Entwurf und Implementierung eines Genetischen Algorithmus zur Erstellung von Zeitplänen

Studienarbeit im Fach Informatik

vorgelegt von

Stefan Büttcher

geboren am 17. November 1978 in Celle

angefertigt am

Institut für Informatik
Lehrstuhl für Programmiersprachen
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: Matthias Gröbner, Peter Wilke

Beginn der Arbeit: 1. März 2002

Abgabe der Arbeit: 29. November 2002

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ich bin damit einverstanden, dass die Arbeit veröffentlicht und dass in wissenschaftlichen Veröffentlichungen auf sie Bezug genommen wird.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Programmiersprachen, wird ein (nicht ausschließliches) Nutzungsrecht an dieser Arbeit sowie an den im Zusammenhang mit ihr erstellten Programmen eingeräumt.

Erlangen, am 29. November 2002

....

(Vorname, Name)

Inhaltsverzeichnis

1 Einleitung und Definitionen **2**
1.1 Einführung in die Problematik . . . 2
1.2 Definitionen und Terminologie . . . 3
1.3 Problemkomplexität . . . 4
1.4 Formale Beschreibung eines Timetabling-Problems . . . 6

2 Struktur eines Timetabling-Problems **8**
2.1 Klassenstruktur . . . 8
2.2 Modellierung der Zeit . . . 10
2.3 Repräsentation der Ereignisse . . . 11
2.4 Mengen von Ressourcen verschiedener Art . . . 12
2.5 Die einzelne Ressource . . . 14
2.6 Festlegen der Rahmenbedingungen . . . 14
2.7 Ausgabe der Ergebnisse . . . 15

3 Der Genetische Algorithmus **17**
3.1 Einführung in Genetische Algorithmen . . . 17
3.2 Klassenstruktur des Genetischen Algorithmus . . . 23
3.3 Ablauf eines Evolutionsvorgangs . . . 24
3.4 Stundenpläne als Chromosomen: Anwendung auf Timetabling-Probleme . . . 24
3.5 Mutation und Crossover bei Zeitplänen . . . 26
3.6 Weiterführende Operatoren . . . 30
3.7 Einsatz hybrider Operatoren . . . 31
3.8 Kapselung von Zeitplänen: `TTChromosome` . . . 33
3.9 Parametrisierung des Genetischen Algorithmus . . . 33

4 Testprobleme **35**
4.1 Mädchen-und-Technik-Praktikum . . . 35
4.2 Personaleinsatzplanung . . . 37
4.3 Schulstundenplanerstellung . . . 38

5 Ergebnisse **40**
5.1 Mädchen-und-Technik-Praktikum . . . 40
5.2 Personaleinsatzplanung . . . 43
5.3 Schulstundenplanerstellung . . . 45
5.4 Einsatz hybrider Operatoren . . . 46

6 Zusammenfassung und Ausblick **49**

I ANHANG **51**

7 Verzeichnisübersicht **52**

8 Programmquellen **53**
8.1 Paketübersicht 53
8.2 Das Paket `ga` 53
8.3 Das Paket `ttclasses` 54
8.4 Das Paket `examples` 55
8.5 Das Paket `helperclasses` 56
8.6 Das Paket `tools` 57

9 Arbeiten mit dem Timetabling-GA **58**
9.1 Konstruieren eines eigenen Timetabling-Problems 58
9.2 Definieren eigener Constraints und Reparaturmechanismen 58
9.3 Einlesen einer Problem-Instanz 59
9.4 Ausgabe eines fertigen Zeitplans 60
9.5 Starten des Genetischen Algorithmus 61
9.6 Verwenden der Protokoll-Informationen 62

10 Der parallele Genetische Algorithmus **64**
10.1 Arten paralleler Genetischer Algorithmen 64
10.2 Struktur der vorliegenden Implementierung 64
10.3 Ergebnisse 65

11 Zusätzliche Hilfsprogramme **67**
11.1 Ein grafisches Frontend für den GA 67
11.2 Ein Timetable-Explorer 68

Literaturverzeichnis **69**

Index **72**

Abbildungsverzeichnis

1.1 Einfaches Färbbarkeitsproblem 5
1.2 Problembeschreibung in GTL 7

2.1 Allgemeine Klassenstruktur eines TT-Problems 9
2.2 Repräsentation von Zeitpunkten und -intervallen 10
2.3 Repräsentation der Ereignisse 11
2.4 Struktur der Klasse `ResourceList` 12
2.5 Ressourcenverwaltung 14
2.6 Repräsentation der Constraints 15
2.7 Ausgabe von Zeitplänen 16

3.1 One-Point und Uniform Crossover 22
3.2 Java-Quelltext eines einfachen Mutationsoperators 23
3.3 Klassenstruktur des Genetischen Algorithmus 23
3.4 Struktogramm für den Ablauf des Evolutionsvorgangs 25
3.5 Verhältnis von `TTProblem` und `GeneticAlgorithm` 26
3.6 Berechnung der Binärdarstellung eines Zeitplans 26
3.7 Struktur eines Schulstundenplans 27
3.8 Mutation eines Zeitplans 27
3.9 Uniform Crossover mit zwei Ereignissen 29
3.10 Crossover mit Ereignissen als Genen 30
3.11 Anwendung hybrider Operatoren (Java-Code) 32
3.12 Verhältnis von `TTProblem` und `TTChromosome` 33

5.1 Evolutionsverlauf beim MuT-Problem 41
5.2 Evolutionsverlauf beim MuT-Problem (detailliert) 42
5.3 Evolutionsverlauf beim Krankenhaus-Problem 43
5.4 Mitarbeiter-Schichtpläne für das Krankenhausproblem 44
5.5 Evolutionsverlauf beim Schulstundenplan-Problem 45
5.6 Individuelle Stundenpläne für das Schul-Problem 46
5.7 Evolution mit hybriden Operatoren (Schulstundenplan) 46
5.8 Auswirkungen von ELITE_PERCENTAGE im Hybrid-Modus 48

9.1 XML-Code für die Instantiierung eines Timetabling-Problems 59
9.2 Grafische Darstellung der Timeslots eines Zeitplans 60
9.3 Text-Ausgabe eines Zeitplans 61
9.4 XML-Ausgabe eines Zeitplans 61
9.5 HTML-Ausgabe eines Zeitplans 62
9.6 Starten des Genetischen Algorithmus 62
9.7 Die Protokolldatei des Genetischen Algorithmus 63

10.1 Kommunikation beim verteilten GA 65
10.2 Vergleich von normalem und parallelem GA 66

11.1 Evolutionsvorgang im GA-Frontend 67
11.2 Der Timetable-Explorer im Einsatz 68

Tabellenverzeichnis

2.1 Variablen der Klasse `ResourceList` 13

4.1 Rahmenbedingungen des „Mädchen und Technik“-Problems 36
4.2 Verschiedene Instanzen des „Mädchen und Technik“-Problems 37
4.3 Rahmenbedingungen des Krankenhaus-Problems 37
4.4 Verschiedene Instanzen des Krankenhaus-Problems 38
4.5 Rahmenbedingungen des Schulstundenplan-Problems 38

5.1 GA-Parametrisierung für das MuT-Praktikum 41
5.2 Constraint-Verletzungen beim MuT-Problem 42
5.3 GA-Parametrisierung für das Krankenhaus-Problem 43
5.4 Constraint-Verletzungen beim Krankenhaus-Problem 44
5.5 Constraint-Verletzungen: Hybrid vs Steady-State (Schulstundenplan) 47

8.1 Paketübersicht 53
8.2 Klassenübersicht für `ga` 54
8.3 Klassenübersicht für `ttclasses` 54
8.4 Klassenübersicht für `examples.maedchen` 55
8.5 Klassenübersicht für `examples.hospital` 56
8.6 Klassenübersicht für `helperclasses` 56
8.7 Klassenübersicht für `tools` 57

Kurzfassung

In vielen Bereichen der Wirtschaft und des öffentlichen Lebens müssen Zeitpläne verschiedenster Art erstellt werden. Zu diesem Zweck sind in der Vergangenheit häufig evolutionäre Verfahren, unter ihnen vor allem Genetische Algorithmen, eingesetzt worden. Leider handelt es sich bei der Mehrzahl um Verfahren, die jeweils lediglich für eine sehr kleine Klasse von Problemen konzipiert worden sind und sich daher auf die meisten Timetabling-Probleme nicht anwenden lassen.

In dieser Arbeit wird eine Grundstruktur vorgestellt, mit der sich beliebige Timetabling-Probleme definieren und bearbeiten lassen. Auf Basis dieser Grundstruktur ist ein Genetischer Algorithmus implementiert worden, mit dem sich alle diese Probleme automatisch lösen bzw. optimieren lassen, so dass im Anschluss an die Erstellung eines Zeitplans mit dem Genetischen Algorithmus nur noch eine geringe Arbeit von Hand zu leisten ist.

Abschließend wird der vorgestellte Genetische Algorithmus anhand von realen Testbeispielen aus verschiedenen Bereichen gestestet. Die Ergebnisse werden mit denen bestehender, auf diese speziellen Probleme zugeschnittener, Algorithmen verglichen. Es zeigt sich, dass der hier gewählte, allgemeine Ansatz sich, wenn problemspezifisches Wissen sinnvoll in die Problemlösung eingebracht wird, nicht hinter für spezielle Probleme entwickelten Verfahren zu verstecken braucht.

Kapitel 1

Einleitung und Definitionen

1.1 Einführung in die Problematik

Zeitpläne und die mit ihnen verbundenen Probleme lassen sich in vielfältiger Form überall finden: Angefangen vom Schulstundenplan, der wohl jedem in diesem Zusammenhang als Erstes in den Sinn kommt, über Personaleinsatzplanung oder die Raumplanung einer Universität bis hin zur Terminvergabe beim Zahnarzt handelt es sich um verschiedene Arten von *Timetabling-Problemen*, also Problemen, bei denen es darum geht, sinnvolle Zeitpläne zu erstellen.

Diese Probleme sind sowohl in ihrer Struktur als auch in ihrer Schwierigkeit sehr verschieden: Die Erstellung des Stundenplans für eine Schule bereitet den verantwortlichen Planern zum Beispiel jedes Jahr aufs Neue viel Arbeit, während das mit den Zahnarztterminen verbundene Timetabling-Problem von einer Angestellten durch einen Blick in den Terminkalender innerhalb kurzer Zeit gelöst werden kann.

Allen Problemen liegt jedoch ein gemeinsames Prinzip zu Grunde: Aus einer Menge von Ereignissen soll jedem Einzelereignis eine Menge von Zeitintervallen, während denen dieses Ereignis stattfindet, und eine Menge von Ressourcen, die dieses Ereignis belegt, zugeteilt werden. Eine Ressource ist in diesem Sinn zum Beispiel ein Klassenraum, in dem eine bestimmte Lehrveranstaltung stattfindet. Aber auch der Lehrer, der das Fach unterrichtet, und die Klasse, die unterrichtet wird, sind von diesem Standpunkt aus betrachtet lediglich Ressourcen.

Was das Erstellen von Stundenplänen so schwierig macht, sind bestimmte Rahmenbedingungen (sog. *Constraints*), die eingehalten werden müssen oder sollen. Wenn man das Beispiel der Schulstundenplanerstellung betrachtet, sind einige Bedingungen völlig offensichtlich, so etwa, dass kein Raum zum selben Zeitpunkt von verschiedenen Ereignissen belegt werden darf oder dass nicht jedes Unterrichtsfach in einem beliebigen Raum unterrichtet werden kann, der Sportunterricht also zum Beispiel in der Turnhalle und nicht im Physiksaal stattfinden muss. Andere Bedingungen sind schon etwas subtilerer Natur, wie zum Beispiel die unter Umständen sinnvolle Forderung, dass jede Klasse beim (unvermeidlichen) Wechsel des Unterrichtsraums möglichst kurze Strecken zurücklegt und nicht etwa nach jeder Stunde einmal den gesamten Gebäudekomplex durchqueren muss.

Man sieht, dass die Schwierigkeit mitunter gar nicht erst beim Erstellen eines Stundenplans beginnt, sondern bereits bei der präzisen Formulierung der einzuhaltenden Rahmenbedingungen. Dies wird besonders schwierig, wenn irgend eine Art von Gerechtigkeit oder Gleichverteilung erwünscht ist, zum Beispiel in der Arbeitsbelastung der Lehrer oder der Stundenlast der Schulklassen an den einzelnen Tagen, die möglichst ausgeglichen sein sollte.

Ziel dieser Arbeit ist es, zunächst eine Menge von Datenstrukturen zu entwerfen und vorzustellen, mit denen allgemeine Timetabling-Probleme beschrieben werden können, und dann einen Genetischen Algorithmus zu implementieren, der tatsächlich in der Lage ist, Zeitpläne dieser allgemeinen Bauart zu optimieren, im besten Fall also ein gegebenes Problem zu lösen.

1.2 Definitionen und Terminologie

In der Literatur findet man häufig verschiedene Bedeutungen des Begriffs *Timetabling-Problem*. So wird er etwa in [Fan94, S. 2] für Probleme verwendet, bei denen es ausschließlich darum geht, eine Menge von Ereignissen auf eine Menge von Zeitintervallen zu verteilen, während in [RCF94] ein Timetabling-Problem mit der Aufgabe, Zuteilungen der Art

$$Ass \;:=\; (e,t,p,a) \;\in\; E\times T\times P\times A,$$

zu finden, gleichgesetzt wird, wobei E eine Menge von Ereignissen, T eine Menge von Zeitintervallen, P eine Menge von Orten und A eine Menge von Agenten ist, eine Zuteilung Ass also die Bedeutung „das Ereignis e findet im Zeitraum t am Ort p mit dem Agenten a statt“ hat.

In [GW02] schließlich ist mit dem Begriff Timetabling-Problem der allgemeinste Fall gemeint, in dem jedem Element e einer Menge von Ereignissen E eine bestimmte Menge von Ressourcen $R \in 2^{\mathcal{R}}$, also eine Untermenge aller zur Verfügung stehenden Ressourcen $\mathcal{R}$, zugeteilt wird. Dabei kann es sich bei den Ressourcen um Räume, Agenten oder Zeitintervalle handeln.

Die Betrachtungen des letzten Abschnitts und der Wunsch, Missverständnisse aufgrund unklaren Sprachgebrauchs zu vermeiden, geben nun genügend Motivation für die folgenden formalen Definitionen :

Sei $\mathcal{E}$ eine Menge von Ereignissen, $|\mathcal{E}| =: e$, und $\mathcal{R}$ eine Menge von Ressourcen. Sei ferner

$$\mathcal{C} := \{c \mid c :\; (\mathcal{E}\times 2^{\mathcal{R}})^e \to \mathbb{N}_0\}$$

die Menge aller mit $\mathcal{E}\times\mathcal{R}$ kompatiblen Constraints, die jeder Verteilung von Ressourcen auf Ereignisse eine natürliche Zahl, die sog. *Penalty Points* zuweisen. Dann bezeichnen $\mathcal{H}\subset\mathcal{C}$ und $\mathcal{S}\subset\mathcal{C}$ die *Hard Constraints* , die nicht verletzt werden dürfen, bzw. die *Soft Constraints* , die zwar verletzt werden dürfen, die aber nicht verletzt werden sollen.

Mit diesen Festlegungen ist die folgende Definition sinnvoll: Das 4-Tupel

$$\mathcal{TP} := (\mathcal{E},\mathcal{R},\mathcal{H},\mathcal{S}),$$

mit den oben eingeführten Größen, heiße ein *Timetabling-Problem* . Es sei bemerkt, dass diese Definition an anderer Stelle (z.B. [Fan94, S. 5]) im Wesentlichen dem Begriff *Scheduling-Problem* entspricht.

Es ist nun jedes Element

$$TT \in \mathcal{T} := (\mathcal{E} \times 2^{\mathcal{R}})^e, \;\; \forall\, \varepsilon \in \mathcal{E}\; \exists\, R \in 2^{\mathcal{R}} : (\varepsilon, R) \in TT,$$

ein *Zeitplan zum gegebenen Timetabling-Problem* und $\mathcal{TT}$ die Menge aller Zeitpläne.

Damit ist klar, dass die Menge aller *Lösungen* eines Timetabling-Problems

$$\begin{aligned} \mathcal{L} &:= \{L \in \mathcal{TT} \mid \sum_{h \in \mathcal{H}} (h(L)) = 0\} \\ &= \{L \in \mathcal{TT} \mid \forall\, h \in \mathcal{H}\; h(L) = 0\} \end{aligned}$$

ist, also genau die Teilmenge von $\mathcal{TT}$, die aus den Stundenplänen besteht, in denen keine Hard Constraints verletzt werden, und

$$\mathcal{L}^* := \{L \in \mathcal{TT}^* \mid \sum_{s \in \mathcal{S}} s(L) = min_{L' \in \mathcal{L}^*}\{\sum_{s \in \mathcal{S}} s(L')\}\ \}$$

die Menge aller *optimalen Lösungen* , also genau die Stundenpläne, in denen keine Hard Constraints verletzt werden und in denen die durch die Soft Constraints hervorgerufenen Penalty Points minimal sind. Es sei an dieser Stelle gleich bemerkt, dass keinesfalls immer eine optimale Lösung existieren muss, da es natürlich Timetabling-Probleme gibt, die keine Lösung besitzen. In solchen Fällen soll dann statt von optimalen Lösungen einfach von *optimalen Zeitplänen* gesprochen werden. Diese terminologische Festlegung widerspricht üblichen Wortverwendung in der einschlägigen Literatur, die auch bei Zeitplänen, in denen Hard Constraints verletzt werden, von *Lösungen* spricht.

Sei nun

$$\mathcal{F} := \{f \mid f:\; \mathcal{E} \to \mathcal{E} \times 2^{\mathcal{R}}\}$$

die Menge aller Funktionen, die jedem Ereignis ein Paar, bestehend aus dem Ereignis selbst und einer Menge von Ressourcen, zuordnet. Dann heiße der zu jedem $f \in \mathcal{F}$ gehörende Prozess ein *Timetabling-Vorgang* und

$$\mathcal{TT} \ni TT_f := \{f(\varepsilon) \mid \varepsilon \in \mathcal{E}\}$$

sein *Ergebnis*.

1.3 Problemkomplexität

Zur Komplexität von allgemeinen Timetabling-Problemen stammt die zentrale Aussage bereits aus dem Jahr 1979: Wir betrachten ein simples Timetabling-Problem, in dem für eine Menge von Ereignissen $\mathcal{E}$ jedem Ereignis $e \in \mathcal{E}$ eine Menge von Ressource eines bestimmten Typs, z.B. Räume $r_{e,1}..r_{e,n}$ aus einer Menge von Räumen $\mathcal{R}$, fest zugewiesen ist. Gesucht ist nun eine Zuordnung

$$f:\; \mathcal{E} \to \mathcal{E} \times \mathcal{T}\; ;\; e \mapsto (e,t),$$

wobei $\mathcal{T}$ eine Menge paarweise nicht überlappender Zeitintervalle ist, so dass kein Raum während des selben Zeitintervalls mehrfach benutzt wird.

Das hier zur Anwendung kommende Constraint, das Kollisionen verbietet (sog. *Clash Constraint*) lässt sich formal beschreiben:

$$\forall\, e \in \mathcal{E}\, \neg(\exists\, e' \in \mathcal{E}\, \exists\, t \in \mathcal{T}:\ f(e) = (e,t) \wedge f(e') = (e',t) \wedge e \neq e' \wedge r_e = r_{e'}).$$

Garey und Johnson zeigen in [GJ79], dass dieses einfache Zuweisungsproblem bereits *NP-vollständig* ist und somit kein bekannter PTIME-Algorithmus existiert, es also nach derzeitigem Wissensstand im Allgemeinen nicht in polynomialer Zeit auf deterministische Weise gelöst werden kann.

Beweisskizze. Dass das beschriebene Problem in NP liegt, ist klar, denn für einen gegebenen Stundenplan lässt sich in polynomialer Zeit leicht überprüfen, ob er eine Lösung des Problems ist.

Es bleibt zu zeigen, dass es NP-hart ist. Dieser Beweis geschieht durch Reduktion des Färbbarkeitsproblems von Graphen auf das Timetabling-Problem. Das Färbbarkeitsproblem lässt sich als Entscheidungs- oder Optimierungsproblem formulieren. Es geht also entweder darum, ob sich die Knoten eines Graphen mit k verschiedenen Farben einfärben lassen, ohne dass zwei durch eine Kante verbundene Knoten die selbe Farbe haben, oder darum, das minimale k zu finden, so dass die Färbbarkeit möglich ist. Färbbarkeit (engl. „Graph Colouring“) ist NP-vollständig, was man u.a. durch Reduktion von 3KNF-SAT auf dieses Problem zeigen kann (vgl. [Sch97, S. 177 ff.]).

Sei $G = (V,E)$ ein ungerichteter Graph mit Knotenmenge V und Kantenmenge E. Dann konstruiert man sich ein zu G äquivalentes Timetabling-Problem, indem man die Knoten des Graphen als Ereignisse betrachtet und für jede Kante $(v_1, v_2) \in E$ den Ereignissen v_1 und v_2 gemeinsam einen neu erzeugten Raum als Ressource fest zuweist.

Die Frage „Lassen sich die Knoten von G mit k Farben einfärben, ohne dass benachbarte (über eine Kante verbundene) Knoten die selbe Farbe haben?“ lässt sich nun einfach umformulieren in: „Lassen sich die Ereignisse des Timetabling-Problems auf k verschiedene Timeslots verteilen, ohne dass ein Raum zum selben Zeitpunkt mehrfach benutzt wird?“

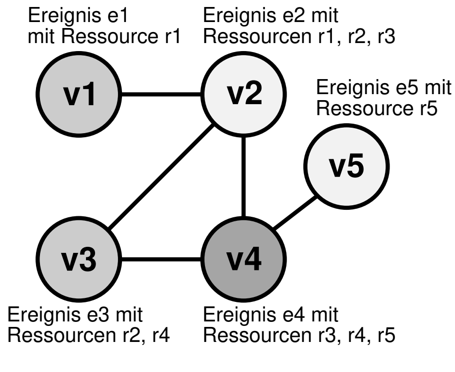


Abbildung 1.1: Einfaches Färbbarkeitsproblem

In Abb. 1.1 sieht man leicht, dass das zu diesem Graphen gehörende Färbbarkeitsproblem äquivalent zu dem skizzierten Timetabling-Problem mit 5 Ereignissen e1, e2, e3, e4 und e5 ist. Weil sich die Knoten des Graphen nicht mit 2 Farben einfärben lassen, ist es auch nicht möglich, die Ereignisse des Timetabling-Problems auf nur zwei zur Verfügung stehende Zeitintervalle zu verteilen. Für $k = 3$ gibt es jedoch eine Lösung.

Es ist offenkundig, dass wegen der NP-Vollständigkeit dieser Klasse simpler Zeitzuweisungsprobleme auch jede Klasse komplexerer Probleme NP-vollständig (bzw. NP-hart, falls jemand auf die Idee kommt, Constraints festzulegen, deren Überprüfung superpolynomialen Aufwand erfordert) ist, denn durch die Hinzunahme weiterer Bedingungen wird die Suche im Allgemeinen nicht einfacher. Es soll hier darauf hingewiesen werden, dass es natürlich spezielle Unterklassen von Timetabling-Problemen geben kann, die effiziente (d.h. polynomiale) Lösungsalgorithmen zulassen. Im Allgemeinen kann man jedoch erwarten, dass die NP-Härte erhalten bleibt.

