagramm eingebaut, das den Verlauf der Qualität darstellt.

## Verbesserungsmöglichkeiten

Zu den eingangs gestellten Anforderungen sind im Laufe des Projekts noch einige zusätzliche Ideen entstanden, die aus Zeitmangel jedoch nicht umgesetzt werden konnten.

Um die Berechnung auch von großen Klassen mit vielen Beziehungen möglichst schnell zu ermöglichen, könnte man die Fitness der einzelnen Chromosomen differentiell errechnen. Bei einer Änderung durch Swap würden die Werte der beiden Schüler errechnet, von der Gesamtfitness abgezogen, an der neuen Position wieder berechnet und zur Gesamtfitness addiert.

Als weitere Optimierung wäre eine Anpassung des Algorithmus auf mehrere Kerne denkbar. Dazu muss der rechenaufwändige Teil so umformuliert werden, dass er parallel läuft und möglichst unabhängig voneinander ist. Diese Änderung ist relativ einfach umsetzbar, indem die Fitnessberechnung auf mehrere Kerne verteilt wird. Eine Übersicht über verschiedene Verfahren dazu gibt [8].

Die höchste Leistung, die mit heutigen handelsüblichen Rechnern möglichst ist, stellt die Verwendung der Grafikkarte dar. Sie ist, im Gegensatz zu der CPU, genau auf die parallele Berechnung ausgelegt. Eine Untersuchung dazu wird gerade von Andreas Bartschat im Rahmen seiner Studienarbeit vorgenommen.

## Fazit

Wie diese Studienarbeit zeigt, bieten die Evolutionären Algorithmen eine praktische und einfache Möglichkeit komplizierte Probleme zu lösen, für die es ohne weiteres keine Lösung gibt oder bei denen es nur unter sehr großem Aufwand möglich ist Lösungen zu errechnen. Die praktische Umsetzung dieses Problems hat mir sehr anschaulich die Herangehensweise zur Implementierung gezeigt. Am Ende war ich sehr überrascht, dass selbst für große Klassen mit vielen Beziehungen sehr schnell eine brauchbare Lösung errechnet werden konnte.

1. Glossar

**EA:** Evolutionäre Algorithmen 16

**GUI:** Graphical User Interface 16

**IDE:** Integrated Development Environment 11

**MVC:** Model View Controller 13

**WYSIWYG:** What You See Is What You Get 12

1. Abbildungsverzeichnis

[Abbildung 1: Gewichtung der Umgebung eines Schülers 6](#_Toc356415641)

[Abbildung 2: Git Versionslog 13](#_Toc356415642)

[Abbildung 3: MVC-Muster 14](#_Toc356415643)

[Abbildung 4: Abdeckung des Codes durch Unit-Tests 15](#_Toc356415644)

[Abbildung 5: Die drei Bereiche des Sitzplaners 17](#_Toc356415645)

[Abbildung 6: Einstellungen des Programms 18](#_Toc356415646)

[Abbildung 7: Klassenraum mit Tafel, Schülern und einer Legende 21](#_Toc356415647)

[Abbildung 8: Die Relationen einer Schülerin werden angezeigt. 22](#_Toc356415648)

[Abbildung 9: Verlauf der Fitness und Fitness der aktuellen Repräsentation 23](#_Toc356415649)

[Abbildung 10: Wie viele Swaps je Nachkommen 25](#_Toc356415650)

[Abbildung 11: Wie viele Inversionen je Nachkommen? 27](#_Toc356415651)

[Abbildung 12: Wird Swap benötigt? 29](#_Toc356415652)

[Abbildung 13: Wird Inversion benötigt? 30](#_Toc356415653)

1. Tabellenverzeichnis

[Tabelle 1: Verwendete Eclipse Plugins 12](#_Toc356415654)

[Tabelle 2: Einstellungsmöglichkeiten Erzeugen und Berechnen 19](#_Toc356415655)

[Tabelle 3: Einstellungsmöglichkeiten Laden und Speichern 19](#_Toc356415656)

[Tabelle 4: Bearbeiten der Konfigurationen Liste 20](#_Toc356415657)

[Tabelle 5: Evaluation – Parameter: Wie viele Swaps je Nachkommen? 25](#_Toc356415658)

[Tabelle 6: Evaluation – Parameter: Wie viele Inversionen je Nachkommen? 27](#_Toc356415659)

[Tabelle 7: Evaluation – Parameter: Wird Swap benötigt? 28](#_Toc356415660)

[Tabelle 8: Evaluation – Parameter: Wird Inversion benötigt? 30](#_Toc356415661)

[Tabelle 9: Vergleich Swap mit und ohne Inversion nach 10 Durchläufen 31](#_Toc356415662)

1. Listings

[Formel 1: Permutationscodierung - ist der Schüler an der Stelle (i,i) 4](#_Toc356415663)

1. Literaturverzeichnis

[1] H.-P. Schwefel, “Evolutionsstrategie und numerische Optimierung,” Technische Universität Berlin, 1975.

[2] I. Rechenberg, “Evolutionsstrategie–Optimierung technisher Systeme nach Prinzipien der biologischen Evolution,” 1973.

[3] D. E. Goldberg and J. H. Holland, “Genetic algorithms and machine learning,” *Machine Learning*, vol. 3, no. 2, pp. 95–99, 1988.

[4] G. Syswerda, “Uniform crossover in genetic algorithms,” 1989.

[5] W. M. Spears and K. A. De Jong, “An analysis of multi-point crossover,” DTIC Document, 1990.

[6] T. Starkweather, S. McDaniel, K. Mathias, D. Whitley, and C. Whitley, *A comparison of genetic sequencing operators*. Citeseer, 1991.

[7] B. R. Fox and M. B. McMahon, “Genetic operators for sequencing problems,” *Foundations of genetic algorithms*, vol. 1, pp. 284–300, 1991.

[8] E. Cantú-Paz, “A survey of parallel genetic algorithms,” *Calculateurs paralleles, reseaux et systems repartis*, vol. 10, no. 2, pp. 141–171, 1998.