Nach diesen grundsätzlichen Überlegungen zur Komplexität der Stundenplanerstellung ist klar, dass man sich in den meisten Fällen mit einem Stundenplan begnügen muss, der im Sinne der obigen Definition noch nicht einmal eine Lösung des Timetabling-Problems, erst recht also keine optimale Lösung darstellt. Weil mehr leider auf absehbare Zeit nicht zu erwarten ist, sucht man nach Verfahren, die möglichst gute Zeitpläne, also lokale Optima im Raum der zu einem Problem gehörenden Zeitpläne, möglichst effizient finden.

In der Vergangenheit sind zu diesem Zweck verschiedene Techniken eingesetzt worden. Costa verwendet zum Beispiel eine Tabu-Suche zum Lösen des einfachen Timetabling-Problems, in dem nur die Ereignisse nur auf Zeitintervalle verteilt werden müssen ([Cos94]). Kang und White stellen in [KW92] ein ein Verfahren der Logischen Programmierung vor, mit dem sie Schulstundenpläne erstellen.

Diese Arbeit widmet sich dem Finden von möglichst guten Zeitplänen mit Hilfe Genetischer Algorithmen für den Fall des allgemeinen Timetabling-Problems.

1.4 Formale Beschreibung eines Timetabling-Problems

Zur Beschreibung von allgemeinen Timetabling-Problemen wurde in [GW02] eine Sprache vorgestellt, mit der sowohl die Relationen zwischen Ressourcen und Ereignissen als auch die jeweiligen Rahmenbedingungen formalisiert werden können. Diese *General Timetabling Language* (*GTL*) kann dann z.B. in Java übersetzt und das Problem mit dem in dieser Arbeit vorgestellten Genetischen Algorithmus gelöst werden.

Die GTL-Beschreibung eines Personalplan-Problems in einem Krankenhaus könnte beispielsweise so aussehen (einige Teile sind aus Platzgründen weggelassen):

```
TimetablingProblem Hospital {
  class Shift extends Event {
    int shiftType;                       // type, e.g. Early Shift
  }
  class Staff extends Resource {
    string realName;                     // real name of the employee
    double workLoad;                     // weekly work load
    reference(Shift)[1] absent;          // absence due to holiday, etc.
  }
  class ClashConstraint extends HardConstraint {
```

Fortsetzung auf der nächsten Seite

```
    public ConstraintViolation compute(PlanningClass owner) {
      int j, k;                              // loop dummies
      Staff res = (Staff)owner;
      List l = res.getAssignments();
      double akku = 0;
      for (j = 0; j < l.size(); j++) {
        Assignment ass1 = (Assignment)l.get(j);
          for (k = j + 1; k < l.size(); k++) {
            Assignment ass2 = (Assignment)l.get(k);
            if (ass1.event.clashes(ass2.event))
              akku += 200;                   // add 200 penalty points
          }
      }
      if (akku > 0)
        return new ConstraintViolation(this, owner, akku);
      return null;
    }
    public boolean repair(ConstraintViolation violation) {
      return false;                          // no reparation method
    }
  }
  class OutOfMinMaxConstraint extends HardConstraint {
    (body left out)
  }
  class DiffHourMonthConstraint extends SoftConstraint {
    (body left out)
  }
}
```

Abbildung 1.2: Problembeschreibung in GTL

Wirklich kompliziert ist dabei allein die Definition der verschiedenen Constraints und ihrer `compute`- und `repair`-Methoden, die Java-ähnlicher Weise geschehen.

Zusätzlich zur Struktur eines Timetabling-Problems, die mit Hilfe der Definitionssprache GTL festgelegt wird, muss zur Lösung oder Optimierung eines solchen Problems auch noch die konkrete Instantiierung vorgegeben werden, also in eindeutiger Weise erklärt sein, welche Instanzen eines bestimmten Ressourcentyps welchen konkreten Ereignissen zugeordnet werden können. Diese Festlegung erfolgt mittels einer XML-Datei. Ein Beispiel kann im Anhang dieser Arbeit gefunden werden.

Die konkrete Ausgestaltung von GTL wurde parallel zur Implementierung dieses Genetischen Algorithmus zur Optimierung von Zeitplänen durchgeführt, so dass eine unmittelbare Rückkopplung möglich war und auf bestimmte Bedürfnisse, wie etwa die Struktur der XML-Eingabe, die Fixierung von Listen und andere Besonderheiten, die später noch beschrieben werden werden, Rücksicht genommen werden konnte.

Kapitel 2

Struktur eines Timetabling-Problems

Die mengentheoretische Struktur eines Timetabling-Problems wurde im vorherigen Kapitel bereits ausführlich beschrieben. Wenn es nun um die Wahl der internen Darstellung von Zeitplänen bzw. der Beschreibung eines Timetabling-Problemen mit Hilfe programmiertechnischer Strukturen geht, sind im Wesentlichen zwei Dinge zu beachten:

- Die verwendeten Strukturen müssen es ermöglichen, jedes beliebige Timetabling-Problem (im Sinne der mathematischen Definition) darzustellen, und
- die Darstellung muss unabhängig vom einzusetzenden Lösungs- bzw. Optimierungsverfahren sein.

Sind diese zwei Bedingungen erfüllt, so ist es ohne weitere Änderungen an der Datenstruktur leicht möglich, ein neues Verfahren für ein gegebenes Timetabling-Problem zu implementieren und das Problem damit zu lösen bzw. zu optimieren.

2.1 Klassenstruktur

Die für die interne Darstellung eines Problems gewählte Struktur, auf der jedes Problemlösungsverfahren basiert, ist in Abb. 2.1 zu sehen.

Wie bereits beschrieben, besteht ein Timetabling-Problem aus

- einer Menge von Ereignissen (*Events*),
- einer Menge von Ressourcen, die den Ereignissen in irgendeiner Weise zugeordnet werden sollen und
- einer Menge von Randbedingungen (*Constraints*), die die Regeln festlegen, nach denen diese Zuordnung zu erfolgen hat.

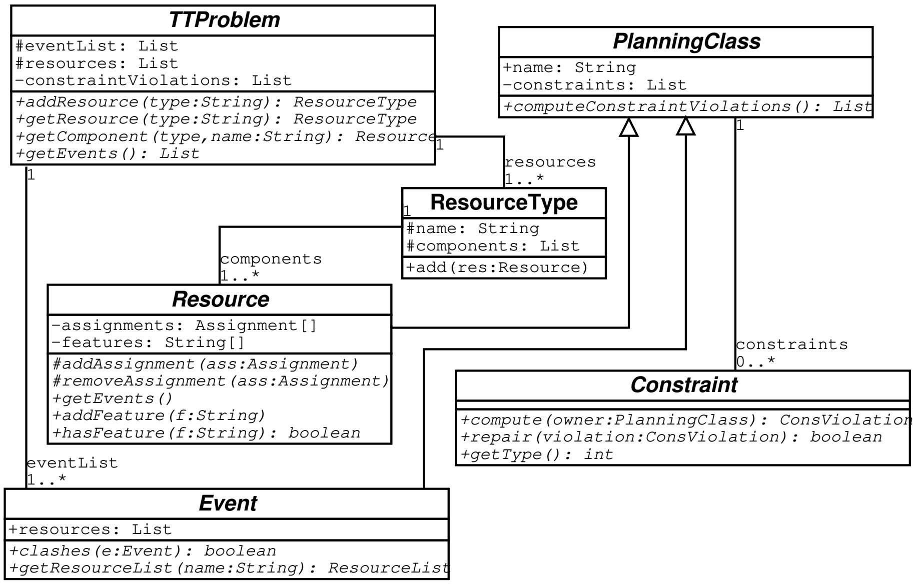


Abbildung 2.1: Allgemeine Klassenstruktur eines TT-Problems

Aus der Abbildung wird deutlich, dass jedes Timetabling-Problem eine Instanz einer Klasse ist, die die abstrakte Klasse `TTProblem` beerbt. `TTProblem` verfügt zum einen über eine Liste `eventList`, die ihr Zugriff auf alle zugehörigen Ereignisse gibt, zum anderen über eine Liste `resources`, über die alle zum Problem gehörenden Ressourcen erreicht werden können.

Die Klasse `ResourceType` dient der Kapselung von Ressourcen verschiedenen Typs. Handelt es sich bei dem TT-Problem beispielsweise um einen Schulstundenplan, so werden Lehrer, Klassen, Stunden und Räume jeweils in einer eigenen Instanz von `ResourceType` gespeichert.

In der gewählten abstrakten Sichtweise wird im Laufe des Planungsprozesses jedem Ereignis eine Menge von Ressourcen zugeteilt, es ist also auch der Zeitraum, in dem ein bestimmtes Ereignis stattfindet, bloß eine gewöhnliche Ressource, die sich allerdings durch eine besondere Eigenschaft, die Fähigkeit, Kollisionen herbeizuführen, von allen anderen Ressourcen unterscheidet: Betrachtet man die drei Zuteilungen

$$A_1 := (e_1, t_1, r_1, a_1),\ A_2 := (e_2, t_2, r_1, a_1),\ A_3 := (e_3, t_1, r_2, a_1)$$

mit den Ereignissen e_i, den Zeitintervallen t_j, Räumen r_k und Agenten a_l, so ergibt sich bezüglich der Zuteilungen A_1 und A_3 eine Kollision, weil der selbe Agent a_1 an zwei verschiedenen Ereignissen teilnimmt, die beide während des Intervalls t_1 stattfinden. Dass die Ereignisse e_1 und e_2 hingegen im selben Raum r_1 stattfinden, führt zu keiner Kollision.

Die Kennzeichnung der Zeitintervalle als Ressourcen mit besonderer Eigenschaft geschieht nun dadurch, dass das Objekt der Klasse `ResourceType`, das alle Zeitintervalle zusammenfasst, immer den Namen „TimeSlot" trägt. Diese Bezeichnung wird an anderer Stelle zur Detektion von Kollisionen (sog. *Clashes*) verwendet und darf daher nicht verändert werden.

`Resource` und `Event` besitzen als Verallgemeinerungen der abstrakten Klasse `PlanningClass` jeweils eine Liste `constraints`, in der alle zu einer Ressource oder einem Ereignis gehörenden Rahmenbedingungen festgehalten sind. Die Lösung, Constraints lokal zu verwalten, entspringt zum einen einem objektorientierten ästhetischen Empfinden, zum anderen praktischen Überlegungen, da so das Reparieren von Verletzungen einfacher möglich wird.

In Ausnahemefällen kann es nötig sein, Constraints global zu definieren, um beispielsweise eine Fairness in der Belastung aller beteiligten Ressourcen eines bestimmten Typs zu gewährleisten. Zu diesem Zweck wird in der abstrakten Klasse `TTProblem` eine Liste `globalConstraints` definiert, die alle Constraints, die sich keinem bestimmten Ereignis und keiner bestimmten Ressource zuordnen lassen, enthält.

Das Feld `assignments` in der Klasse `Resource` schließlich ermöglicht konsistente Sichten der einzelnen am Timetabling-Prozess beteiligten Ressourcen. Dadurch, dass alle Ereignisse, denen eine Ressource zugeteilt worden ist, hier festgehalten werden, ist es mit geringem Aufwand möglich, zeitliche Überschneidungen bei der Zuteilung zu erkennen oder den kompletten Zeitplan in der Sicht der jeweiligen Ressource zu erstellen.

2.2 Modellierung der Zeit

Es ist klar, dass bei der Erstellung von Zeitplänen die Modellierung der Zeit, für die etwas geplant werden soll, eine wichtige Rolle spielt. Es existieren Timetabling-Probleme für viele verschiedene zeitliche Strukturen:

- Schulstundenpläne, die einen wöchentlich wiederkehrenden Ereignisablauf für eine Menge von Timeslots (z.B. 10 Schulstunden an 5 Tagen, also 50 Timeslots) festlegen,
- Arbeitszeiten in einem Unternehmen, die für jeweils einen Monat verschiedene Schichten (Früh-, Spät-, Nachtschicht) festlegen, oder
- Prüfungszeitpläne einer Universität, die zum Beispiel die an der Semesterabschlussprüfung teilnehmenden Studenten nach bestimmten Kriterien auf die Vor- und Nachmittage von drei verschiedenen Tagen verteilen.

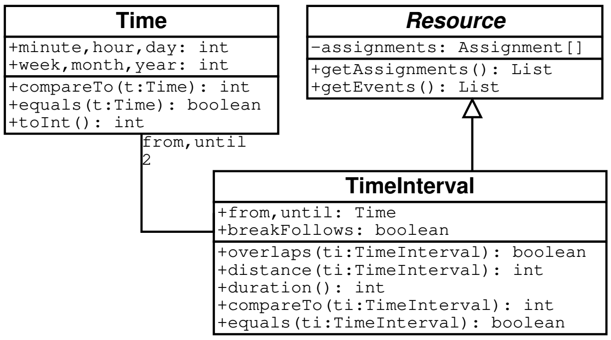


Abbildung 2.2: Repräsentation von Zeitpunkten und -intervallen

Die Klassen `Time` und `TimeInterval`, die in Abb. 2.2 dargestellt sind, sollen der Verschiedenheit der zu verplanenden Zeiträume Rechnung tragen. Mit Hilfe der unterschiedlichen Konstruktoren der Klasse `Time` lassen sich Zeitpunkte erzeugen, die in den Zusammenhang eines einzelnen Tages, einer Woche, eines Monats oder eines Jahres eingebettet sind.

Manchmal ist es sinnvoll, mehrere Zeitintervalle zu einer untrennbaren Einheit zusammenzufassen. Dies ist zum Beispiel beim Schulstundenplan der Fall, wenn ein bestimmtes Unterrichtsfach, z.B. Sport, in einer Doppelstunde unterrichtet werden soll. In einem solchen Fall kann eine Instanz der Klasse `TimeIntervals` verwendet werden, um eine Menge von Zeitintervallen zu bündeln.

2.3 Repräsentation der Ereignisse

Ein einzelnes Ereignis innerhalb eines Zeitplans kann als Sammelstelle von Ressourcen verschiedener Arten betrachtet werden. Im Beispiel der Schulstundenplanerstellung versammeln sich Ressourcen der Typen *Schulklasse*, *Lehrer*, *Raum* und *Schulstunde* in Ereignissen des Typs *Unterricht*.

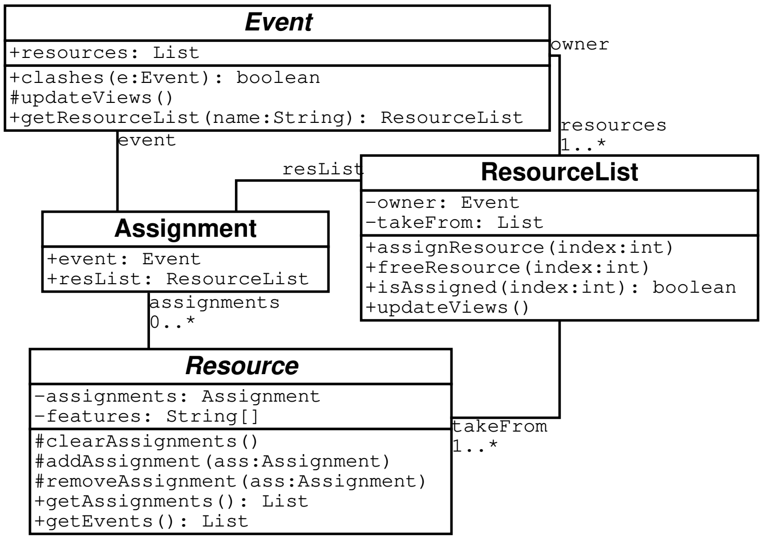


Abbildung 2.3: Repräsentation der Ereignisse

Zeitliche Kollisionen werden mit Hilfe der Methode `clashes` erkannt, die untersucht, ob sich zwei Ereignisse zeitlich überschneiden. Für das Reservieren von Ressourcen für ein Ereignis und das Freigeben von früher einmal reservierten Ressourcen sind die Methoden `assignResource` und `freeResource` der Klasse `ResourceList` verantwortlich, die ihrerseits durch Aufrufen der Methoden `addEvent` und `removeEvent` der Klasse `Resource` die Konsistenz des Stundenplans gewährleisten.

`clearEvents` implementiert im Zusammenspiel mit `updateViews` eine Reset-Funktionalität.

2.4 Mengen von Ressourcen verschiedener Art

Die Klasse `ResourceList`, die gewissermaßen im Auftrag eines Ereignisses eine Liste von Ressourcen verwaltet, ist der eigentliche Kern eines jeden Zeitplans: Hier werden die Ressourcen angefordert und freigegeben, und nur hier finden effektive Manipulationen am Zeitplan statt.

Ein Ereignis selbst hat nur stark eingeschränkte Zugriffsmöglichkeit auf die Ressourcen, die ihm zugeordnet sind. So kann es beispielsweise über die Methode `assignResource(Resource)` eine bestimmte Ressource anfordern oder mit `freeResource(Resource)` wieder freigeben. Aber die tatsächliche Kontrolle liegt bei `ResourceList`.

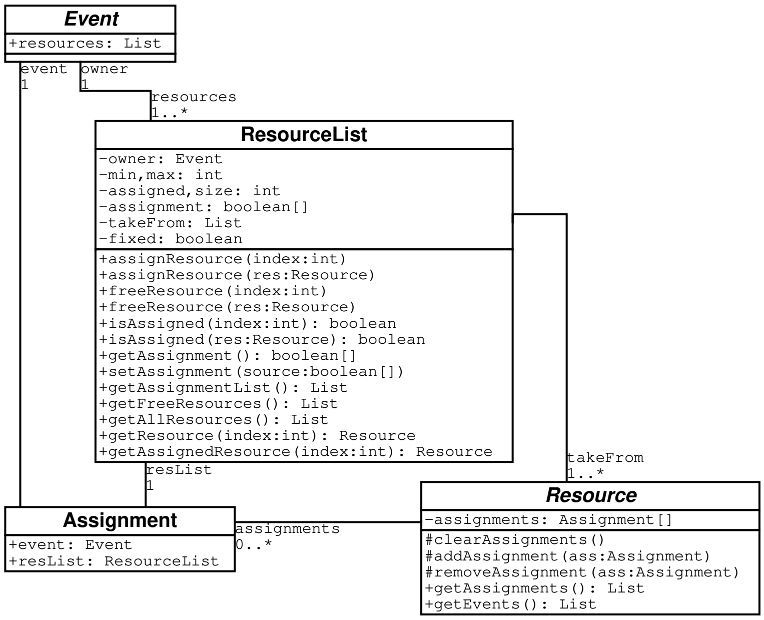


Abbildung 2.4: Struktur der Klasse `ResourceList`

Ressourcenlisten werden während der Initialisierung eines TT-Problems erzeugt und mit den notwendigen Daten versorgt. Die Liste `takeFrom`, in der alle Elemente aufgeführt sind, die dem zu einer Ressourcenliste gehörenden Ereignis theoretisch zugeordnet werden können, sowie die Werte von `min`, `max` und `fixed` bleiben ab diesem Zeitpunkt unverändert. Lediglich der Bitvektor `assignment`, in dem zu jedem Zeitpunkt die tatsächliche Zuordnung steht, wird im Laufe des Timetabling-Prozesses immer wieder verändert, bis ein zufriedenstellender Zeitplan, im besten Falle eine optimale Lösung des Problems, gefunden ist.

Dabei sind die einzelnen Komponenten des Vektors `assignment` mit der natürlichen Bedeutung versehen: Ein `true`-Eintrag in der i-ten Komponenten bedeutet, dass das i-te Element von `takeFrom` dem Ereignis zugeordnet ist, `false`, dass es nicht zugeordnet ist.

Die Tabelle 2.1 gibt eine kurze Übersicht über die Funktionen der wichtigsten Variablen dieser Klasse, die unmittelbar mit dem TT-Problem verbunden sind.

Die Bedeutung der Liste `takeFrom` wird deutlich, wenn man sich Gedanken zur Größe des Suchraums macht, der mit einem vorgegebenen Timetabling-Problem verbunden ist: Jedes zusätzliche Element in `takeFrom` verdoppelt die Anzahl der möglichen Zeitpläne. Deshalb wird man z.B.

Variablenname	Aufgabe
`List takeFrom`	Liste aller Ressourcen, die dieser Ressourcenliste (und damit dem Ereignis) theoretisch zugeordnet werden können
`int min, max`	Dieses Paar legt fest, wie viele Elemente aus `takeFrom` mindestens und wie viele höchstens zugeordnet werden. Diese Werte sind nur im Zusammenspiel mit einem entsprechenden Constraint wirksam.
`boolean[] assignment`	Der Bitvektor enthält die tatsächlich stattgefundenen Zuteilungen der Ressourcen.
`boolean fixed`	Ist `fixed = true`, dann sind die durch `min` und `max` festgelegten Werte als verbindlich zu betrachten, und das Intervall zwischen ihnen wird bei der Suche nach einer Lösung des Timetabling-Problems nie verlassen. Dies ist zum Beispiel sinnvoll, wenn man ausnahmslos genau weiß, dass pro Unterrichtsstunde genau ein Lehrer benötigt wird.

Tabelle 2.1: Variablen der Klasse `ResourceList`

in der Schulstundenplanerstellung für die Ressourcenliste „Raum“ eines Ereignisses „Sportunterricht“ nur solche Räume zulassen, in denen auch Sportunterricht stattfinden kann, und `takeFrom` bei der Erzeugung des `ResourceList`-Objekts entsprechend initialisieren.

Eine ähnliche Bedeutung kommt dem Flag `fixed` zu. Diese Variable legt fest, ob die durch `min` und `max` gesetzte unverbindliche Ober- bzw. Untergrenze für die Anzahl der zugeteilten Ressourcen verbindlich sein soll. In der Schulstundenplanerstellung ist zum Beispiel von Anfang an klar, welche Klasse zu einer bestimmten Unterrichtsstunde gehört oder dass der Unterricht in genau einem Raum stattfindet. Hier ist es sinnvoll, den Optimierungsalgorithmus von dieser Festlegung in Kenntnis zu setzen, indem das Flag `fixed` gesetzt wird, das Veränderungen am Zuteilungsvektor `assignment`, die die Grenze `min` bzw. `max` verletzen, verbietet.

Ebenso, wie die Zeitintervalle aus Sicht der Klasse `TTProblem` besonders ausgezeichnet sind, indem das Objekt der Klasse `ResourceType`, das alle Zeitintervalle zusammenfasst, den Namen „TimeSlot“ trägt, so muss auch aus Sicht eines jeden Ereignisses eine solche Auszeichnung stattfinden; es ist klar, dass die Methode `clashes` nur dann überprüfen kann, ob sich zwei Ereignisse überschneiden, wenn sie in Erfahrung bringen kann, welche Zeitintervalle dem Ereignis überhaupt zugeteilt worden sind. Deshalb muss auch hier die Instanz der Klasse `ResourceList`, die für die Zeiträume, zu denen ein Event stattfindet, verantwortlich ist, den eindeutigen Namen „TimeSlots“ tragen.

Um das Verständnis zu erleichtern, folgen zwei kurze Beispiele:

- `takeFrom` = {“Montag, 7.55 - 8.40“, “Montag, 8.45 - 9.30“},
  `min` = 2, `max` = 2, `fixed` = `true`
  Diese Konstellation könnte man zum Beispiel in einer Ressourcenliste vom Typ „TimeSlots“ innerhalb eines Schulstundenplans vorfinden. Das Ereignis, zu dem die Liste gehört, hat hier

vom Planer die beiden Zeitintervalle „Montag, 7.55 - 8.40“ und „Montag, 8.45 - 9.30“ fest zugewiesen bekommen.

- `takeFrom` = {“Bode“, “Klose“, “Jancker“},
  `min` = 1, `max` = 2, `fixed` = `false`
  Eine solche Konstellation könnte in einer Ressourcenliste vom Typ „Stürmer“ innerhalb des Ereignisses „Fußballspiel“ gefunden werden und bedeuten, dass aus einer Liste von drei Stürmern einer oder zwei ausgewählt werden sollen. Wenn in einem Stundenplan keiner oder drei eingeteilt sind, wird das entsprechend den festgelegten Constraints bestraft.

2.5 Die einzelne Ressource

Die verschiedenen Ressourcen, die in einem Timetabling-Problem auftauchen, sind Verallgemeinerungen der abstrakten Klasse `Resource`, die wiederum von `PlanningClass` abgeleitet ist und damit über bestimmte Constraints verfügen kann.

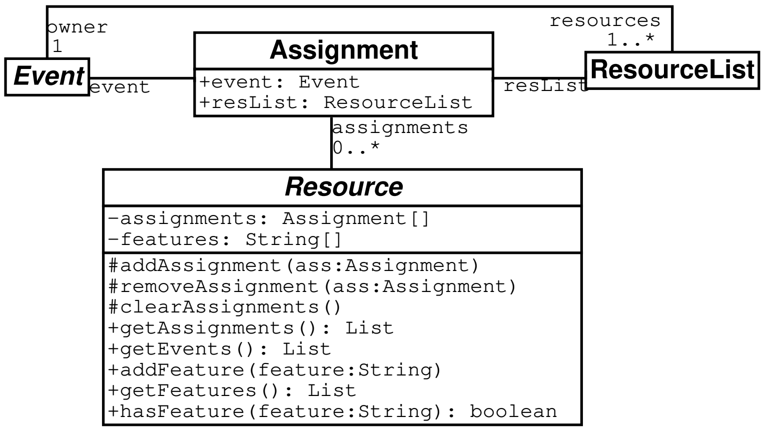


Abbildung 2.5: Ressourcenverwaltung

Wesentliche Komponente einer jeden Ressource ist der `assignments`-Vektor, in dem alle Zuordnungen einer Ressource zu verschiedenen Ereignissen bzw. Ressourcenlisten innerhalb dieser Ereignisse festgehalten werden. Über diesen Vektor kann jederzeit die zu einer Ressource gehörende Sicht, also ihr individueller Zeitplan, konstruiert werden.

Ressourcen können zusätzlich noch über bestimmte Features verfügen, die sie für die Einteilung in bestimmte Positionen qualifizieren.

2.6 Festlegen der Rahmenbedingungen

Die bei der Zuteilung der Ressourcen an die am Timetabling-Prozess beteiligten Ereignisse zu beachtenden Rahmenbedingungen werden in Objekten der Klasse `Constraint` verwaltet. Die

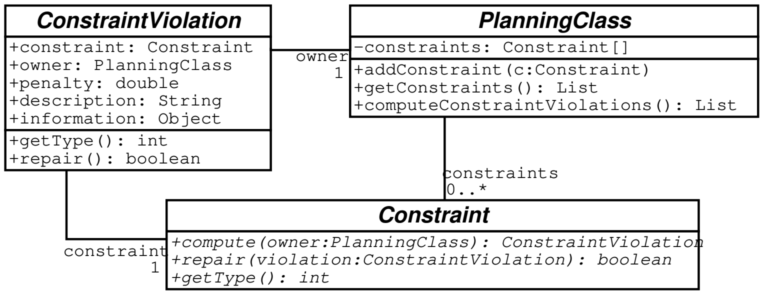


Abbildung 2.6: Repräsentation der Constraints

Struktur dieser Klasse und ihre Einbettung in die gesamte Klassenhierarchie werden in Abb. 2.6 deutlich.

Wird eine Instanz von `PlanningClass`, also eine Ressource oder ein Ereignis, durch Aufruf der Methode `computeConstraintViolations` dazu veranlasst, die Verletzungen der lokalen Bedingungen zu berechnen, so ruft sie nacheinander die Methode `compute` aller ihr zugewiesenen Constraints auf.

Die Methode `compute` der Klasse `Constraint` ermittelt, ob das jeweilige Constraint verletzt ist und liefert im Falle einer Verletzung eine neue Instanz von `ConstraintViolation` zurück. Constraint Violations tragen die der Verletzung zugeordneten Penalty Points, einen Zeiger auf das verletzte Constraint und einen Zeiger auf das Objekt, bei dem die Verletzung aufgetreten ist, mit sich. Optional können Sie noch einen Beschreibungstext (`description`) und Informationen zur Art der Constraint-Verletzung (`information`) enthalten.

Die Methoden `repair` von `Constraint` und `ConstraintViolation` können zur Reparatur von entstandenen Verletzungen mit Hilfe von problemspezifischem Wissen implementiert werden. Die Variable `information`, der bei der Erzeugung einer Constraint-Verletzung ein Wert zugewiesen werden kann, hilft der Reparaturmethode, bereits vorhandenes Wissen über die Art der Verletzung zur Behebung des Problems zu verwenden.

2.7 Ausgabe der Ergebnisse

Zur Ausgabe des Ergebnisses eines Timetabling-Prozesses stellt die Klasse `TTProblem` drei Methoden zur Verfügung:

- `getSummary()` liefert eine textuelle Beschreibung eines Zeitplans,
- `toXML()` liefert als Ergebnis eine XML-Beschreibung des `TTProblem`-Objekts, die die Struktur des erzeugten Zeitplans eindeutig festlegt und die zur späteren Weiterverarbeitung des Zeitplans, zum Beispiel zur Verbesserung des Plans durch ein anderes Verfahren, genutzt werden kann, und
- `toHTML()` erzeugt eine HTML-Beschreibung des erstellten Zeitplans, aus der man u.a. die jeweiligen Sichten der beteiligten Ressourcen direkt ablesen kann.

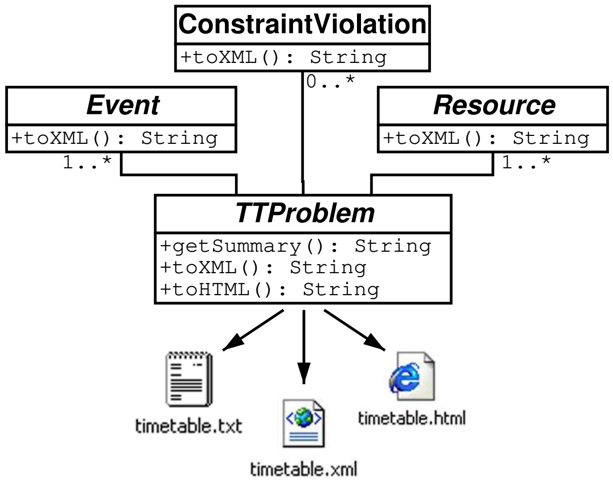


Abbildung 2.7: Ausgabe von Zeitplänen

Auf diese Weise kann ein einmal erzeugter Zeitplan sowohl automatisch als auch von einem menschlichen Planer problemlos weiterverarbeitet werden.

Die Struktur der Klartext-Ausgabe und des XML-Codes des erzeugten Zeitplans sind selbsterklärend, so dass an dieser Stelle auf eine eingehende Beschreibung verzichtet werden kann. Kurze Beispiele sind im Anhang dieser Arbeit aufgeführt.

Kapitel 3

Der Genetische Algorithmus

3.1 Einführung in Genetische Algorithmen

3.1.1 Was sind Genetische Algorithmen?

Um der Frage, was Genetische Algorithmen (GAs) eigentlich sind, nachzugehen, ist es nicht unüblich, David Goldberg zu zitieren, der in [Gol89] den Grundstein für die Verwendung Genetischer Algorithmen in Suchverfahren gelegt hat:

> Genetic algorithms are search algorithms based on the mechanics of natural selection and natural genetics. They combine survival of the fittest among string structures with a structured yet randomized information exchange to form a search algorithm with some of the innovative flair of human search. In every generation, a new set of artificial creatures (strings) is created using bits and pieces of the fittest of the old; an occasional new part is tried for good measure. While randomized, genetic algorithms are no simple random walk. They efficiently exploit historical information to speculate on new search points with expected improved performance.

Mit GAs versucht man, einen in der Natur stattfindenden Evolutionsablauf zu simulieren, indem man evolutionäre Techniken wie Mutation und Fortpflanzung von Individuen anwendet und so aus einer in der Regel zufällig erzeugten Initialpopulation immer neuere, und hoffentlich bessere (im Sinne einer problemabhängigen Fitness-Funktion), Generationen erzeugen lässt.

Die den Genetischen Algorithmen zugrundeliegende Idee ist die Hoffnung, dass sich die zuerst von Charles Darwin in [Dar59] geäußerte These der natürlichen Selektion auf künstliche Umgebungen im Speicher eines Rechners übertragen lässt. Die Darwinsche Theorie lässt sich folgendermaßen zusammenfassen:

- Eltern geben ihre Eigenschaften teilweise an ihre Nachkommen weiter.

- Individuen mit besseren Eigenschaften haben tendenziell mehr Nachkommen als solche mit schlechteren Eigenschaften.
- Auf lange Sicht werden sich bessere Eigenschaften gegenüber schlechteren durchsetzen: „Survival of the Fittest“. Dadurch entsteht eine insgesamt stärkere Population.

Die Frage nach der Bedeutung von „besser“ und „schlechter“ wird im Zusammenhang mit Genetischen Algorithmen durch eine künstliche Fitnessfunktion beantwortet, die jedem Individuum einen in der Regel eindeutigen Fitness-Wert zuweist, der die einzelnen Individuen einer Population vergleichbar werden lässt.

Goldberg, der sich in seiner Arbeit auf die Theorie Hollands (z.B. [Hol75]) stützt, bezieht sich bei seiner Definition des Begriffs „Genetischer Algorithmus“ in [Gol89] nur auf Algorithmen, die direkt auf Bitvektoren arbeiten. Es ist klar, dass sich jedes Individuum, also z.B. eine Lösung eines Suchproblems, als purer Bitvektor codieren lässt. Das führt jedoch in vielen Fällen zu einem erhöhten Aufwand in den genetischen Operatoren wie dem Mutationsoperator, da häufig Individuen erzeugt werden, die nicht nur schlechte, sondern schlicht unsinnige Lösungen darstellen.

Daher verzichtet man häufig darauf, die reine Bitstring-Repräsentation zu verwenden, und lässt die genetischen Operatoren statt dessen direkt auf den natürlichen Strukturen arbeiten. Als Konsequenz hat sich der Sprachgebrauch in den letzten 10 Jahren gewandelt, und der Begriff „Genetischer Algorithmus“ ist inzwischen unabhängig von der Darstellung der Individuen als Bitstrings.

Die Freiheit bei der Wahl einer Fitness-Funktion zur Bewertung der Individuen erlaubt es, Genetische Algorithmen in vielen verschiedenen Bereichen einzusetzen. Wählt man eine geeignete Fitness-Funktion, so lassen sie sich als Optimierungsverfahren für eine große Klasse von Problemen verwenden. Ein berühmtes Beispiel ist hier das Problem des Handelsreisenden (Travelling Salesperson Problem, TSP) : Codiert man in jedem Individuum eine Permutation der zu bereisenden Städte und berechnet dann die Fitness einer solchen Permutation als die Summe der euklidischen Abstände der einzelnen Orte, dann ergibt sich sofort ein Such- bzw. Optimierungsverfahren für das TSP.

Der Vorteil Genetischer Algorithmen gegenüber herkömmlichen Suchverfahren, z.B. Hill-Climbing-Algorithmen, liegt in mehreren Eigenschaften begründet:

- GAs suchen von einer Menge (Population) von potentiellen Lösungen ausgehend, nicht nur von einer einzigen.
- GAs verwenden zur Suche ausschließlich eine Fitness-Funktion, durch die festgelegt ist, welche Lösungen gut und welche schlecht sind. Es werden keine Anforderungen an die Differenzierbarkeit dieser Funktion gestellt.
- Ein GA kann mit minimalem Zusatzaufwand zur Lösung grundverschiedener Suchprobleme verwendet werden, ohne dass der Kern des Algorithmus angetastet wird.

3.1.2 Terminologie

Da mit Genetischen Algorithmen ein Stück Natur im Rechner simuliert wird, bedient man sich sinnvollerweise des Vokabulars, das die Biologie in diesem Zusammenhang zur Verfügung stellt. Um Unklarheiten zu vermeiden, werden an dieser Stelle die wichtigsten Begriffe mit einer kurzen Erklärung ihrer biologischen Bedeutung und ihrer Verwendung im Kontext Genetischer Algorithmen eingeführt:

Individuum Eine einzelne Lebensform.
Im Zusammenhang mit Genetischen Algorithmen wählt man häufig eine abstraktere Sicht und bezeichnet mit „Individuum" einfach eine Codierung von Erbanlagen, also eine Menge von Chromosomen.

Chromosomen Stukturen im Zellkern tierischer und pflanzlicher Zellen. Sie sind Träger der genetischen Information. Bei den höheren Lebewesen sind sind Chromosomen in allen Körperzellen in doppelter Ausführung vorhanden. Je ein Chromosomensatz stammt aus der mütterlichen bzw. väterlichen Geschlechtszelle. Der einfache (*haploide*) Satz wird bei der Keimzellenreifung durch Reduktion des doppelten (*diploiden*) Satzes hergestellt. Die diploide Chromosomenzahl schwankt zwischen 2 und 100. Der Mensch besitzt 46.
In GAs zum Einsatz kommende künstliche Lebensformen verfügen in der Regel nur über ein einziges Chromosom. Daher werden die Begriffe „Chromosom" und „Individuum" oftmals synonym gebraucht.

Gene Erbfaktoren. Ein vom Botaniker W. L. Johannsen geprägter Begriff für Teile der in den Chromosomen lokalisierten Erbinformation. Gene sind frei kombinierbar und damit Einheiten der Rekombination, lassen sich (natürlich und künstlich) verändern und bewirken die Ausbildung jeweils eines bestimmten Merkmals. Die Gesamtheit der Gene eines Individuums heißt *Genotyp*.
Die freie Kombinierbarkeit der Gene ist dann eingeschränkt, wenn die betreffenden Gene innerhalb des selben Chromosoms liegen. Dann besteht jedoch Austauschmöglichkeit durch *Crossover*.
Welche Struktur innerhalb eines GAs als Gen betrachtet wird, ist nicht ganz klar und hängt von der Implementierung ab. Häufig ist jedes einzelne Bit des Genoms bereits ein Gen.

Allele Gene, die auf homologen Chromosomen zweier oder mehrerer Chromosomensätze den gleichen Genort besetzen. Sie bewirken eine unterschiedliche Ausbildung des zugehörigen Erbmerkmals.

Crossover (auch: Rekombination) Austausch genetischer Information zwischen zwei homologen Chromosomen durch die Überkreuzung der beiden Chromatiden-Stränge. Auf diese Weise werden Erbanlagen neu kombiniert.

Mutation Von H. de Vries geprägter Begriff für eine sprunghafte Änderung im Erbgut eines Organismus, die zu phänotypisch oder funktionell abweichenden Nachkommen führt. Man unterscheidet zwischen verschiedenen Arten der Mutation:

- Veränderungen der Zahl der Chromosomen

- Veränderung größerer Chromosomenabschnitte
- Veränderungen in Genen (Genmutation)

In Genetischen Algorithmen wird die Mutation oft durch eine einfache Inversion einer zufälligen Menge von Bits in der Bitstring-Repräsentation eines Individuums realisiert.

3.1.3 Prinzipielle Ablaufstruktur

Betrachtet man den Ablauf eines Genetischen Algorithmus, ohne näher auf die verschiedenen genetischen Operatoren einzugehen, so muss man zwischen zwei prinzipiellen Grundmustern unterscheiden:

- Der erste Ansatz ist der klassische GA und wurde zuerst von Holland in [Hol75] vorgestellt. Bei dieser Methode wird in jedem Schritt des Genetischen Algorithmus die gesamte Population durch neue Individuen, die durch Crossover oder Mutation aus der alten Population erzeugt worden sind, ersetzt. Es entsteht so eine Abfolge disjunkter Populationen, die man Generationen nennt. Deshalb trägt diese Spielart Genetischer Algorithmen auch den Namen „Generational GA".
- Der zweite Ansatz stammt von Whitley ([Whi89]) und wird mit „Steady-State GA" bezeichnet. Hier wird aus zwei aus der aktuellen Population ausgewählten Individuen ein Nachkomme erzeugt und ggf. einer Mutation unterzogen. Ist dieses neu erzeugte Individuum besser als das schlechteste Individuum der aktuellen Population, so ersetzt es dieses. Anderenfalls geschieht nichts.

Obwohl der zweite Ansatz alles andere als intuitiv ist, weil er sich überhaupt nicht am natürlichen Vorbild orientiert, braucht er in vielen Problemfeldern sich hinsichtlich der erreichten Ergebnisse nicht hinter dem klassischen GA zu verstecken. Letztlich ist die Frage, welcher der beiden Ansätze erfolgversprechender ist, aber stark vom zu optimierenden Problem abhängig.

Der Steady-State GA, dessen Bezeichnung nicht von seinem Erfinder Whitley, sondern von Syswerda stammt, sammelt während eines Evolutionsvorgangs die besten n Individuen, denen er bisher begegnet ist, in der Population. Durch das sofortige Ersetzen schlechter Individuen bei Auffinden einer besseren Lösung ist der Selektionsdruck beim Steady-State GA wesentlich größer als beim kanonischen Generational GA. Er zeichnet sich daher durch eine schnelle, aber unter Umständen zu frühe Konvergenz aus. Eine vergleichende Analyse beider Ansätze findet sich in der Arbeit von Syswerda ([Sys91]).

3.1.4 Selektionsoperatoren

Nachdem der prinzipielle Ablauf eines Evolutionsvorgangs geklärt ist, stellt sich die Frage nach der Auswahl der Eltern, aus denen durch Paarung ein neues Individuum erzeugt werden soll. Es existiert eine ganze Reihe verschiedener Selektionsalgorithmen, die zum Teil sehr speziell auf bestimmte Problemklassen zugeschnitten sind. Eine Übersicht bieten zum Beispiel Blickle und Thiele ([BT95a]) oder Goldberg und Deb ([GD91]).

Die gängigsten Selektionsverfahren werden an dieser Stelle kurz vorgestellt:

Fitness-proportionale Selektion (Fitness-proportional Selection) ist die kanonische Methode, Individuen auszuwählen. Dabei ist die Wahrscheinlichkeit, an einem Paarungsvorgang beteiligt zu sein, für jedes Individuum direkt proportional zu seinem Fitness-Wert. Diese Vorgehensweise ist problematisch bzw. unbrauchbar, wenn die Fitness-Werte aller Individuen einer Population relativ zur durchschnittlichen Fitness nah beieinander liegen oder innerhalb der Population Individuen mit negativem Fitness-Wert auftauchen. In solchen Fällen müssen Skalierungsmethoden eingesetzt werden, die die Fitness-Werte der Population in geeigneter Weise transformieren. Eine Möglichkeit der Transformation wird in [Gol89, S. 76-79] vorgestellt.

Rang-basierte Selektion (Rank-based Selection) wurde zuerst im Jahr 1985 von Baker in [Bak85] vorgestellt. Sie erlaubt es jedem Individuum mit einer Wahrscheinlichkeit, die proportional zu seinem Rang innerhalb der Population ist, an einer Paarungsprozedur teilzunehmen. Ist die Populationsgröße n, so hat zum Beispiel das beste Individuum eine Selektionswahrscheinlichkeit

$$p(n) = \frac{n}{\sum_{k=1}^{n} k} = \frac{2n}{n(n+1)},$$

und das schlechteste wird mit der Wahrscheinlichkeit

$$p(1) = \frac{2}{n(n+1)}.$$

ausgewählt.

Turnier-Selektion (Tournament Selection), die zum ersten Mal in [Bri81] beschrieben wurde, ist die einfachste der hier vorgestellten Auswahlmethoden: Es werden zufällig t Individuen aus der Population entnommen (Gleichverteilung). Das beste dieser t Individuen wird dann für einen Reproduktionsvorgang herangezogen.

Wegen der Skalierungsprobleme, die sich im Fall der Fitness-proportionalen Selektion im Laufe des Evolutionsprozesses ergeben, ist es häufig einfacher, Rang-basierte Selektion oder die Turnier-Methode zu verwenden (die für $t = 2$ übrigens identisch ist mit der Rang-basierten Auswahl).

3.1.5 Crossover

Nachdem zwei Elternteile für die Erzeugung von Nachkommen mit einer der vorgestellten Methoden ausgewählt worden sind, stellt sich die Frage, auf welche Weise aus diesen zwei Individuen neue Individuen konstruiert werden. In anderen Worten: Es geht darum, mit welchen Methoden das Crossover, also die Rekombination der elterlichen Erbanlagen, durchgeführt wird. Dazu bestehe in diesem Abschnitt ein Individuum immer aus einem einzelnen Chromosom, das ein Bitvektor der Länge n, also ein Punkt im Vektorraum $\mathcal{B}^n$ ist. Im Sinne der obigen Begriffsdefinition ist dann weiter ein Gen genau eine Komponente dieses Vektors. Die Eltern seien dann die Vektoren $\vec{b}_V := (b_{V,0}, b_{V,1}, ..., b_{V,n-1})$ (Vater) und $\vec{b}_M := (b_{M,0}, b_{M,1}, ..., b_{M,n-1})$ (Mutter).

In der Geschichte der Genetischen Algorithmen sind im Wesentlichen drei verschiedene Arten des Crossover zum Einsatz gekommen, die hier kurz vorgestellt werden:

One-Point Crossover Es wird ein Crossover-Punkt $0 \leq p < n-1$ zufällig gewählt. Es ergeben sich dann zwei Kinder als

$$\vec{b}_S := (b_{V,0}, ..., b_{V,p}, b_{M,p+1}, ..., b_{M,n-1})$$

und

$$\vec{b}_T := (b_{M,0}, ..., b_{M,p}, b_{V,p+1}, ..., b_{V,n-1}).$$

Two-Point Crossover Es werden zwei Crossover-Punkte $0 \leq p_1 < p_2 < n-1$ zufällig gewählt. Dann entstehen aus dem Crossover-Prozess wiederum zwei Nachkommen

$$\vec{b}_S := (b_{V,0}, ..., b_{V,p_1}, b_{M,p_1+1}, ..., b_{M,p_2}, b_{V,p_2+1}, ...b_{V,n-1})$$

und

$$\vec{b}_T := (b_{M,0}, ..., b_{M,p_1}, b_{V,p_1+1}, ..., b_{V,p_2}, b_{M,p_2+1}, ...b_{M,n-1}).$$

Uniform Crossover Bei jeder Komponenten des Kind-Vektors wird zufällig gewählt, ob der Wert (das Allel) vom Vater oder von der Mutter übernommen werden soll. Die Kinder ergeben sich dann zum Beispiel

$$\vec{b}_S := (b_{V,0}, b_{V,1}, b_{M,2}, b_{V,3}, b_{M,4}, ..., b_{V,n-1})$$

und

$$\vec{b}_S := (b_{M,0}, b_{M,1}, b_{V,2}, b_{M,3}, b_{V,4}, ..., b_{M,n-1}).$$

Abb. 3.1 veranschaulicht beispielhaft die Vorgehensweise bei One-Point und Uniform Crossover. Ein Two-Point Crossover läuft in ähnlicher Weise ab. Gänzlich analog zum One- und Two-Point Crossover lässt sich das Multi-Point Crossover mit einer fest vorgegebenen Anzahl c von Crossover-Punkten erklären.

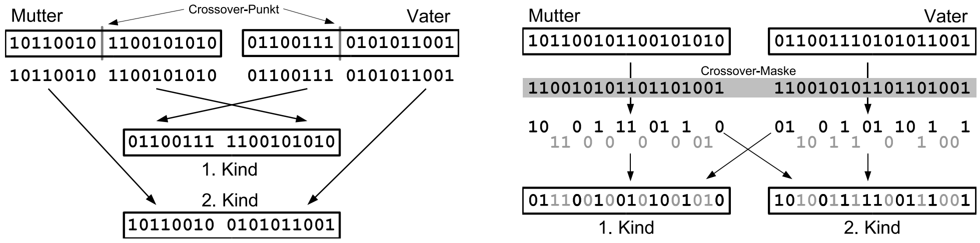


Abbildung 3.1: One-Point und Uniform Crossover

Eine ausführlichere Beschreibung und eine eingehende Analyse der verschiedenen Crossover-Operatoren findet man in [Sys89].

3.1.6 Mutation

Die Betrachtung des Mutationsoperators beschränkt sich an dieser Stelle wiederum auf Chromosomen, die Punkte im Vektorraum $\mathcal{B}^n$ sind. Der in den meisten Implementierungen Genetischer Algorithmen zum Einsatz kommene Mutationsoperator ist dann sehr einfach.

Gegeben sei eine Wahrscheinlichkeit μ, die sog. Mutationsrate. Dann gewinnt man das durch Mutation aus einem vorhandenen Chromosom entstehende neue Chromosom durch folgenden Algorithmus:

```
for (j = 0; j < n; j++)
  if (random.nextDouble() < μ)
    newChromosome[j] = !oldChromosome[j];
  else
    newChromosome[j] = oldChromosome[j];
```

Abbildung 3.2: Java-Quelltext eines einfachen Mutationsoperators

Verwendet man diesen Mutationsoperator, so ist aus Sicht des Genetischen Algorithmus jedes einzelne Bit in der Binärdarstellung eines Individuums ein eigenständiges Gen. Es ist klar, dass diese Vorgehensweise bei bestimmten Problemen nicht sinnvoll ist. Auf diese Problematik wird in Abschnitt 3.5 näher eingegangen werden.

3.2 Klassenstruktur des Genetischen Algorithmus

Der im Rahmen dieser Arbeit implementierte GA besteht aus der Klasse `GeneticAlgorithm` und der abstrakten Klasse `GeneticIndividual`.

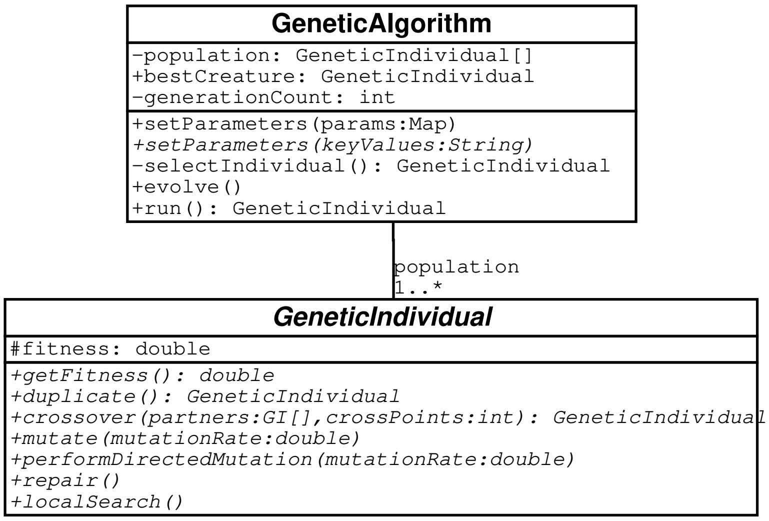


Abbildung 3.3: Klassenstruktur des Genetischen Algorithmus

Der Genetische Algorithmus und die notwendigen Kontrollstrukturen befinden sich in der Klasse `GeneticAlgorithm`, während sich die Population, auf der der GA arbeitet, aus Instanzen der Klasse `GeneticIndividual` zusammensetzt.

Neben den klassischen genetischen Operatoren Mutation und Crossover stellt jedes genetische Individuum mit den Methoden `performDirectedMutation`, `repair`, `climb` und `localSearch` auch hybride Operatoren zur Verfügung, die im Zusammenhang mit der Evolution von Zeitplänen in Abschnitt 3.6 erklärt werden.

3.3 Ablauf eines Evolutionsvorgangs

Der Genetische Algorithmus unterstützt zunächst die in Abschnitt 3.1 vorgestellten prinzipiell verschiedenen Arbeitsweisen Steady-State und Generational.

Beim Generational GA wird in jedem Evolutionsschritt die aktuelle Population komplett (bzw. teilweise, z.B. wenn durch Anwendung des Elite-Operators ein bestimmter Anteil der aktuellen Population als `Elite` direkt in die nächste übernommen wird) durch neue, durch Crossover oder Mutation erzeugte Individuen ersetzt wird, während beim Steady-State GA in jedem Schritt nur jeweils ein Individuum durch Crossover oder Mutation neu erzeugt wird und, falls es besser als das schlechteste Individuum der aktuellen Population ist, dieses ersetzt.

Der Ablauf eines Evolutionsvorgangs für beide Arbeitsweisen ist ist in Abb. 3.4 als Struktogramm dargestellt.

Ein dritter Arbeitsmodus, in dem weitere Operatoren, zum Beispiel ein Reparatur-Operator, der durch Einsatz problemspezifischen Wissens gezielt bestimmte Constraint-Verletzungen repariert, ein Hill-Climbing-Operator oder exhaustive lokale Suche, den Genetischen Algorithmus zu einem hybriden Verfahren erweitern, wird in Abschnitt 3.6 besprochen.

Sind diese zusätzlichen Operatoren geeignet gewählt und geschickt implementiert, so kann dies sowohl die Geschwindigkeit des Genetischen Algorithmus als auch die Qualität der erzeugten Populationen dramatisch verbessern.

3.4 Stundenpläne als Chromosomen: Anwendung auf Timetabling-Probleme

Damit ein durch die Klasse `TTProblem` beschriebenes Timetabling-Problem durch den Genetischen Algorithmus gelöst bzw. optimiert werden kann, muss `TTProblem` die Klasse `GeneticIndividual` beerben. Andererseits ist es wünschenswert, dass ein gegebenes Problem durch verschiedene Verfahren optimiert werden kann. Die Abb. 3.5, die das Verhältnis von `TTProblem` und `GeneticAlgorithm` beschreibt, erfüllt diese Anforderungen.

In der Einleitung dieses Kapitels ist behauptet worden, jedes künstliche Individuum lasse sich als reiner Bitvektor darstellen. Diese Aussage ist trivial und muss nicht bewiesen werden. Es ist jedoch interessant, den Aufbau dieses Bitvektors im Fall des Zeitplans etwas genauer zu untersuchen.

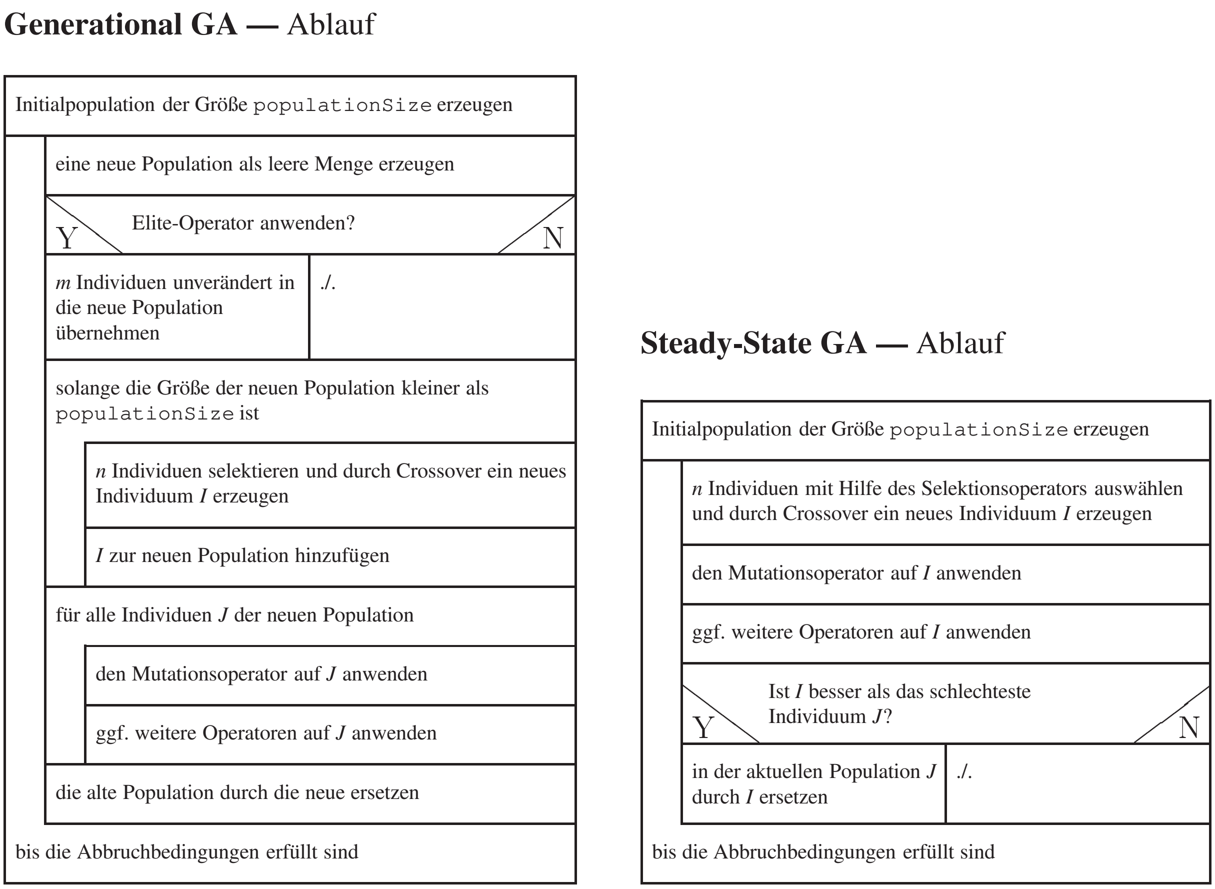


Abbildung 3.4: Struktogramm für den Ablauf des Evolutionsvorgangs

Wie in Kapitel 2 dargestellt, besteht ein Zeitplan aus einer geordneten Menge von Ereignissen. Jedes Ereignis wiederum setzt sich im Wesentlichen aus einer geordneten Menge von Ressourcenlisten zusammen. In diesem Ressourcenlisten entscheidet ein Bitvektor über die tatsächliche Zuteilung der verschiedenen Ressourcen.

Wenn man nun mit $\{\}$ den Bitvektor der Länge 0 und mit

$$\circ : \mathcal{B}^m \times \mathcal{B}^n \to \mathcal{B}^{m+n};\ (\vec{b}_1, \vec{b}_2) \mapsto \vec{b}_1 \circ \vec{b}_2$$

die Konkatenation zweier Bitvektoren bezeichnet, so lässt sich die Binärdarstellung eines Zeitplans einfach berechnen. Einen Algorithmus findet man in Abb. 3.6.

Theoretisch wäre es nun möglich, die in Abschnitt 3.1 eingeführten Crossover- und Mutationsoperatoren einzusetzen und auf diese Weise einen Evolutionsprozess durchzuführen. Dass dies nicht geschieht, hat bestimmte Gründe, auf die im nächsten Abschnitt detailliert eingegangen wird.

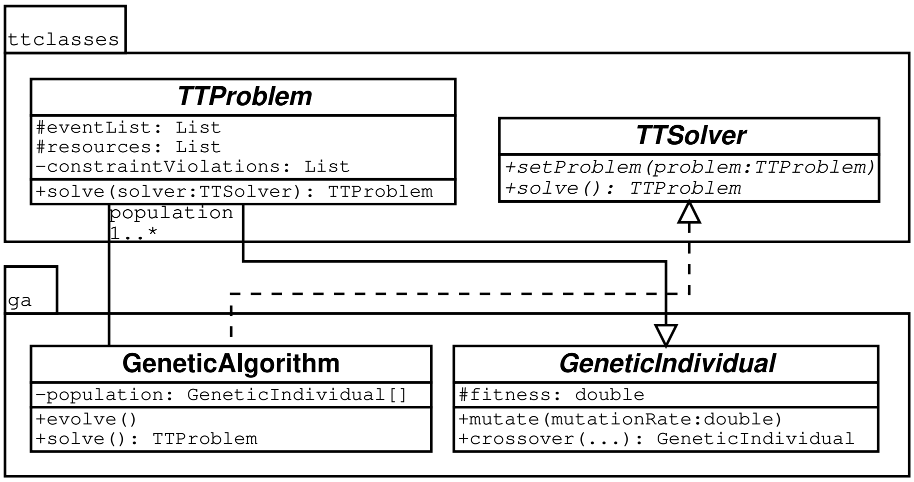


Abbildung 3.5: Verhältnis von `TTProblem` und `GeneticAlgorithm`

```
bitString = {};
foreach Event e in eventList
  foreach ResourceList rl in e.resources
    bitString = bitString ∘ rl.getAssignment();
```

Abbildung 3.6: Berechnung der Binärdarstellung eines Zeitplans

3.5 Mutation und Crossover bei Zeitplänen

Ein Zeitplan, der als Genetisches Individuum zur Evolution herangezogen werden kann, muss, wie besprochen, zunächst Methoden zur Durchführung von Mutation und Crossover zur Verfügung stellen. Diese Operationen werden jedoch in dieser Implementierung nicht auf der Binärdarstellung durchgeführt, sondern direkt auf den in Kapitel 2 vorgestellten Strukturen.

3.5.1 Mutation

Der in Abschnitt 3.1 vorgestellte Ansatz, Mutation einfach durch Umkippen zufällig gewählter Bits in der Binärdarstellung eines Individuums zu realisieren, führt in vielen Anwendungsbereichen zu brauchbaren Ergebnissen, ist im Fall des allgemeinen Timetabling-Problems jedoch zum Scheitern verurteilt. Dies kann das folgende Szenario verdeutlichen:

Es soll ein Schulstundenplan erstellt werden. Dabei sind die Unterrichtsstunden der fünften und sechsten Klassen der Schule auf 30 verschiedene Timeslots (Montag bis Freitag, jeweils 6 Stunden) zu verteilen. Jede Unterrichtsstunde dieser Klassen findet in einem von 30 Räumen statt. Die Struktur eines Ereignisses sieht dann beispielsweise so aus:

Das Ereignis „Geschichte/6b“ findet montags in der 5. und donnerstags in der 4. Stunde, jeweils im Raum Nr. 13, statt. Klasse und Lehrer dieses Ereignisses sind sinnvollerweise fest. In der Binärdarstellung des Stundenplans taucht dann also an irgendeiner Stelle, nämlich dort, wo sich das Ereig-

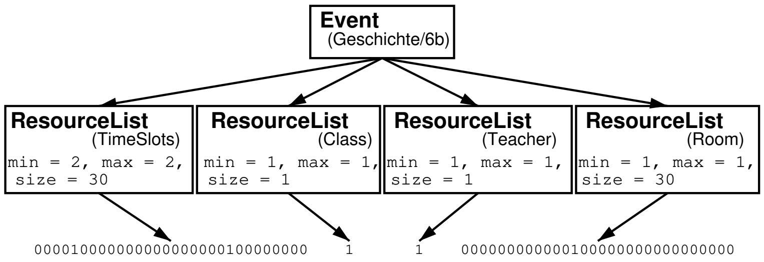


Abbildung 3.7: Struktur eines Schulstundenplans

nis „Geschichte/6b“ befindet, die Bitkette

00001000000000000000001000000011000000000000100000000000000000

auf. Wenn nun der einfache Mutationsoperator angewandt wird, der zufällig ausgewählte Bits invertiert, dann wird in fast allen Fällen ein unsinniger Stundenplan entstehen, z.B.

00001000000001000000001000000011000000000000100000100000000000

mit 3 Stunden pro Woche in zwei verschiedenen Räumen oder

00000000000000000000001000000010000000000000100000000001000000

mit einer Stunde in zwei Räumen und dafür ohne Lehrer. Dies ist ein prinzipielles Problem, das immer dann auftritt, wenn die Anzahl von Nullen und Einsen in der Binärdarstellung eines Zeitplans stark unterschiedlich ist.

Um dieses Abdriften durch einen ungeeignete Mutationsoperator zu vermeiden, wird die Mutation direkt auf den Strukturen des Timetabling-Problems durchgeführt.

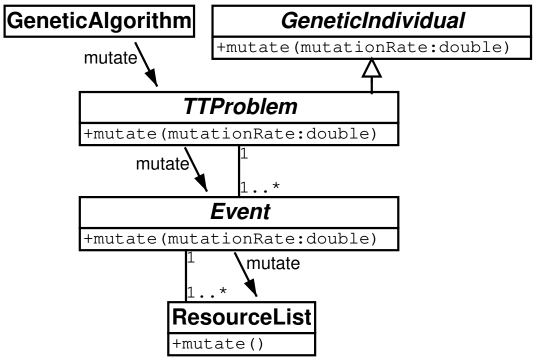


Abbildung 3.8: Mutation eines Zeitplans

Abbildung 3.8 verdeutlicht, dass effektive Änderungen am Zeitplan nur innerhalb der Klasse `ResourceList` stattfinden.

Der beim Aufruf `mutate` von `TTProblem` übergebene Parameter `mutationRate` wird von dort direkt an die einzelnen Ereignisse (Instanzen von `Event`) weitergegeben. Dort wird schließlich für jede einzelne Ressourcenliste der Zufallsgenerator befragt, ob sich innerhalb der Liste eine Mutation ereignen soll:

```
for (i = 0; i < resources.size(); i++)
  if (random.nextDouble() < mutationRate)
    ((ResourceList)resources.get(i)).mutate();
```

Hat der Zufallsgenerator entschieden, dass der Mutationsoperator auf eine Ressourcenliste anzuwenden ist, können innerhalb der Klasse `ResourceList` zwei verschiedene Basis-Operationen durchgeführt werden:

1. Eine momentan nicht zugeteilte Ressource wird zufällig ausgewählt und dem Ereignis zugeteilt, die entsprechende Komponente im Vektor `assignment` auf 1 gesetzt (ASSIGN).
2. Eine momentan zugeteilte Ressource wird zufällig ausgewählt und freigegegen, die entsprechende Komponente in `assignment` auf 0 gesetzt (FREE).

Bei beiden Operationen wird durch Aufruf der Methode `addAssignment` bzw. `removeAssignment` der Klasse `Resource` automatisch die Konsistenz des Zeitplans erhalten.

Beim Einsatz der beiden Basis-Operationen innerhalb einer Ressourcenlist sind zwei Fälle zu unterscheiden:

- Die Ressourcenliste wurde fixiert (`fixed=true`): In diesem Fall darf das Hamminggewicht des Bitvektors `assignment` nicht verändert werden. Konsequenterweise besteht die Mutation in diesem Fall aus der Nacheinanderausführung von FREE und ASSIGN.
- Die Ressourcenliste ist frei (`fixed=false`): Dann wird zufällig eine der Operationen ASSIGN, FREE gewählt und ausgeführt.

Möglich wäre auch, pro Anwendung des Mutationsoperators statt einer Basis-Operation mehrere durchzuführen. Tatsächlich wirkt sich diese Vorgehensweise am Anfang eines Evolutionsprozesses merkbar auf die Konvergenzgeschwindigkeit aus. Es hat sich jedoch gezeigt, dass die simultane Durchführung mehrerer Basis-Operationen am Ende der Evolution, wenn sich die Population in der Nähe eines Optimums befindet, zum Überspringen von Peaks führt, was nicht gewünscht ist. Daher empfiehlt es sich in den meisten Fällen, auf die bessere Konvergenz zu Beginn der Evolution zu verzichten und nur eine einzige Basis-Operation (bzw. zwei Operationen bei einer fixierten Ressourcenliste) durchzuführen.

3.5.2 Crossover

Ähnlich wie bei der Mutation eines Zeitplans hat es sich auch beim Crossover herausgestellt, dass die in 3.1 präsentierten Crossover-Methoden, die direkt auf Bitvektoren arbeiten, für Timetabling-Probleme nicht geeignet sind. Dabei tauchen die Probleme nur im Falle fixierter Ressourcenlisten auf.

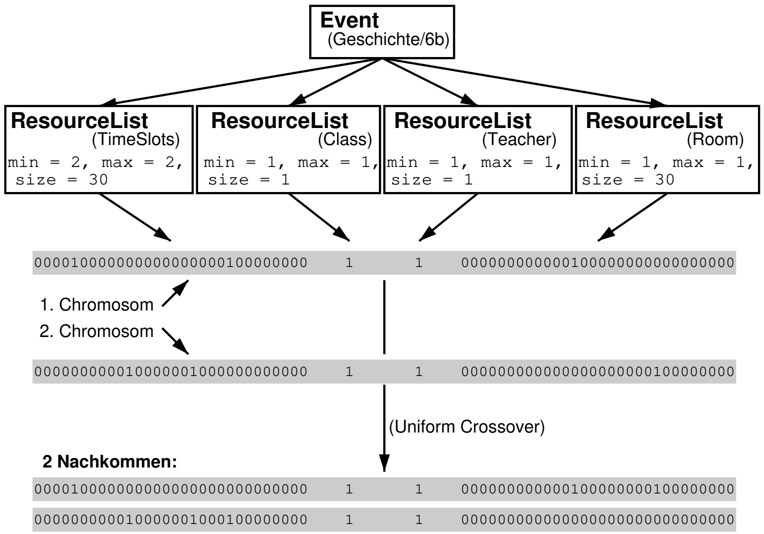


Abbildung 3.9: Uniform Crossover mit zwei Ereignissen

Wie man in Abbildung 3.9 sieht, kommt es durch Anwendung von Uniform Crossover auf die Bitvektoren zu Verletzungen der Minimum-/Maximum-Bedingungen der Ressourcenlisten „TimeSlots“ und „Room“. Das ist nicht nur in diesem speziellen Fall so, sondern es lässt sich zeigen, dass es im Allgemeinen in etwa 50% aller Fälle zu solchen Verletzungen kommt.

Dies ist keine charakteristische Eigenschaft des uniformen Crossover-Operators, sondern es gilt, jedenfalls in einer Umgebung der Crossover-Punkte, ebenso für One- und Multi-Point Crossover. Eine Möglichkeit, diesem Phänomen zu begegnen, ist, nach jeder Rekombination eine Reparatur durchzuführen, nach der die Min-/Max-Bedingungen aller fixierten Ressourcenlisten wieder erfüllt sind. Da dies mit hohem Rechenaufwand verbunden ist zudem dem Sinn des Crossovers widerspricht, ist es angebracht, nach einer Möglichkeit des Crossovers zu suchen, bei der die beschriebenen Verletzungen gar nicht erst auftreten.

Die Lösung ist, jede Ressourcenliste als Gen zu betrachten. Sie ist damit unteilbar, und der `assignment`-Vektor einer solchen Liste kann nun durch einen Crossover-Vorgang nicht mehr zerrissen werden. Das Hamminggewicht behält in jedem Fall den gewünschten Wert.

Probleme ergeben sich durch dieses Verfahren, wenn man mit Zeitplänen arbeitet, die nur aus sehr wenigen Ereignissen mit sehr großen Ressourcenlisten bestehen. Hier wird die relative Crossover-Granularität so groß, dass nur noch ein sehr kleiner Teil des Suchraums durch Crossover erreicht werden kann. Daher ist ggf. bei der Definition eines Timetabling-Problems mit der Beschreibungssprache GTL darauf zu achten, überall dort, wo es möglich ist, ein großes Ereignis in mehrere kleine aufzuspalten.

Der angepasste Crossover-Vorgang mit kompletten Ressourcenlisten als Genen auf dem Chromosom `TTProblem` wird in der Abbildung 3.10 für einen Zeitplan mit 3 Ereignissen, die jeweils über vier verschiedene Ressourcenlisten verfügen, dargestellt. Die Abbildung zeigt ein uniformes Crossover. One- und Multi-Point-Crossover lassen sich in analoger Weise implementieren, indem die Crossing-Punkte jeweils nur zwischen zwei Gene (Ressourcenlisten) gelegt werden, so dass jedes Allel als Ganzes erhalten bleibt.

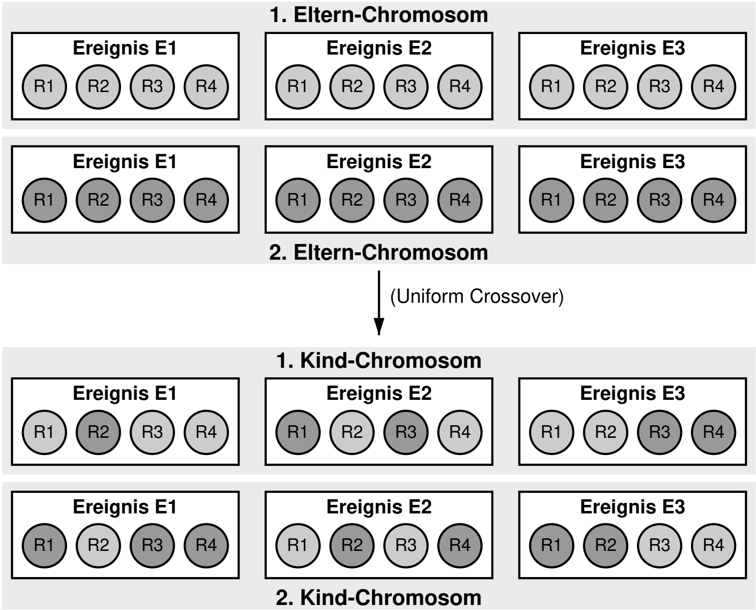


Abbildung 3.10: Crossover mit Ereignissen als Genen

3.6 Weiterführende Operatoren

Genetische Algorithmen zeichnen sich dadurch aus, dass sie, ohne außer der Berechnungsvorschrift für eine Fitness-Funktion irgendein Wissen vom zu bearbeitenden Problem zu haben, für viele verschiedene Problemklassen recht gute Ergebnisse liefern. Oft können die durch Mutation, Selektion und Crossover bereitgestellten Optimierungsmöglichkeiten jedoch den Ansprüchen der Anwender nicht genügen. In solchen Fällen wird der zum Einsatz kommende Genetische Algorithmus durch Hinzufügen von nicht-genetischen Operatoren hybridisiert. Auf diese Weise ist es zum Beispiel möglich, problemspezifisches Wissen mit in den Optimierungsprozess einfließen zu lassen.

In der Schulstundenplanerstellung kann so ein alter Hase (bzw. Lehrer), der jahrelange Erfahrung im Erstellen von Stundenplänen von Hand hat, sein gesammeltes Wissen in die computergestützte Planung einbringen, indem er dem Planungsalgorithmus beispielsweise mitteilt, auf welche Weise sich zeitliche Kollisionen in Lehrerstundenplänen erfahrungsgemäß am besten auflösen lassen.

3.6.1 Der Reparatur-Operator

Für das Auflösen festgestellter Constraint-Verletzungen ist ein Reparatur-Operator zuständig. Wie man in der Abbildung 3.3 sehen kann, stellt jede Instanz von `GeneticIndividual` eine Methode `repair` bereit, durch die die Reparatur von Verletzungen durchgeführt wird.

Der im Genetischen Algorithmus festgelegte Wert des Parameters `repairRate` bestimmt hier, welcher Anteil der innerhalb eines Individuums festgestellten Constraint-Verletzungen repariert werden soll. Dieser Wert sollte nicht zu groß gewählt werden, da zum einen Reparatur-Operationen recht aufwändig sind, zum anderen nicht auszuschließen ist, dass die Reparatur einer Constraint-Verletzung eine andere Verletzung hervorruft.

3.6.2 Hill-Climbing

Der Begriff „Hill-Climbing“ entspringt der Vorstellung, ein Optimierungs- oder Suchalgorithmus sei in der selben Situation wie ein Bergsteiger, der inmitten eines Gebirges steht, in dem er den höchsten Gipfel, also ein globales Maximum, sucht. Natürlich verdecken ihm bei dieser Suche diverse lokale Maxima die Sicht. Da er aber immer weiß, in welche Richtung er sich bewegen muss, um bergauf zu gehen, gelingt es ihm, ständig hinaufzusteigen und schließlich hoffentlich ein lokales Maximum zu erreichen.

Von `GeneticIndividual` abgeleitete Klassen stellen die Methode `climb` zur Verfügung, mit der ein einzelnes Individuum zum Hill-Climbing veranlasst werden kann. Bei Zeitplänen wird das Hill-Climbing durch zufällige Mutation realisiert; Mutationen, die zu keiner Verbesserung des Fitness-Werts führen, werden zurückgenommen.

3.6.3 Lokale Suche

Lokale exhaustive Suche ist eine weitere Möglichkeit, den Genetischen Algorithmus zu erweitern. Das Attribut „lokal“ bedeutet hier, dass die Suche innerhalb einzelner Ressourcenlisten stattfindet, also keine Querverbindungen überprüft werden, um die Komplexität des Suchvorgangs in erträglichen Grenzen zu halten.

Ist die Liste `takeFrom` einer Ressourcenliste nicht allzu groß, so findet innerhalb dieser Ressourcenliste tatsächlich eine exhaustive Suche statt. Ist sie hingegen so groß, dass eine komplette Suche zu einem nicht mehr vertretbaren zeitlichen Aufwand führt, kommt ein randomisiertes Suchverfahren zur Anwendung, bei dem der Suchbaum an bestimmten Stellen abgeschnitten wird.

3.6.4 Gerichtete Mutation

Die Idee, die hinter gerichteter Mutation (*directed mutation*) steckt, ist die, dass es sinnvoll ist, den Mutationsoperator tendenziell eher auf Gegenden eines Chromosoms anzuwenden, die für Probleme sorgen. Dies steht im Gegensatz zum klassischen Mutationsoperator, der mit gleicher Wahrscheinlichkeit auf allen Abschnitten eines Chromosoms wirkt. Gerichtete Mutation führt zu einem erhöhten Rechenaufwand, da vor ihrer Durchführung zunächst noch alle Constraint-Verletzungen neu zu berechnen sind, dieser höhere Aufwand zahlt sich aber unter Umständen durch bessere Endergebnisse aus.

3.7 Einsatz hybrider Operatoren

Um die im vorangegangenen Abschnitt besprochenen erweiterten Operatoren sinnvoll einsetzen zu können, unterstützt der Genetische Algorithmus einen Hybrid-Modus. Dieser Arbeitsmodus ist dem bereits vorgestellten Steady-State GA sehr ähnlich, es existieren jedoch drei prinzipiell verschiedene Möglichkeiten, aus der aktuellen Population ein neues Individuum zu erzeugen:

- durch Crossover und anschließende Mutation,
- durch den Einsatz des Reparatur-Operators oder
- durch den Einsatz von lokaler Suche.

Während bei einem Crossover-Prozess das gerade erzeugte Individuum das jeweils schlechteste Individuum der aktuellen Population ersetzt, wird sowohl bei der Reparatur als auch bei der lokalen Suche zufällig ein Individuum aus der aktuellen Population selektiert und dann einem der beiden Operatoren unterzogen. Falls der Vorgang erfolgreich ist, d.h. falls das so erzeugte Individuum besser ist als sein Vorgänger, wird der Vorgänger ersetzt.

Der Einsatz der drei Operatoren wird dabei aus ihrer historischen Erfolgswahrscheinlichkeit berechnet: Je erfolgreicher einer der Operatoren in der letzten Zeit war, desto häufiger wird er in der nahen Zukunft eingesetzt werden. Dabei wird jedoch durchweg sichergestellt, dass die Anwendungswahrscheinlichkeit keines Operators auf Null absinkt. Die Grundidee, den Einsatz der Operatoren von ihrer jeweiligen Erfolgswahrscheinlichkeit abhängig zu machen, ist aus [TR98] entnommen; die konkrete Implementierung ist jedoch grundverschieden von dem dort vorgestellten Verfahren.

```
sum = crossoverSuccess + repairSuccess + localSearchSuccess;
if (allSum == 0)
  crossoverProbab = repairProbab = localSearchProbab = 0.33;
else {
  double factor = (1.0 - 3 * MIN_PROBAB) / allSum;
  crossoverProbab = MIN_PROBAB + factor * crossoverSuccess;
  repairProbab = MIN_PROBAB + factor * repairSuccess;
  localSearchProbab = MIN_PROBAB + factor * localSearchSuccess;
}
selection = random.nextDouble();
if (selection < crossoverProbab)
  //perform crossover operation
else if (selection < crossoverProbab + repairProbab)
  //perform repair operation
else
  //perform local search
```

Abbildung 3.11: Anwendung hybrider Operatoren (Java-Code)

Die Veränderung der jeweiligen Erfolgsraten (`crossoverSuccess`, etc.) erfolgt dabei ähnlich einer gedämpften Schwingung mit ständig wechselnder Frequenz, wobei der Erfolg der letzten Operation zum Beispiel mit $0,01$ gewichtet wird (exponentielle Gewichtung).

Da die hybriden Operatoren im Vergleich zum einfachen Crossover einen wesentlich höheren Rechenaufwand verursachen, muss genau überlegt werden, wann sie eingesetzt werden. Durch den Parameter **ELITE_PERCENTAGE**, kann im Hybrid-Modus festgelegt werden, dass lokale Suche und Reparatur nur bei den besten $n\%$ zur Anwendung kommt.

3.8 Kapselung von Zeitplänen: `TTChromosome`

Obgleich die abstrakte Klasse `TTProblem` die Klasse `GeneticIndividual` beerbt und damit einer Optimierung durch den Genetischen Algorithmus zugänglich ist, ist es sinnvoll, den GA nicht direkt auf Instanzen von `TTProblem` arbeiten zu lassen.

Die Instantiierung eines Zeitplans findet üblicherweise durch das Einlesen einer XML-Datei statt, die sein konkrete Aussehen vorgibt. Dieses Einlesen erfordert ein Parsen der XML-codierten Informationen in der Datei und stellt einen nicht unerheblichen zeitlichen Aufwand dar. Würde der Genetische Algorithmus nun direkt auf `TTProblem`-Instanzen arbeiten, müsste bei jedem Crossover-Vorgang eine neue Instanz erzeugt und daher die XML-Datei komplett neu verarbeitet werden.

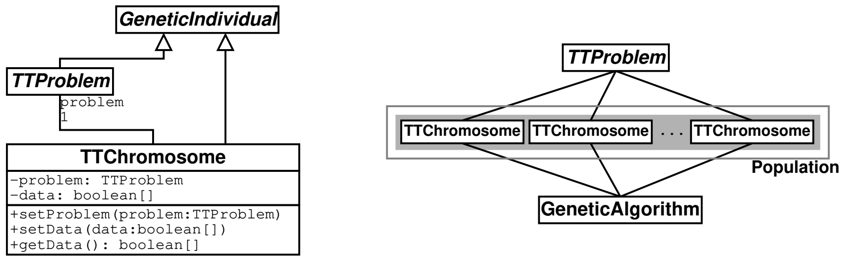


Abbildung 3.12: Verhältnis von `TTProblem` und `TTChromosome`

Diesen unerträglichen Zustand schafft die Klasse `TTChromosome` aus der Welt, indem sie lediglich eine Instanz des jeweiligen Problems zur Berechnung der Constraint-Verletzungen und des Fitness-Werts, zur Mutation usw. benutzt, die eigentlichen Ressourcenzuteilungen des Zeitplans aber in einem eigenen Bitvektor, `data`, gespeichert hält. Dieser Bitvektor ist nichts Anderes als die Konkatenation der einzelnen `assignment`-Vektoren in den Ressourcenlisten des zu Grunde liegenden Timetabling-Problems.

Findet nun ein Crossover-Vorgang oder eine Duplikation statt, so wird der daraus resultierenden neuen `TTChromosome`-Instanz einfach eine Referenz auf das verwaltete `TTProblem` übergeben, ohne dass ein neues Objekt erzeugt werden muss, was zu stark erhöhtem Aufwand führen würde.

Ein weiterer Vorteil der Kapselung von Zeitplänen durch die Klasse `TTChromosome` ist die einfache Übertragung der Pläne über ein Netzwerk, da dann nur noch der reine Bitvektor `data` verschickt werden muss, was keine besondere Herausforderung darstellt. Von dieser Funktionalität macht der verteilte Genetische Algorithmus Gebrauch, der im Anhang dieser Arbeit kurz vorgestellt wird.

3.9 Parametrisierung des Genetischen Algorithmus

Der Genetische Algorithmus, der hier zum Erstellen von Stundenplänen verwendet wird, bietet viele Möglichkeiten der Parametrisierung. Dies ist nicht bloß einfach eine Frage des Geschmacks, sondern in der Tat führen verschiedene Parameterkonstellationen in Abhängigkeit vom zu lösenden Problem zu grundverschiedenen Ergebnissen.

Die wichtigsten Parameter, die über die Methode `setParameters(Map)` der Klasse `GeneticAlgorithm` eingestellt werden können, werden im Folgenden kurz vorgestellt:

CROSSOVER_INDIVIDUALS legt die Anzahl der Individuen fest, die für einen Rekombinationsvorgang herangezogen werden. Üblicherweise sind dies zwei Individuen, nämlich Vater und Mutter. Es ist aber auch möglich, jedes neue Individuum aus einer Menge von 3 oder 4 Individuen zu konstruieren.

CROSSOVER_POINTS erlaubt die Wahl zwischen verschiedenen Arten des Crossing-over: One-Point, Multi-Point und uniformes Crossover.

DIRECTED_MUTATION_RATE bestimmt die Mutationsrate für gerichtete Mutation. Die Idee, die hinter gerichteter Mutation steckt, ist die Annahme, dass es sich vorteilhaft auf den Evolutionsprozess auswirkt, wenn tendenziell eher solche Gene mutiert werden, die zu einem schlechten Fitness-Wert führen.
Die Bestimmung der für einen schlechten Fitness-Wert verantwortlichen Gene findet innerhalb der Klasse `GeneticIndividual` statt. Bei Stundenplänen geschieht dies, indem jedem Ereignis die Anzahl von Strafpunkten zugeordnet wird, die von ihm verursacht sind.

ELITE_PERCENTAGE legt beim Generational GA den Anteil der Individuen fest, die durch den Elite-Operator direkt in die nächste Generation übernommen werden sollen; im Hybrid-Modus bestimmt der Parameter die Größe der Subpopulation, auf die hybride Operatoren angewandt werden dürfen.

KEEP_BEST veranlasst, wenn es gesetzt ist, dass das beste Individuum der aktuellen Population automatisch in die Folgegeneration übernommen wird.

MUTATION_RATE legt die Mutationsrate fest, also die Wahrscheinlichkeit, mit der ein festes Gen innerhalb eines Chromosoms (Individuums) einer Mutation unterzogen wird. Diese Wahrscheinlichkeit ist für alle Gene aller Individuen innerhalb der Population gleich. Ist MUTATION_RATE=0, so findet keine Mutation statt.

POPULATION_SIZE bestimmt die Größe der Population, die während der Evolution im Speicher gehalten wird, also die Anzahl simultan existierender verschiedener Individuen.

RUN_MODE bietet die Möglichkeit, zwischen einem *Generational GA* und einem *Steady-State GA* zu wählen. Der *Steady-State GA* kann zudem noch in einen Hybrid-Modus geschaltet werden, indem zum Beispiel auftretende Constraint-Verletzungen durch einen Reparatur-Operator behandelt werden.

SELECTION_METHOD In der Entwicklung Genetischer Algorithmen haben sich verschiedene Algorithmen durchgesetzt, die die Selektion der Eltern zur Durchführung einer Crossover-Operation regeln. Im vorliegenden GA kann zwischen fitnessproportionaler Selektion, Rang-basierter Selektion und Turnier-Selektion (*Tournament Selection*) gewählt werden.

TOURNAMENT_SIZE Falls die Turniermethode als Selektionsalgorithmus gewählt wird, kann hier die Anzahl der am Turnier beteiligten Individuen festgelegt werden.

Kapitel 4

Testprobleme

Um die Fähigkeit des vorgestellten Genetischen Algorithmus, allgemeine Timetabling-Probleme zu optimieren, empirisch zu belegen, werden in diesem Kapitel einige Beispielprobleme vorgestellt, anhand derer der GA getestet worden ist. Die Ergebnisse dieser Tests werden in Kapitel 5 präsentiert. Insbesondere wird dort untersucht werden, welchen Einfluss die hybriden Operatoren auf die Qualität der erstellten Zeitpläne haben.

4.1 Mädchen-und-Technik-Praktikum

Das erste reale Problem, mit dem der Genetische Algorithmus getestet wurde, ist das Praktikum „Mädchen und Technik“, das in jedem Jahr an der Technischen Fakultät der FAU stattfindet. Das Praktikum soll interessierten Schülerinnen die Möglichkeit geben, einen Eindruck von den Studienfächern zu bekommen, für die sie sich interessieren.

Im Rahmen des Praktikums, das eine Woche lang dauert, werden für eine bestimmte Anzahl von Teilnehmerinnen Projekte aus verschiedenen technischen Bereichen angeboten. Die Projekte bekommen je nach Art einen oder zwei Timeslots fest zugewiesen, wobei ein Timeslot entweder ein Vor- oder ein Nachmittag eines bestimmten Tages ist.

Im Vorfeld des Praktikums haben die Teilnehmerinnen die Möglichkeit, bis zu vier Projekte auszuwählen, an denen sie teilnehmen möchten, und weitere vier Alternativkurse anzugeben, an denen sie gern teilnehmen würden, falls die Kurse ihrer ersten Wahl belegt sind. Zusätzlich kann jedes Mädchen eine Menge von *Freundinnen* definieren, mit denen zusammen es an den selben Projekten teilnehmen möchte.

Die Constraints, die bei der Zuteilung der Teilnehmerinnen auf die einzelnen Projekte zu beachten sind, sieht man in Tab. 4.1. Das mit dem Mädchen-und-Technik-Praktikum verbundene Timetabling-Problem ist in seiner Größe moderat. Die Eckdaten für die Jahre 2000 bis 2002 zeigt Tab. 4.2.

Man kann erkennen, dass sowohl die Zahl der durchgeführten Projekte als auch die der am Praktikum teilnehmenden Personen in den drei untersuchten Jahren kontinuierlich gewachsen ist. Daher

Constraint	**Beschreibung**
`MinMaxConstraint`	*Hard Constraint.* Bestraft jedes Über- oder Unterschreiten des in einer Ressourcenlisten festgelegten `min`- bzw. `max`-Wertes proportional zur Größe der Über- oder Unterschreitung mit 10 Penalty Points pro fehlender oder überschüssiger Belegung.
`ClashConstraint`	*Hard Constraint.* Bestraft jede Kollision, also zwei oder mehr gleichzeitig stattfindende Veranstaltungen im Stundenplan einer Teilnehmerin mit 20 Punkten.
`DoubleSignatureConstraint`	*Hard Constraint.* Verhindert, dass die selbe Teilnehmerin für zwei Projekte der selben Art (mit der selben Signatur, z.B. „inf1") eingeplant wird. Jedes Verletzung dieses Constraints wird mit 20 Strafpunkten geahndet.
`HardProjectConstraint`	*Hard Constraint.* Stellt sicher, dass jede Teilnehmerin zu wenigstens einem Kurs eingeteilt wird, aber nicht zu mehr Kursen, als sie bei der Anmeldung angegeben hat. Pro Constraint-Verletzung fallen 10 Strafpunkte an.
`SoftProjectConstraint`	*Soft Constraint.* Bestraft Zeitpläne, in denen einem Mädchen weniger Projekte zugeteilt werden, als es sich bei der Anmeldung gewünscht hat. Für jeden Kurs, den eine Teilnehmerin weniger zugeteilt bekommt als bei der Anmeldung gewünscht, werden 2 Penalty Points verteilt.
`FriendsConstraint`	*Soft Constraint.* Soll sicherstellen, dass die in der Anmeldung angegebenen Freundschaftsbeziehungen bei der Kurseinteilung berücksichtigt werden. Belegt ein Mädchen einen Kurs, in dem die angegebenen Freundinnen nicht anwesend sind, fällt 1 Strafpunkt an.
`SoftChoiceConstraint`	*Soft Constraint.* Bewirkt eine leichte Bestrafung von Zuteilungen von Alternativkursen, so dass die Teilnehmerinnen tendenziell Kursen ihrer ersten Wahl zugeteilt werden. Bestrafung pro Kurs zweiter Wahl: 0,25 Penalty Points.

Tabelle 4.1: Rahmenbedingungen des „Mädchen und Technik"-Problems

eignet sich das Mädchen-und-Technik-Praktikum hervorragend für eine Untersuchung der Auswirkungen einer Vergrößerung der Ressourcenmenge bei gleichbleibender Problemstruktur auf das Laufzeitverhalten des Genetischen Algorithmus.

Trotz seiner im Vergleich zu anderen Problemen geringen Größe ist es offenkundig, dass ein Einteilen der Teilnehmerinnen von Hand, selbst wenn dabei nur die Hard Constraints Beachtung fänden, einen enormen Aufwand darstellen würde, der an einem einzigen Tag bestimmt nicht zu bewältigen wäre. Eine Optimierung der hinsichtlich der Soft Constraints ist ohne Zuhilfenahme eines Computers völlig undenkbar.

Jahr	Projekte	Projekttypen	Teilnehmerinnen
2000	146	37	190
2001	188	38	241
2002	199	38	303

Tabelle 4.2: Verschiedene Instanzen des „Mädchen und Technik“-Problems

4.2 Personaleinsatzplanung

Das zweite reale Problem, anhand dessen der Timetabling-GA getestet wird, ist die Personaleinsatzplanung in einem Krankenhaus, das in [GW01] vorgestellt wurde. Für jeweils eine Station eines Krankenhauses sind die Angestellten bestimmten Schichten zuzuordnen, die festlegen, wann sie arbeiten müssen.

Constraint	Beschreibung
`OutOfMinMaxConstraint`	*Hard Constraint.* Bestraft jedes Über- oder Unterschreiten der `min`-/`max`-Werte für die einer bestimmten Schicht zuzuteilenden Angestellten proportional zur Über- bzw. Unterschreitung mit 200 Punkten pro fehlendem bzw. überschüssigem Mitarbeiter.
`ClashConstraint`	*Hard Constraint.* Bestraft jede Kollision, also jede Einteilung derselben Person in zwei Schichten, die sich zeitlich überlappen mit 200 Penalty Points.
`DiffHourMonthConstraint`	*Soft Constraint.* Bestraft jedes Über- oder Unterschreiten der festgelegten durchschnittlichen Monatsarbeitszeit eines Angestellten proportional zum Betrag der Differenz (ArbeitIst - ArbeitSoll) mit 12 Punkten pro Stunde.
`OnlyOneDayFreeConstraint`	*Soft Constraint.* Soll Schichtpläne verhindern, in denen eine Person zwischen Arbeitsperioden nur einen einzigen freien Tag hat. Bestrafung: 150 Punkte.
`OnlyOneWorkingDayConstraint`	*Soft Constraint.* Gegenstück zu `OnlyOneDayFree-Constraint` für einzelne Arbeitstage zwischen dienstfreien Zeiten. Bestrafung: 150 Punkte.
`LittleFreeTimeConstraint`	*Soft Constraint.* Bestraft Zeitpläne, in denen ein Angestellter zwischen Dienstende und nächstem Diensantritt weniger als 10 Stunden frei hat pro Auftreten einer solchen Situation mit 150 Punkten.
`ContinuationBonus`	*Soft Constraint*. Ist ein Constraint, das mit Belohnung statt Bestrafung arbeitet: Für jeden Tag, den ein Mitarbeiter der selben Schicht zugeteilt ist wie am Vortag, bekommt ein Zeitplan 20 Bonuspunkte gutgeschrieben.

Tabelle 4.3: Rahmenbedingungen des Krankenhaus-Problems

Die Mitarbeiter werden nicht unmittelbar einer Schicht zugeordnet, sondern bestimmten Funktionen innerhalb dieser Schicht, die sie gemäß ihrer individuellen Qualifikationen erfüllen können, zum Beispiel Schichtleitung, Praktikant oder examiniertes Personal.

Eine Übersicht über die zu beachtenden Rahmenbedingungen gibt Tabelle 4.3.

Die Größenverhältnisse, in denen sich die Probleminstanzen der Personaleinsatzplanung bewegen, sind in Tab. 4.4 zu sehen. Das Zuteilungsproblem ist hier ein wenig anders gelagert als beim Mädchen-und-Technik-Praktikum: Zwar sind hier mit nur 21 bzw. 24 Mitarbeitern wesentlichen weniger Personen beteiligt als beim Praktikum, dafür ist aber jeder einzelne Mitarbeiter einer großen Anzahl von Schichten zuzuteilen (bei 38 Sollarbeitsstunden pro Woche sind es 15 bis 25 Schichtdienste im Monat), während im Rahmen des Mädchen-und-Technik-Praktikums jede Teilnehmerin maximal vier verschiedenen Projekten zuzuordnen ist.

Stationsname	**Mitarbeiter**	**Schichttypen**
Geriatrie	24	Frühschicht, Spätschicht
Innere	21	Früh I, Früh II, Spät, Nacht

Tabelle 4.4: Verschiedene Instanzen des Krankenhaus-Problems

Insgesamt scheinen die zwei Probleme also ungefähr von der gleichen Schwierigkeit zu sein. Diese Einschätzung wird in Kap. 5 durch die empirischen Ergebnisse bestätigt werden.

4.3 Schulstundenplanerstellung

Das dritte Testproblem schließlich, mit dem die Leistungsfähigkeit des Timetabling-GA untersucht wird, ist die Erstellung eines Stundenplans für ein mittelgroßes deutsches Gymnasium mit 113 Lehrern, 100 Räumen und 43 Schulklassen. Dieses Timetabling-Problem ist in [Ost01] eingehend untersucht worden und stellt von allen bearbeiteten Problem das mit der größten Komplexität dar. Dies liegt nicht nur an der großen Zahl der zu verplanenden Ressourcen, sondern vor allem daran, dass pro Ereignis (Schulstunde) nicht nur der Zeitraum festgelegt werden muss, während dessen es stattfindet, sondern auch der Raum, in dem man sich trifft. Lediglich die Klasse und der Lehrer sind im Voraus von der Schulleitung festgelegt und dürfen nicht verändert werden.

Constraint	**Beschreibung**
`ClashConstraint`	*Hard Constraint.* Bestraft jede Kollision im Stundeplan einer Klasse, eines Raumes oder eines Lehrers mit 200 Punkten.
`PreferenceConstraint`	*Soft Constraint.* Für jede an der Planung beteiligte Ressource kann im Voraus eine Präferenztabelle erstellt werden, die angibt, zu welchen Zeiten eine Einplanung unerwünscht ist. Wird eine Ressource einem Ereignis zugeteilt, dass während eines unerwünschten Zeitraums stattfindet, wird diese Einteilung entsprechend bestraft. Die Gewichtung liegt zwischen 0 und 9.
`RoomSizeConstraint`	*Hard Constraint.* Soll sicherstellen, dass keine Klasse in einem Raum unterrichtet wird, der zu klein für sie ist. Tritt dieser Fall ein, wird er mit 50 Strafpunkten geahndet.

Tabelle 4.5: Rahmenbedingungen des Schulstundenplan-Problems

Ein Problem dieser Art erfordert normalerweise ein hohes Maß an problemspezifischem Wissen, das beispielsweise in die `repair`-Methode der Zeitplan-Klasse integriert werden kann. Da dieses problemspezifische Wissen dem Autor nicht zur Verfügung stand, sind, mit dem Ziel, das Problem ein wenig zu vereinfachen, die üblicherweise zu beachtenden Constraints etwas eingeschränkt und nur drei Arten von Constraints beibehalten worden (siehe Tab. 4.5). Diese Einschränkung wird in den zum Teil etwas seltsamen Klassenstundenplänen deutlich werden, die beispielsweise 3 Stunden Mathematikunterricht über einen Tag verstreut vorsehen könnten.

Normalerweise wird man zusätzliche Constraints definieren, die etwa regeln, dass bestimmte Unterrichtsstunden, zum Beispiel in Hauptfächern, zu Doppelstunden gebündelt werden, und so zu schöneren Stundenplänen führen.

Kapitel 5

Ergebnisse

Es soll zunächst untersucht werden, wie weit das in den vorherigen Kapiteln vorgestellte Gerüst zur Optimierung beliebiger Zeitpläne geeignet ist. Dazu wird der Genetische Algorithmus im Folgenden in seiner einfachen Form ohne hybride Operatoren auf die in Kap. 4 präsentierten Referenzprobleme angesetzt.

Bei der Betrachtung der Verlaufskurven für die einzelnen Programmläufe des Genetischen Algorithmus, die in diesem Kapitel zu sehen sind, wird dem Leser auffallen, dass in den meisten Fällen an der waagerechten Achse nicht die aktuelle Generation, sondern die benötigte Zeit aufgetragen ist. Dies soll der besseren Vergleichbarkeit der einzelnen Verfahren, zum Beispiel von Generational GA und Steady-State GA dienen, da ein Generationen-bezogener Vergleich hier nur mit gleichzeitiger Skalierung (je nach Populationsgröße) sinnvoll wäre, beim Einsatz von hybriden Operatoren wie zum Beispiel dem Reparatur-Operator aber zu gänzlich unsinnigen Ergebnissen führen müsste.

Alle Testläufe haben unter Linux 2.4.18 auf einem Intel-Celeron mit 1200 MHz stattgefunden. Die verwendete JVM (Java 1.4.0-b92 von Sun) lief dabei im Server-Modus, was den langen Laufzeiten der einzelnen Prozesse Rechnung tragen sollte. Der Server-Modus ist ein spezieller Arbeitsmodus der Java Virtual Machine, in dem während der ersten Sekunden zusätzliche Analysen des Laufzeitverhaltens (z.B. Speichernutzung) durchgeführt werden, die zu einer höheren Geschwindigkeit führen sollen (vgl. [Mic]).

5.1 Mädchen-und-Technik-Praktikum

Zunächst wurde der Genetische Algorithmus auf Instanzen des MuT-Praktikums angewendet. Über die dabei verwendete Parametrisierung gibt die Tab. 5.1 Auskunft.

Die gewählten Terminierungskriterien entsprechen den Erfahrungen, die im Umgang mit Zeitplänen dieser Art gesammelt wurden: Eine Stunde (`TIME_LIMIT`) ist bei der gewählten Programmstruktur eine realistische Obergrenze, nach der der Genetische Algorithmus mit hoher Wahrscheinlichkeit keine besseren Zeitpläne mehr produziert. Andererseits kann man davon ausgehen,

Parameter	Wert	Parameter	Wert
POPULATION_SIZE	50	MUTATION_RATE	0,005
TIME_LIMIT	3.600	DIRECTED_MUTATION_RATE	0,000
STAGNATION_LIMIT	300	CROSSOVER_POINTS	0 (uniform)

Tabelle 5.1: GA-Parametrisierung für das MuT-Praktikum

dass ein Evolutionsprozess nach 5-minütiger Stagnation (d.h. keiner Verbesserung des besten Individuums, `STAGNATION_LIMIT`) am Ende angelangt ist bzw. nur noch marginale Verbesserungen herbeiführen wird.

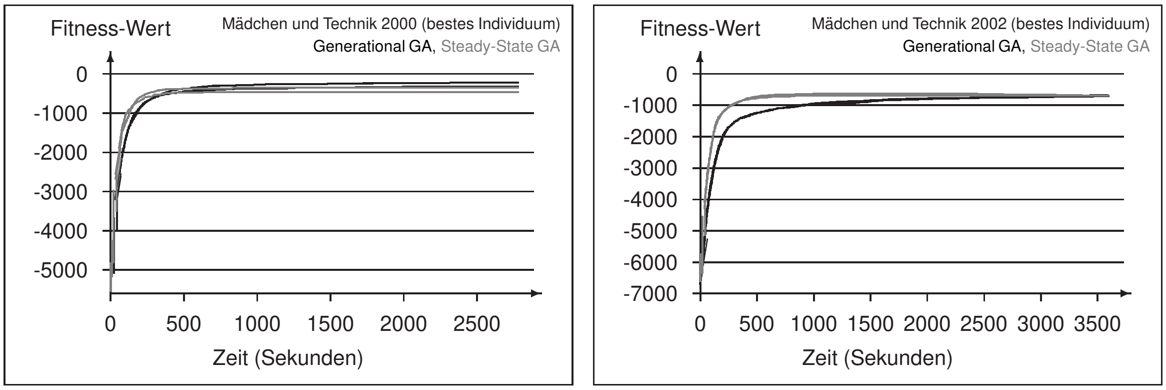


Abbildung 5.1: Evolutionsverlauf beim MuT-Problem

Zunächst wurde der Genetische Algorithmus in beiden Varianten (Generational und Steady-State) auf die 3 Probleminstanzen angesetzt. Die Graphen in Abb. 5.1 zeigen den typischen exponentiellen Verlauf des Fitness-Werts des besten Individuums für Zeitpläne der Jahre 2000 und 2002. In der benötigten Zeit zeigt sich der nicht vernachlässigbare Größenunterschied der beiden Probleminstanzen: Die 190 Teilnehmerinnen des Jahres 2000 können in ungefähr 40 Minuten verplant werden, und der Genetische Algorithmus in beiden Varianten nach spätestens 45 Minuten ab, weil keine Verbesserung mehr zu erwarten ist, wohingegen der Zeitplan für die 303 Teilnehmerinnen des Jahres 2002, wenigstens bei der Optimierung mit Hilfe des Generational GA, nach einer Stunde noch immer leichte Verbesserungen erfahren kann. Hier hat das Erreichen des eingestellten Zeit-Limits den Evolutionsprozess zum Stoppen gebracht.

Pro Jahr und GA-Variante wurde der Genetische Algorithmus fünfmal gestartet. Die Ergebnisse der einzelnen Läufe weisen dabei keine großen Unterschiede auf. In Abb. 5.2 sind die jeweils besten und schlechtesten Evolutionsabläufe für die Jahre 2000 und 2002 für den Generational (**schwarz**) und Steady-State GA (**grau**) dargestellt. Um die Unterschiede im späteren Abschnitt der Evolution deutlicher zu Tage treten zu lassen, wurden die ersten 120 Sekunden des Evolutionsprozesses entfernt.

In den Diagrammen zeigt sich deutlich die höhere Konvergenzgeschwindigkeit des Steady-State GAs im Vergleich zum Generational GA bei sonst identischer Parametrisierung. Während für die Probleminstanz aus dem Jahr 2000 der Steady-State GA erwartungsgemäß nach ungefähr 10 Minuten überholt wird, tritt dies für das Jahr 2002 nicht ein. Das liegt daran, dass der Generationen-basierte Genetische Algorithmus, will man dem Graphen Glauben schenken, im zweiten Fall noch

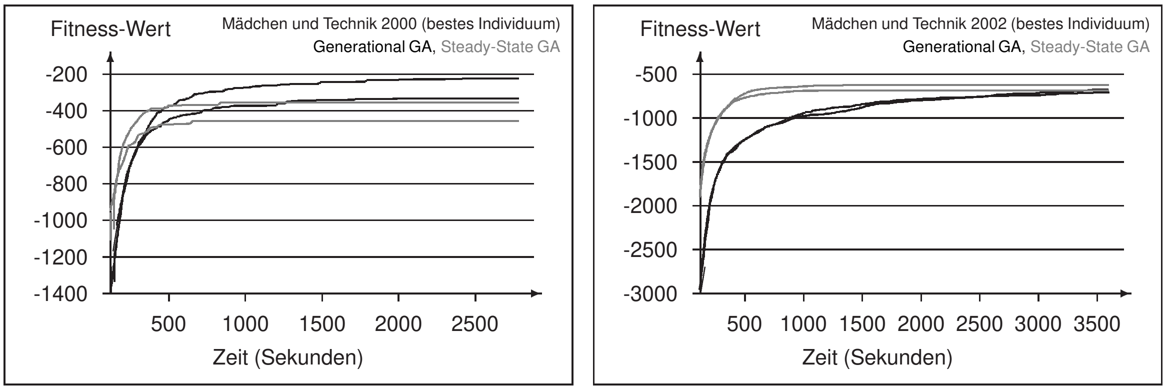


Abbildung 5.2: Evolutionsverlauf beim MuT-Problem (detailliert)

nicht an seine Grenzen gestoßen ist. Hätte der Evolutionsprozess nicht aus zeitlichen Gründen abgebrochen werden müssen, wäre das Endergebnis mit hoher Wahrscheinlichkeit wenigstens so gut gewesen wie das des Steady-State GAs.

Um die Qualität des durch den Genetischen Algorithmus konstruierten Zeitplans zu dem gegebenen Timetabling-Problem zu untersuchen, ist es notwendig, die Constraint-Verletzungen im erstellten Zeitplan genauer zu betrachten. Die konkreten Zahlen beziehen sich dabei jeweils auf den besten und den schlechtesten Zeitplan für das entsprechende Jahr. Sie können somit als repräsentativ betrachtet werden.

Constraint	**#Verletzungen/Strafpkt. (2000)**	**#Verletzungen/Strafpkt. (2002)**
MinMaxConstraint	1/10,00 (10/220,00)	1/10,00 (2/30,00)
SoftProjectConstraint	28/72,00 (39/100,00)	79/282,00 (115/376,00)
SoftChoiceConstraint	162/84,25 (160/80,75)	259/122,50 (267/116,75)
FriendsConstraint	19/58,00 (19/56,00)	78/208,00 (81/185,00)

Tabelle 5.2: Constraint-Verletzungen beim MuT-Problem

Die in Tab. 5.2 nicht aufgeführten Constraints konnten vollständig eingehalten werden. OutOfMinMaxConstraint ist somit die einzige harte Bedingung, die bei der Verteilung der Teilnehmerinnen auf die Kurse verletzt wurde.

Die einzelnen Einträge sind so zu lesen (beispielhaft und jeweils für die besten Zeitpläne):

- für 2000 werden lediglich 28 Teilnehmerinnen weniger Projekte zugewiesen als sie auf dem Anmeldeformular gewünscht hatte,
- für 2002 werden 490 ($= \frac{122,50}{0,25}$) Zuteilungen auf Kurse zweiter Wahl vorgenommen; pro Teilnehmerin sind das ungefähr $1,6$ Kurse.

Die erzeugten Zeitpläne für das Praktikum haben ziemlich genau die selbe Qualität wie die, die in den vergangenen Jahren an der Universität mit Hilfe eines Genetischen Algorithmus erzeugt wurden, der eigens für dieses Problem implementiert worden ist. Die erreichten Zahlen scheinen sehr nah am theoretisch möglichen Optimum zu liegen. Diese Einschätzung bestätigt auch die Untersuchung der erstellten Pläne, die eine vollständige Auslastung der meisten Ereignisse offenbart, d.h. Es gibt nur wenige Projekte, in denen noch Plätze frei sind.

5.2 Personaleinsatzplanung

Die zur Optimierung des Krankenhaus-Problems mit seinen beiden Instanzen verwendete Parametereinstellung des Genetischen Algorithmus ist identisch mit der, die beim Mädchen-und-Technik-Praktikum zum Einsatz gekommen ist.

Parameter	**Wert**	**Parameter**	**Wert**
POPULATION_SIZE	50	MUTATION_RATE	0,005
TIME_LIMIT	3.600	DIRECTED_MUTATION_RATE	0,000
STAGNATION_LIMIT	300	CROSSOVER_POINTS	0 (uniform)

Tabelle 5.3: GA-Parametrisierung für das Krankenhaus-Problem

Es wurden auch hier zunächst für beide Arten des Genetischen Algorithmus, Generational und Steady-State, 5 Testläufe durchgeführt, wobei beim Steady-State GA der Parameter `REPLACE_WORST` auf `true` gesetzt worden ist. Diese Parametrisierung bewirkt, dass das Produkt eines Crossover-Vorgangs immer das momentan schlechteste Individuum einer Population ersetzt, unabhängig von den Fitness-Werten der beiden. Ergebnisse der Evolutionsprozesse sind in Abb. 5.3 zu sehen.

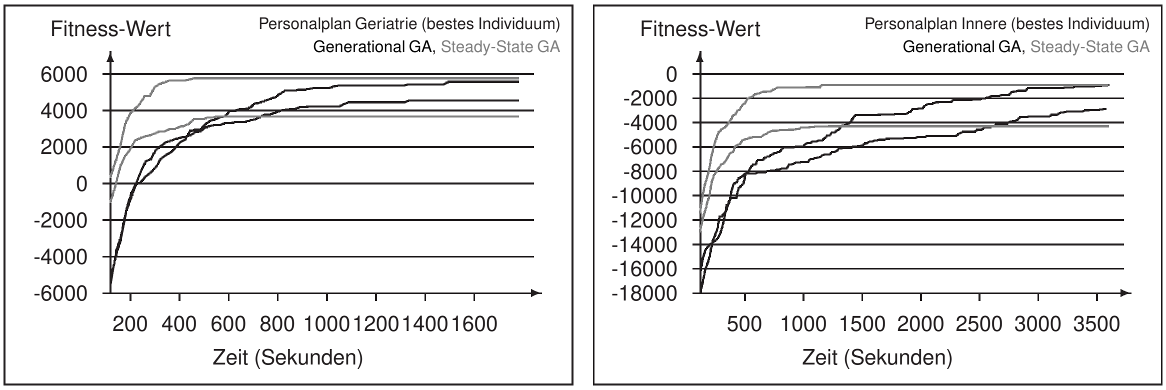


Abbildung 5.3: Evolutionsverlauf beim Krankenhaus-Problem

In den Diagrammen sind, für die beiden Probleminstanzen Geriatrie und Innere, der jeweils beste und schlechteste GA-Lauf des Generational (**schwarz**) und Steady-State GA (**grau**) eingezeichnet, wobei die ersten 5 Minuten des Evolutionsvorgangs gestrichen wurden, um den wesentlichen Teil deutlicher zu Tage treten zu lassen.

Es fällt auf, dass trotz der bereits etwas großzügiger gewählten Populationsgröße von 50 Individuen beträchtliche Schwankungen im Endergebnis auftreten: Beim Generational GA zwischen -2.892 und -922 Fitness-Punkten, beim Steady-State zwischen -4.294 und -900, jeweils für die Station „Innere“. Diese Zahlen allein sind jedoch nicht besonders aufschlussreich, interessanter ist es, sich einen Überblick über die Verletzungen der einzelnen Constraints zu verschaffen:

Die in Tab. 5.4 präsentierten Zahlen sollen anhand der besten Zeitpläne für die Station „Innere“ kurz erläutert werden:

Constraint	#Verletzg./Strafpkt. (Geriatrie)	#Verletzg./Strafpkt. (Innere)
ClashConstraint	0/0,0 (0/0,0)	0/0,0 (0/0,0)
DiffHourMonthConstraint	24/2577,0 (24/3696,0)	21/450,0 (21/594,0)
LittleFreeTimeConstraint	6/900,0 (10/1800,0)	18/4800,0 (18/5550,0)
OnlyOneDayFreeConstraint	1/150,0 (2/300,0)	0/0,0 (2/300,0)
OnlyOneWorkingDayConstraint	0/0,0 (1/150,0)	1/150,0 (3/450,0)
OutOfMinMaxConstraint	2/600,0 (1/200,0)	5/1200,0 (10/2400,0)

Tabelle 5.4: Constraint-Verletzungen beim Krankenhaus-Problem

- `DiffHourMonthConstraint`: Die mittlere Abweichung von der Sollstundenzahl beträgt pro Mitarbeiter, auf den gesamten Monat gerechnet, 1,8 Stunden ($\frac{450}{21 \times 12}$).
- `LittleFreeTimeConstraint`: Wird bei 18 Mitarbeitern verletzt, die im Schnitt ungefähr 2x weniger als 10 Stunden Freizeit zwischen zwei Schichten haben (z.B. Spätschicht mit direkt anschließender Frühschicht).
- `OnlyOneWorkingDayConstraint`: Nur ein einziger Mitarbeiter hat in seinem Schichtplan einen einzelnen Arbeitstag, dem keine anderen Schichten direkt vorausgehen oder folgen.
- `OutOfMinMaxConstraint`: 5 der insgesamt 132 Schichten werden zu viele oder zu wenige Mitarbeiter zugewiesen.

Die Auswirkung des Anti-Constraints `ContinuationBonus`, das jede Zuteilung eines Mitarbeiters zur selben Schicht über mehrere Tage hinweg belohnt, werden in 5.4 sichtbar, die beispielhaft einen Ausschnitt aus der HTML-Darstellung des erstellten Zeitplans zeigt.

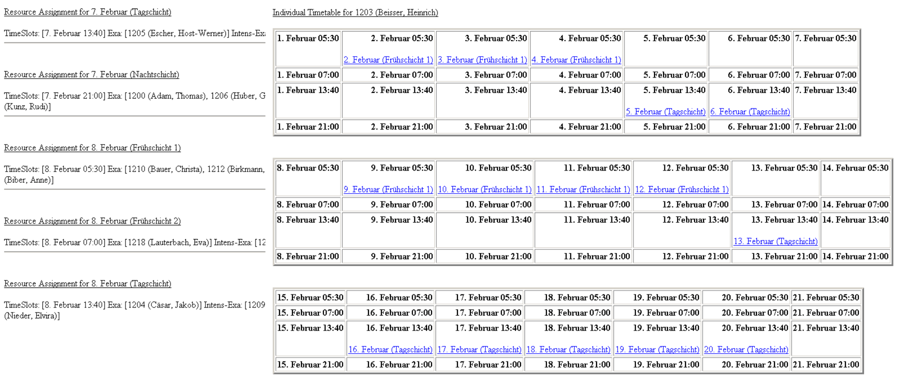


Abbildung 5.4: Mitarbeiter-Schichtpläne für das Krankenhausproblem

Der Vergleich mit den in [GW01] vorgestellten Ergebnissen für dieses Problem zeigt, dass der hier vorgestellte Genetische Algorithmus, jedenfalls ohne den Einsatz hybrider Operatoren, nicht an die Ergebnisse eines speziell für das Problem entwickelten Algorithmus herankommt, was sich vor allem in der unangenehm hohen Zahl verletzter Hard Constraints zeigt.

5.3 Schulstundenplanerstellung

Die Erwartung, dass es sich bei der Schulstundenplanerstellung um ein Problem handelt, das wesentlich komplexer ist als die beiden vorausgegangenen, schlägt sich in der Parameterwahl für dieses Problem nieder: Bereits bei der relativ einfachen Personaleinsatzplanung hat eine Populationsgröße von 50 Individuen mitunter zu frühzeitiger Konvergenz geführt. Um dieses Phänomen zu vermeiden, wurde der Genetische Algorithmus im Gegensatz zu der in [Ost01] vorgeschlagenenen Parametrisierung mit einer leicht vergößerten Population von 100 Individuen auf den Schulstundenplan angesetzt. Das Zeit-Limit wurde auf 6 Stunden festgesetzt. Die übrigen relevanten Parameter blieben gegenüber den beiden anderen Problemen unverändert.

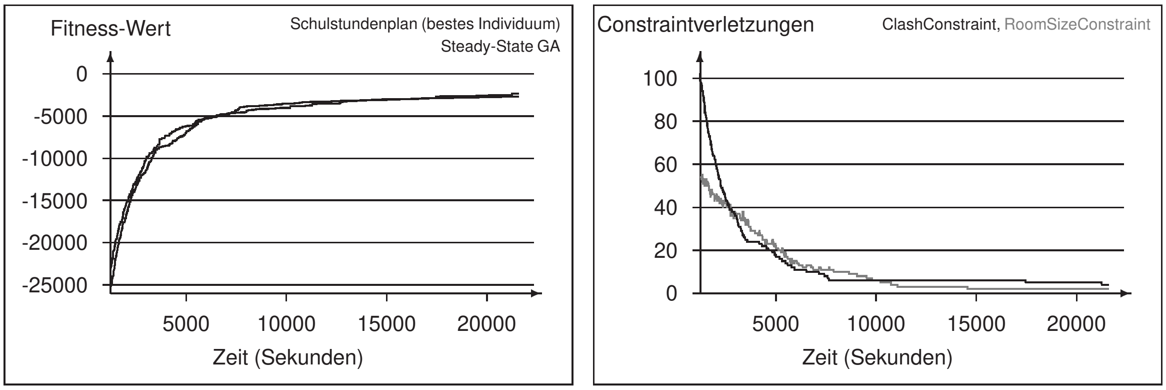


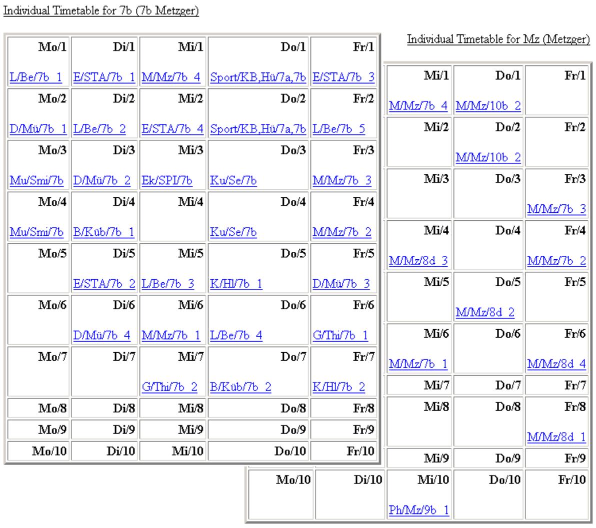
Abbildung 5.5: Evolutionsverlauf beim Schulstundenplan-Problem

Der Ablauf der Evolution ist in Abb. 5.5 dargestellt, wobei auf der linken Seite die Fitness-Werte der besten Stundenpläne im besten und im schlechtesten von drei Läufen des Steady-State GA abgebildet ist, während die rechte Seite die Entwicklung der Hard-Constraint-Verletzungen für den Fall des besten Laufes zeigt.

In den 6 Stunden, die dem Genetischen Algorithmus zur Verfügung standen, hat er immerhin eine Reduzierung der Kollisionen (`ClashConstraint`) auf 4 erreichen können. In 2 Fällen findet am Ende des Evolutionsprozesses der Unterricht in einem Raum statt, der für die jeweilige Klasse zu klein ist (Verletzung von `RoomSizeConstraint`). Insgesamt kann der GA also einen Stundenplan erstellen, in dem noch 6 Verletzungen von Hard Constraints auftreten. Das ist zwar nicht besonders viel, andererseits aber auch nicht wirklich zufriedenstellend. Im nächsten Abschnitt soll untersucht werden, wieweit der Einsatz hybrider Operatoren in dieser Hinsicht weiterhelfen kann.

Beispiele für Stundenpläne aus der Sicht einer Klasse und eines Lehrers gibt Abb. 5.6. Die Berücksichtigung der zeitlichen Präferenzen aller am Timetabling-Vorgang beteiligten Ressourcen (`PreferenceConstraint`) kommt dabei in der relativ geringen Zahl von Unterrichtsstunden am Nachmittag zum Ausdruck.

Interessant wäre auch bei der Schulstundenplanerstellung wieder der Vergleich mit den in [Ost01] präsentierten Ergebnissen. Leider finden sich in der Arbeit nur wenige Informationen über Details der erstellten Stundenpläne. Allein die Aussage, dass der finale Plan keinerlei Kollisionen mehr aufweise, kann zum Vergleich mit den hier vorgestellten Ergebnissen verwendet werden, und es zeigt sich, dass der GA mit 4 verbleibenden Kollisionen noch einen kleinen Optimierungsspielraum lässt.



Individual Timetable for 7b (7b Metzger)

Mo/1 L/Be/7b 1	Di/1 E/STA/7b 1	Mi/1 M/Mz/7b 4	Do/1 Sport/KB,Hü/7a,7b	Fr/1 E/STA/7b 3
Mo/2 D/Mü/7b 1	Di/2 L/Be/7b 2	Mi/2 E/STA/7b 4	Do/2 Sport/KB,Hü/7a,7b	Fr/2 L/Be/7b 5
Mo/3 Mu/Smi/7b	Di/3 D/Mü/7b 2	Mi/3 Ek/SPI/7b	Do/3 Ku/Se/7b	Fr/3 M/Mz/7b 3
Mo/4 Mu/Smi/7b	Di/4 B/Küb/7b 1	Mi/4	Do/4 Ku/Se/7b	Fr/4 M/Mz/7b 2
Mo/5	Di/5 E/STA/7b 2	Mi/5 L/Be/7b 3	Do/5 K/Hl/7b 1	Fr/5 D/Mü/7b 3
Mo/6	Di/6 D/Mü/7b 4	Mi/6 M/Mz/7b 1	Do/6 L/Be/7b 4	Fr/6 G/Thi/7b 1
Mo/7	Di/7	Mi/7 G/Thi/7b 2	Do/7 B/Küb/7b 2	Fr/7 K/Hl/7b 2
Mo/8	Di/8	Mi/8	Do/8	Fr/8
Mo/9	Di/9	Mi/9	Do/9	Fr/9
Mo/10	Di/10	Mi/10	Do/10	Fr/10

Individual Timetable for Mz (Metzger)

		Mi/1 M/Mz/7b 4	Do/1 M/Mz/10b 2	Fr/1
		Mi/2	Do/2 M/Mz/10b 2	Fr/2
		Mi/3	Do/3	Fr/3 M/Mz/7b 3
		Mi/4 M/Mz/8d 3	Do/4	Fr/4 M/Mz/7b 2
		Mi/5	Do/5 M/Mz/8d 2	Fr/5
		Mi/6 M/Mz/7b 1	Do/6	Fr/6 M/Mz/8d 4
		Mi/7	Do/7	Fr/7
		Mi/8	Do/8	Fr/8 M/Mz/8d 1
		Mi/9	Do/9	Fr/9
Mo/10	Di/10	Mi/10 Ph/Mz/9b 1	Do/10	Fr/10

Abbildung 5.6: Individuelle Stundenpläne für das Schul-Problem

5.4 Einsatz hybrider Operatoren

Bei der Untersuchung der Ergebnisse, die der Genetische Algorithmus, angewandt auf die verschiedenen Probleme, liefert, erkennt man leicht, dass besonders bei den etwas komplizierteren Problemen *Innere* und *Schulstundenplan* in den erstellten Zeitplänen eine unangenehm hohe Anzahl von Hard Constraints verletzt wird.

Der GA unterstützt zur Verbesserung des Evolutionsverlaufs verschiedene hybride Operatoren, die in 3.6 vorgestellt wurden und deren Auswirkungen im Folgenden beispielhaft untersucht werden sollen.

Auch wenn die Ergebnisse des Genetischen Algorithmus bei der Schulstundenplanerstellung auch ohne den Einsatz hybrider Operatoren bereits recht brauchbar sind, können sie bei der Personaleinsatzplanung und der Schulstundenplanerstellung noch nicht ganz an die problemspezifischer Algorithmen heranreichen. Daher ist es angebracht, zu untersuchen, inwiefern die Ergebnisse durch lokale Suche und Reparaturoperatoren, wie in 3.6 beschrieben, noch verbessert werden können.

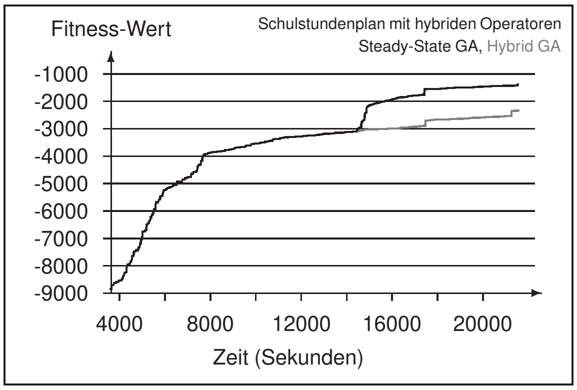


Abbildung 5.7: Evolution mit hybriden Operatoren (Schulstundenplan)

Zu diesem Zweck wurden Reparatur-Methoden für die Constraints `ClashConstraint` und `RoomSizeConstraint` implementiert, die an den Stellen, an denen Verletzungen festgestellt werden, entweder die Raumzuteilung oder die zugewiesenen Timeslots zufällig ändern. Dabei wird nicht überprüft, ob es durch diese zufällige Änderung zu anderen Constraint-Verletzungen kommt.

Der Hybrid-Modus wurde anhand des Schulstundenplan-Problems getestet, indem zunächst der Steady-State GA wie im vorhergehenden Abschnitt mit einer Population von 100 Individuen gestartet wurde. Nach 2 Stunden wurde er angehalten und danach mit der Population, die nach 2 Stunden errechnet war, zweimal neu gestartet, einmal wieder als einfacher Steady-State GA, einmal im Hybrid-Modus. Die Gesamtlaufzeit war wie im obigen Beispiel auf 6 Stunden begrenzt.

Abb. 5.7 zeigt deutlich, dass Reparatur und lokale Suche, im Hybrid-Modus des Genetischen Algorithmus eingesetzt, die Ergebnisse des Evolutionsprozesses deutlich verbessern können. Um die Unterschiede besser sichtbar werden zu lassen, wurde die erste Stunde des GA-Laufs abgeschnitten. Man kann in dem Diagramm sehr gut den Zeitpunkt erkennen, in dem der Hybrid-Modus aktiv wird und sich von den Ergebnissen, die der einfache Steady-State GA (grau) hervorbringen konnte, abhebt. Die hybriden Operatoren wurden in diesem Lauf nur auf die besten 5% der Population angewandt.

Constraint	**Strafpunkte (Steady-State)**	**Strafpunkte (Hybrid)**
ClashConstraint	800	200
PreferenceConstraint	1.431	1.089
RoomSizeConstraint	100	50

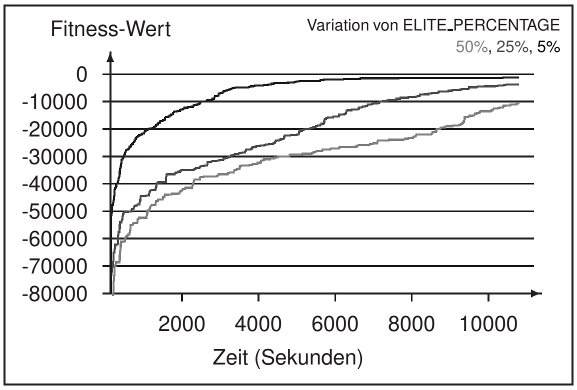
Tabelle 5.5: Constraint-Verletzungen: Hybrid vs Steady-State (Schulstundenplan)

Tabelle 5.5 gibt Aufschluss über die genauen Verbesserungen, die durch die hybriden Operatoren erzielt werden konnten. Hier zeigt sich, dass die Zahl der verletzten Hard Constraints von 6 auf 2 reduziert werden kann. Aber auch die Berücksichtigung der individuellen Präferenzen von Klassen und Lehrern können durch den Einsatz lokaler Suche wesentlich besser berücksichtigt werden.

Die Frage, auf welche Individuen die hybriden Operatoren überhaupt angewendet werden sollen, ist bereits in Kap. 3 angesprochen worden. Es ist zu befürchten, dass eine Reparatur schlechter Zeitpläne einen Aufwand verursacht, der durch die Verbesserung eines schlechten Plans zu einem mittelmäßigen Plan nicht gerechtfertigt werden kann. Um die Auswirkungen von Änderungen am Parameter **ELITE_PERCENTAGE**, der bestimmt, welcher Anteil der aktuellen Population den hybriden Operatoren zugänglich ist zu untersuchen, wurde der Genetische Algorithmus mit drei verschiedenen Werten (5%, 25% und 50%) auf das Schulproblem angesetzt. Eine Elite-Größe von 5% bedeutet dabei, dass nur die besten 5% der Bevölkerung repariert werden können.

Es zeigt sich, dass der GA, wenn er nur die besten 5% der Population der Anwendung hybrider Operatoren unterzieht, wesentlich schneller zu guten Ergebnissen kommt als bei 25% oder gar 50%. Kleinere Werte als 5% haben hingegen in den Experimenten keine Verbesserung mehr bewirken können.

Festzuhalten ist also, dass es gar nicht so einfach ist, die geeignete Parametrisierung zu finden. Es steht außer Frage, dass der Genetische Algorithmus auch mit einem Wert von 100% für ELITE_PERCENTAGE in der Lage gewesen wäre, Ergebnisse ähnlicher Qualität zu erzeugen wie bei



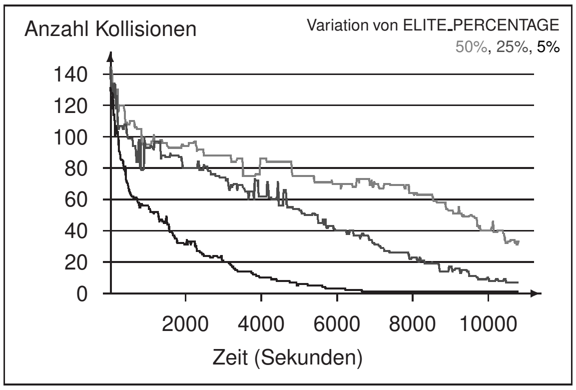


Abbildung 5.8: Auswirkungen von ELITE_PERCENTAGE im Hybrid-Modus

5%. Entscheidend ist aber nicht nur die Qualität der erstellten Zeitpläne, sondern auch die zur Optimierung benötigte Zeit, und hier zeigen sich rasch sehr große Unterschiede.

Wenn man nun den erneuten Vergleich mit [Ost01] wagt, so sieht man, dass keine schwerwiegenden Unterschiede mehr festzustellen sind: Der hybride GA ist in der Lage, in akzeptabler Zeit Pläne zu generieren, in denen lediglich ein einziges Hard Constraint verletzt wird; Kollisionen finden keine mehr statt. Hätte der Genetische Algorithmus, ebenso wie der spezielle Algorithmus in [Ost01], die vollen 10 Stunden Rechenzeit bekommen, wäre er diesem hinsichtlich seiner Ergebnisse ganz sicher nicht unterlegen gewesen.

Kapitel 6

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde ein Gerüst vorgestellt, mit Hilfe dessen sich Instanzen einer sehr großen Klasse von Timetabling-Problemen beschreiben und lösen lassen. Alle in der Timetabling-Beschreibungssprache GTL formalisierten Probleme lassen sich auf einfache Weise in Java-Klassen übersetzen, die – in dieses Gerüst eingebettet – dafür sorgen, dass jedes solche Problem durch den implementierten Genetischen Algorithmus optimiert werden kann.

Anhand von realen Beispielproblemen (Mädchen-und-Technik-Praktikum, Personaleinsatzplanung, Schulstundenplanerstellung) konnte gezeigt werden, dass die zur internen Repräsentation der Zeitpläne gewählten Strukturen angemessen sind und der zur Lösung der Timetabling-Probleme gewählte Genetische Algorithmus in der Lage ist, den gewünschten Optimierungsprozess durchzuführen.

Die Ergebnisse, zu denen der Genetische Algorithmus bei den verschiedenen Testproblemen gekommen ist, wurden mit denen speziell für das jeweilige Problem erstellter Optimierungsalgorithmen verglichen, und es konnten hinsichtlich der Qualität der Zeitpläne keine gravierenden Unterschiede festgestellt werden. Der zeitliche Aufwand für die Berechnung der Zeitpläne liegt ebenfalls auf einem Niveau mit dem anderer Lösungsalgorithmen, wenn man unvermeidbaren Geschwindigkeitsnachteil, den die Verwendung von Java als verwendete Plattform mit sich bringt, herausrechnet.

Weiterhin ist es gelungen, der Notwendigkeit problemspezifischer Operatoren durch den Einsatz hybrider Operatoren, zum Beispiel zur Reparatur festgestellter Constraint-Verletzungen, in sinnvoller Weise Rechnung zu tragen. Es hat sich gezeigt, dass Reparatur und lokale Suche, wenn ihr Einsatz nicht blind erfolgt, sondern einigen einfachen Regeln gehorcht, die Ergebnisse des reinen Genetischen Algorithmus noch einmal erheblich verbessern können.

Die Notwendigkeit, solche Operatoren einzuführen, ist besonders am Beispiel des Schulstundenplans deutlich geworden, bei dem es dem einfachen Genetische Algorithmus nicht gelungen ist, einen kollisionsfreien Zeitplan zu erstellen. Mit Hilfe von Reparatur-Methoden und dem Einsatz von lokaler Suche im Verlauf des Evolutionsprozesses konnte diese Unzulänglichkeit behoben werden.

Problematisch ist im Zusammenhang mit dem Genetischen Algorithmus vor allem seine Benutzung: Weder die einem Problem zu Grunde liegende GTL-Definition noch die Instanzbeschreibung in XML können von einem Laien ohne Weiteres erstellt werden. Noch schwieriger sind die Definition von Constraints und eventuellen Reparaturmethoden, die beide eine detaillierte Kenntnis der Klassenstruktur erfordern.

Wünschenswert ist also eine einfach zu bedienende Oberfläche, mit der sich Probleme durch wenige Klicks komponieren lassen und die eine Menge von Basis-Constraints zu Verfügung stellt, die dann nicht mehr neu programmiert werden müssten. Durch einen einzigen Knopfdruck sollte es dann möglich sein, den Genetischen Algorithmus zu starten und wenige Stunden später einen fertigen Zeitplan in den Händen zu halten.

Weiter ist es sinnvoll, die an einer Schule oder Universität in großem Umfang zur Verfügung stehenden Rechnerkapazitäten zur Verbesserung eines Zeitplans bzw. zur Beschleunigung des Evolutionsprozesses zu nutzen. Einen Vorschlag für eine solche Nutzung findet man im Anhang dieser Arbeit; bessere Techniken wären sehr wünschenswert.

Teil I

ANHANG

Kapitel 7

Verzeichnisübersicht

Zu dieser Arbeit gehört eine Menge von Dateien, die auf dem beigefügten Datenträger zu finden sind. Die Dateien sind in verschiedenen Verzeichnissen abgelegt, über die an dieser Stelle ein kurzer Überblick gegeben wird.

Verzeichnisname	**Inhalt**
`/databases`	Beispielprobleme
`/databases/hospital`	Problemdefinitionen der verschiedenen Krankenhaus-Instanzen (proprietäres Format)
`/databases/maedchen`	Problemdefinitionen der verschiedenen Mädchen-und-Technik-Instanzen (proprietäres Format)
`/databases/school`	Problemdefinition des Schulstundenplanproblems (proprietäres Format)
`/databases/xml`	Probleminstantiierungsdaten im XML-Format
`/latex`	LaTeX-Quelltext dieser Arbeit (inkl. Grafiken und Diagramme)
`/src`	Java-Quellen des Timetabling-GA
`/src/doc`	JavaDoc-Dokumentation zu den Java-Quellen
`/src/logs`	Protokollinformationen ausgewählter GA-Läufe
`/src/protocol`	Protokollinformationen vieler verschiedener GA-Läufe

Kapitel 8

Programmquellen

8.1 Paketübersicht

Der vorgestellte Genetische Algorithmus zur Erstellung von Zeitplänen besteht aus einer Menge von Java-Klassen, die - je nach Funktionalität - in verschiedenen Paketen zu finden sind:

Tabelle 8.1: Paketübersicht

Paketname	**Beschreibung**
`examples`	Beispielprobleme, die mit dem GA gelöst werden können. Dies sind zum einen Standard-Suchprobleme, wie etwa die Aufgabe, das Minimum einer einfachen Funktion zu finden, zum anderen realistische Timetabling-Probleme.
`ga`	Ein voll funktionsfähiger GA, der für die Lösung beliebiger Such- und Optimierungsprobleme geeignet ist.
`helperclasses`	Nützliche Klassen, die keinem anderen Bereich zugeordnet werden konnten, z.B. eine sortierende Variante der `ArrayList` mit logarithmischem Suchaufwand oder eine Klasse zum Austausch globaler Informationen.
`tools`	Nützliches rund um den Genetischen Algorithmus und Timetabling, zum Beispiel eine grafische Oberfläche für den GA oder ein Adapter, der aus der XML-Beschreibung eines Stundenplans ein hübsches HTML-Dokument erstellt.
`ttclasses`	Alle Klassen, die zur Beschreibung eines Timetabling-Problems benötigt werden, also zum Beispiel Klassen für Ereignisse, Ressourcen und Constraints.

Die einzelnen Klassen, geordnet nach den Paketen, zu denen sie gehören, werden im Folgenden vorgestellt, wobei auf unwesentliche und private Klassen verzichtet wird. Eine genauere Beschreibung kann in der Java-Dokumentation im Verzeichnis src/doc auf dem zur Arbeit gehörenden Datenträger gefunden werden.

8.2 Das Paket `ga`

Tabelle 8.2: Klassenübersicht für `ga`

Klassenname	**Aufgabe**
Klassenname	**Aufgabe**
`GeneticAlgorithm`	Der Kern des Genetischen Algorithmus, der den Ablauf des Evolutionsprozesses steuert.
`GeneticIndividual`	Eine abstrakte Klasse, die die Funktionalität jedes durch den GA optimierbaren Problems festlegt, indem sie zum Beispiel Methoden für Mutation und Crossover definiert.
`IslandFerry`	Die Fähre ist in der Lage, über eine Netzwerkverbindung Individuen der aktuellen Population an andere Rechner (Inseln) zu verschicken oder Individuen, die an einem anderen Ort abgesandt worden sind, in Empfang zu nehmen.

8.3 Das Paket `ttclasses`

Tabelle 8.3: Klassenübersicht für `ttclasses`

Klassenname	**Aufgabe**
`Constraint`	Die abstrakte Klasse `Constraint` dient als Superklasse für alle innerhalb eines Timetabling-Problems zum Einsatz kommenden Rahmenbedingungen.
`HardConstraint`	Superklasse für alle Hard Constraints.
`SoftConstraint`	Superklasse für alle Soft Constraints.
`ConstraintViolation`	Alle Constraint-Verletzungen, die von der `compute`-Methode eines Constraint-Objekts festgestellt werden, werden in einer neu erzeugten Instanz von `ConstraintViolation` festgehalten.
`Event`	Dies ist die abstrakte Superklasse für alle im Rahmen eines Timetabling-Problems vorkommenden Ereignisse.
`PlanningClass`	Superklasse von `Event` und `Resource`.
`Resource`	Alle Ressourcen, z.B. Zeitintervalle, Lehrer, Räume, sind von dieser Klasse abgeleitet.
`ResourceList`	Kapselt alle Ressourcen eines bestimmten Typs innerhalb eines Ereignisses. Ein Ereignis besteht beispielsweise aus einer Liste der Timeslots, die ihm zugeordnet sind, und einer Liste von Agenten, die an dem Ereignis teilnehmen.
`ResourceType`	Mit Hilfe dieser Klasse werden alle Ressourcen, die an einem Timetabling-Vorgang teilnehmen, nach ihrem Typ voneinander gestrennt.
`TTChromosome`	Diese Klasse dient der Kapselung von Instanzen der Klasse `TTProblem`. Sie ist von `ga.GeneticIndividual` abgeleitet und dient in erster Linie der Beschleunigung des Evolutionsvorgangs: Würde der Genetische Algorithmus direkt auf `TTProblem` arbeiten, müsste z.B. bei jedem Crossover-Vorgang eine neue Instanz des TT-Problems erzeugt und initialisiert werden, was sehr zeitaufwändig ist.

Tabelle 8.3: - *Fortsetzung-* Klassenübersicht für `ttclasses`

Klassenname	Aufgabe
`TTProblem`	Basis-Klasse für die Beschreibung von Timetabling-Problemen. `TTProblem` ist von `ga.GeneticIndividual` abgeleitet, beim Einsatz des Genetischen Algorithmus sollte jedoch nicht direkt auf `TTProblem`, sondern auf `TTChromosome` gearbeitet werden.
`TTSolver`	Das Interface `TTSolver` definiert die Funktionalität aller Algorithmen, die in der Lage sind, ein gegebenes Timetabling-Problem zu lösen (oder optimieren).
`Time`	Die Klasse `Time` die der Modellierung von Zeit im Rahmen der Zeitplanerstellung.
`TimeInterval`	Ein `TimeInterval` ist das geschlossene Intervall zwischen zwei Zeitpunkten, also Objekten vom Typ `Time`.

8.4 Das Paket `examples`

Das Paket `examples` setzt sich aus mehreren Teilpaketen zusammen.

8.4.1 Das Paket `examples.maedchen`

Tabelle 8.4: Klassenübersicht für `examples.maedchen`

Klassenname	Aufgabe
`ClashConstraint`	Überprüft einen gegebenen Zeitplan auf das Auftreten von Kollisionen (*Clashes*).
`DoubleSignatureConstraint`	Überprüft, ob einer Teilnehmerin mehrere Projekte des selben Typs zugeordnet worden sind, sie also de facto zweimal am selben Projekt teilnimmt.
`FriendsConstraint`	Versucht, die Einhaltung der angegebenen Freundschaften in die Kurseinteilung einfließen zu lassen.
`Girl`	Klasse für die Teilnehmerinnen des Praktikums.
`HardProjectConstraint`	Bestraft Konstellationen, in denen eine Teilnehmerin keinem einzigen Projekt zugeteilt ist oder aber mehr Projekten als die angegeben hat.
`MaedchenUndTechnik`	Die Kernklasse des „Mädchen und Technik“-Problems ist von der Klasse `TTProblem` abgeleitet.
`Groebner`	Eine Variante des Mädchen-und-Technik-Problems, die die von Matthias Gröbner in seinem ursprünglichen GA verwendeten Constraints benutzt.
`MinMaxConstraint`	Überprüft einen Zeitplan auf Verletzung der festgelegten Ober- und Untergrenzen für die Anzahl der einem Kurs zugeteilten Teilnehmerinnen.
`Project`	Ein Projekt (Ereignis) innerhalb des Praktikums.

Tabelle 8.4: - *Fortsetzung-* Klassenübersicht für `examples.maedchen`

Klassenname	**Aufgabe**
`SoftChoiceConstraint`	Bestraft jede Zuteilung einer Teilnehmerin zu einem Projekt zweiter Wahl.
`SoftProjectConstraint`	Verteilt Penalty Points, falls eine Teilnehmerin weniger Projekte zugeteilt bekommen hat als sie sich gewünscht hatte.

8.4.2 Das Paket `examples.hospital`

Tabelle 8.5: Klassenübersicht für `examples.hospital`

Klassenname	**Aufgabe**
`ClashConstraint`	Verhindert Kollisionen in den Zeitplänen der Mitarbeiter.
`DiffHourMonthConstraint`	Überprüft die Abweichung zwischen monatlicher Soll- und Ist-Arbeitszeit in einem Schichtplan. Die Bestrafung erfolgt proportional zum Betrag der Differenz.
`Hospital`	Die Hauptklasse des Krankenhaus-Problems ist von der Klasse `TTProblem` abgeleitet.
`LittleFreeTimeConstraint`	Bestraft Zeitpläne, in denen ein Mitarbeiter zwischen zwei Schichten weniger als 10 Stunden freier Zeit hat.
`OnlyOneDayFreeConstraint`	Hat ein Angestellter zwischen zwei Arbeitsperioden (bestehend aus einem oder mehreren aufeinander folgenden Diensttagen) nur einen Tag frei, wird das bestraft.
`OnlyOneWorkingDayConstraint`	Gegenstück zu `OnlyOneDayFreeConstraint` für einzelne Arbeitstage.
`OutOfMinMaxConstraint`	Für jede Position der einzelnen Schichten (z.B. „Praktikant“) sind eine obere und eine untere Grenze festgelegt, deren Einhaltung hier überprüft wird.
`Shift`	Eine Schicht (Ereignis) innerhalb der Personalplanung.
`Staff`	Ein Mitarbeiter oder eine Mitarbeiterin.

8.5 Das Paket `helperclasses`

Tabelle 8.6: Klassenübersicht für `helperclasses`

Klassenname	**Aufgabe**
`GlobalMethods`	Enthält Methoden, die häufig benutzt werden und daher sinnvollerweise an einer für alle Klassen zugänglichen Stelle abgelegt werden, z.B. zur Umwandlung von Strings in Listen.
`GlobalObjects`	Enthält nur ein einziges globales Objekt, nämlich eine statische Instanz von `java.util.Random`.

Tabelle 8.6: - *Fortsetzung-* Klassenübersicht für `helperclasses`

Klassenname	**Aufgabe**
`God`	Gott ist in der Lage, im Auftrag des Genetischen Algorithmus oder anderer Klassen mittels des Java-Reflection-Mechanismus neue Instanzen von Individuen zu erzeugen.
`MultiList`	Eine mehrdimensionale, erweiterbare Liste von Objekten, ähnlich einer `ArrayList`.
`SortedList`	Eine Implementierung von `java.util.List`, mit logarithmischem Suchaufwand bei jedem Aufruf der Methode `contains`.

8.6 Das Paket `tools`

Tabelle 8.7: Klassenübersicht für `tools`

Klassenname	**Aufgabe**
`ExtractColumns`	Diese Klasse ist in der Lage, einzelne Spalten aus einer vom GA erzeugten Protokolldatei zu extrahieren und so für eine weitere Verarbeitung zur Verfügung zu stellen.
`frontend.FrontEnd`	Ein grafisches Frontend für den Genetischen Algorithmus.
`ttexplorer.TTExplorer`	Ein grafisches Hilfsmittel zum Inspizieren der erstellten Zeitpläne. Der TTExplorer verarbeitet die XML-Beschreibung eines Zeitplans.

Kapitel 9

Arbeiten mit dem Timetabling-GA

In diesem Kapitel sollen die Grundlagen im Umgang mit dem in der Arbeit vorgestellten Genetischen Algorithmus dargelegt werden. Dies betrifft vor allem die Ein- und Ausgabe-Schnittstelle, aber auch die Initialisierung des GAs.

9.1 Konstruieren eines eigenen Timetabling-Problems

Wie eingangs erläutert, besteht ein Timetabling-Problem im Zusammenhang mit diesem Programmpaket aus einer GTL-Definition, die die Struktur des Problems festlegt und einer XML-Datei, der die konkreten Werte für ein Problem entnommen werden können. Ein Beispiel für die Beschreibungssprache GTL findet sich bereits in Kap. 1.

Eine als GTL-Datei vorliegende Problemdefinition kann mittels des Hilfsprogramms `gtl2java`, das nicht Bestandteil dieser Arbeit ist, nach Java übersetzt werden. Dabei werden die verschiedenen Klassen für Ereignisse, Ressourcen und Constraints automatisch erzeugt und in dem gewünschten Verzeichnis abgelegt. Nachdem dieser Schritt vollzogen ist, muss nur noch die konkrete Instantiierung des Timetabling-Problems, die bearbeitet werden soll, als XML-Datei bereitgestellt werden, um das Problem durch den GA optmieren zu lassen.

9.2 Definieren eigener Constraints und Reparaturmechanismen

Will man mittels der Beschreibungssprache GTL ein eigenes Timetabling-Problem definieren, liegt das Hauptproblem auf der präzisen Formulierung der einzelnen Constraints und, falls hybride Operatoren eingesetzt werden sollen, der zu den Constraints gehörenden Reparaturmethoden.

Zunächst empfiehlt es sich, einen Blick in die Java-Quellen der Beispielprobleme und der ihnen zugeordneten Constraintklassen zu werfen. Sind dann noch Fragen offen, kann das Studium der zum Paket `ttclasses` gehörenden Dokumentationsdateien diese hoffentlich beantworten. Dennoch bleiben einige Hinweise offen, die unbedingt beachtet werden sollten, wenn man zu akzeptablen Ergebnissen bei der Lösung des neu geschriebenen Problems kommen will:

- Grundsätzlich sollten nur die Verletzungen von Hard Constraints repariert werden. Dies ist zum einen deshalb sinnvoll, weil die Reparatur einer jeden Verletzung einen gewissen zeitlichen Aufwand darstellt, zum anderen, weil durch jede Repair-Operation möglicherweise neue Verletzungen auftreten.
- Wenn man gerichtete Mutation (Parameter `DIRECTED_MUTATION_RATE`) als hybriden Operator einsetzen möchte, muss irgendwo ermittelt werden, wie viele Penalty Points durch ein bestimmtes Ereignis hervorgerufen werden. Der einzige Punkt im gesamten Ablauf des Evolutionsprozesses, an dem dies möglich ist, ist die Berechnung der Constraint-Verletzungen, die in den verschiedenen Constraint-Klassen stattfindet.
  Gänzlich analog zu dem in den Beispielen zu findenden Code muss also in der `compute`-Methode jedes Constraints jedem Ereignis, das eine Verletzung verursacht, ein Kontingent an Strafpunkten zugewiesen werden, damit die gerichtete Mutation eine Wirkung haben kann.

9.3 Einlesen einer Problem-Instanz

In Abb. 9.1 sieht man eine zu dem in Kap. 1 vorgestellten GTL-Code für die Personaleinsatzplanung in einem Krankenhaus passende XML-Beschreibung einer konkreten Probleminstanz.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<TTProblem layout="7,4,4" description="Schichtplanung Innere">
  <Resource type="TimeSlot" name="1. Feb. 05:30" from="1998/02/01/05:30"
    until="1998/02/01/13:10" layout="1,1,1"/>
  <Resource type="TimeSlot" name="1. Feb. 07:00" from="1998/02/01/07:00"
    until="1998/02/01/15:10" layout="1,1,1"/>
  <Resource type="Staff" name="1200" description="Adam, Eva" workLoad="38.5"
    features="EXA,W" constraints="Clash,DiffHourMonth,OnlyOneDayFree,...">
  <Resource type="Staff" name="1205" description="Eva, Adam" workLoad="38.5"
    features="EXA,INTENS,M" constraints="Clash,DiffHourMonth,OnlyOneDayFree,...">
  <Event type="Shift" name="1. Feb. (Früh 1)" constraints="MinMaxConstraint" shiftType="0">
   <ResourceList name="TimeSlots" min="1" max="1" takeFrom="1. Feb. 05:30" fixed="true">
   <ResourceList name="Exa" min="1" max="2" extra="1" takeFrom="1200,1205">
   <ResourceList name="Intens-Exa" min="1" max="2" extra="1" takeFrom="1205">
  </Event>
</TTProblem>
```

Abbildung 9.1: XML-Code für die Instantiierung eines Timetabling-Problems

Der abgedruckte Text zeigt einen leicht abgewandelten Ausschnitt aus der XML-Datei, die für die Instantiierung der Personaleinsatzplanung für die Station „Innere“ eines Krankenhauses verantwortlich ist (Pfad: *databases/xml/innere_februar.xml*).

Dabei verdient lediglich das Attribut *layout* in der Definition von *TTProblem* und *TimeSlot* eine nähere Betrachtung: Die Layout-Angabe wird lediglich für die HTML-Ausgabe eines fertigen Stundenplans verwandt. Dabei kann die Visualisierung eines Stundenplans als beschriftete Punktmenge im Vektorraum $\mathbb{N}^3$ betrachtet werden.

Die Visualisierung der in Abb. 9.1 vorgestellten Probleminstanz lässt sich am besten grafisch beschreiben, was in Abb. 9.2 geschieht. Da die zweidimensionale Projektion der dreidimensionalen Ausgabe eines Zeitplans nicht besonders übersichtlich ist, werden verschiedene Z-Komponenten in der HTML-Ausgabe als untereinander liegende Tabellen dargestellt.

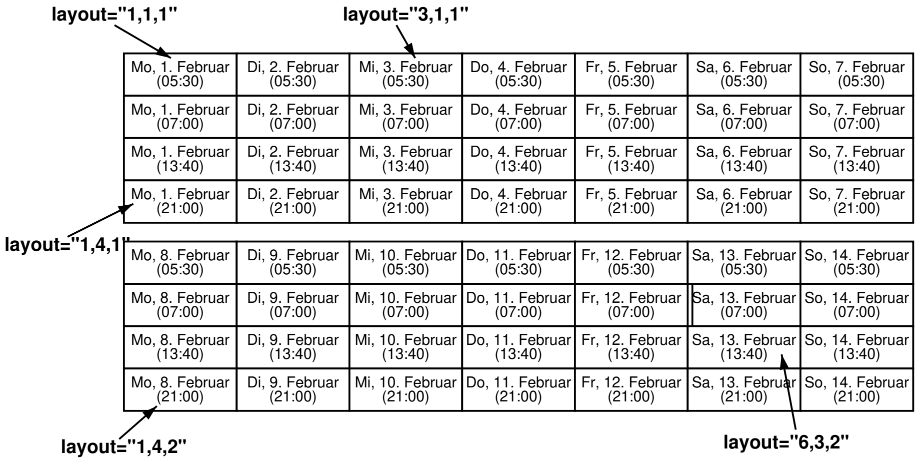


Abbildung 9.2: Grafische Darstellung der Timeslots eines Zeitplans

Nachdem die GTL-Beschreibung eines Timetabling-Problems nach Java konvertiert worden ist, kann ein konkretes, als XML-Datei gegebenes, Problem bearbeitet werden, indem eine Instanz der Basis-Klasse, in den vorgestellten Beispielen also eine der Klassen `MaedchenUndTechnik`, `Hospital` oder `School`, mit dem Namen der XML-Datei als Konstruktor-Parameter erzeugt wird. Eine so erzeugte Probleminstanz kann dann direkt oder über den Umweg der Klasse `TTChromosome` an den Genetischen Algorithmus übergeben und gelöst werden.

Zur eingehenderen Beschäftigung mit dem XML-Eingabeformat empfiehlt es sich, die Beispieldateien im Verzeichnis */databases* des zur Arbeit gehörenden Datenträgers anzusehen.

9.4 Ausgabe eines fertigen Zeitplans

Innerhalb des Genetischen Algorithmus kann die Ausgabe des Endprodukts eines Evolutionsprozesses durch den Parameter OUTPUT gesteuert werden, der das Zielverzeichnis angibt, in dem verschiedene Beschreibungen des am Ende der Evolution besten Individuums abgelegt werden sollen.

Es handelt sich um die Dateien OUTPUT/ga.txt, OUTPUT/ga.html und OUTPUT/ga.xml, die eine textuelle Beschreibung, eine HTML-Ansicht und die konkrete Struktur als XML-Code des so gespeicherten Individuums beinhalten.

Die Text-Ausgabe, zu sehen in Abb. 9.3, verzichtet auf die Struktur des Zeitplans und bietet so einen schnellen Überblick über die Einteilungen und die aufgetretenen Constraint-Verletzungen. Für die Ressourcenlisten aller beteiligten Ereignisse sind die zugeteilten Ressourcen aufgeführt, und am Ende der Datei findet man eine Übersicht aller aufgetretenen Constraint-Verletzungen.

Die XML-Beschreibung eines fertigen Stundenplans unterscheidet sich in ihrer Struktur kaum von der XML-Beschreibung eines konkreten Probleminstanz und bedarf daher keiner weiteren Erklärung.

Die Ausgabe eines Zeitplans als HTML-Dokument ist die übersichtlichste Art, den Zeitplan darzustellen, da hier u.a. Rücksicht auf die zeitliche Anordnung der zu belegendes Timeslots genommen wird. In der HTML-Ausgabe finden sich die Zuteilungen für alle Ereignisse sowie die einzelnen Sichten der am Timetabling-Vorgang beteiligten Ressourcen, also zum Beispiel die individuellen Zeitplane aller Mitarbeiter der Krankenhaus-Station.

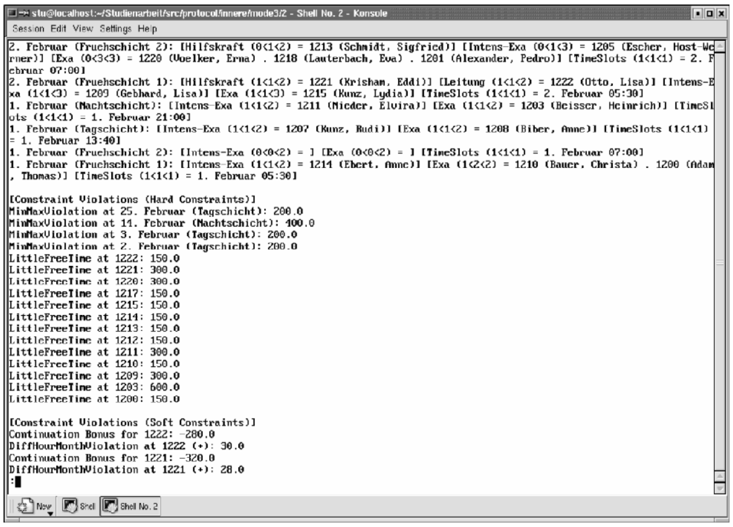


Abbildung 9.3: Text-Ausgabe eines Zeitplans

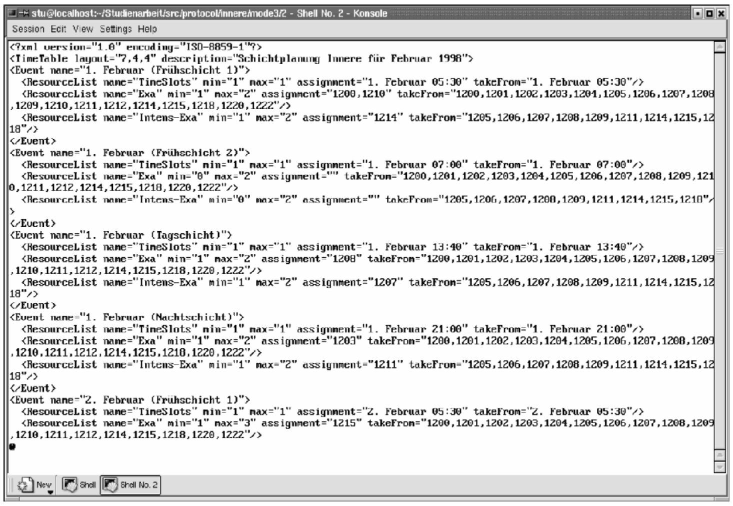


Abbildung 9.4: XML-Ausgabe eines Zeitplans

9.5 Starten des Genetischen Algorithmus

Sind aus der GTL-Beschreibung eines Timetabling-Problems die notwendigen Java-Klassen erzeugt worden und steht eine Probleminstanz als XML-Datei zur Verfügung, dann kann der Timetabling-Prozess durch Starten des Genetischen Algorithmus einfach eingeleitet werden.

Der Startvorgang ist in Abb. 9.6 abgebildet. Dabei sind alle Eingaben des Benutzers in Grau gehalten, während die Ausgaben des Systems schwarz dargestellt sind.

Wie man sieht, kann die Problemklasse, die bearbeitet werden soll, mittels des Java-Reflection-Mechanismus leicht durch Eingabe des Klassennamens (in der Abbildung „examples.hospital.Hospital“) ausgewählt werden. Die XML-Datei, die zur Instantiierung des Problems herangezogen werden soll, wird ebenfalls durch Eingabe des Pfadnamens festgelegt. Die dritte Eingabe, die beim Starten des Genetischen

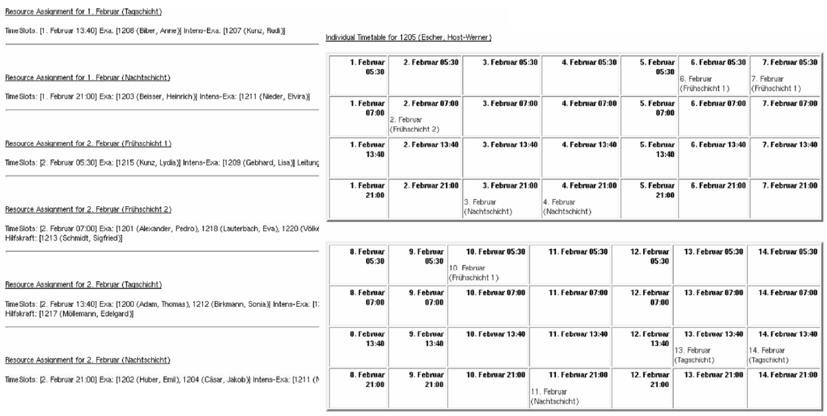


Abbildung 9.5: HTML-Ausgabe eines Zeitplans

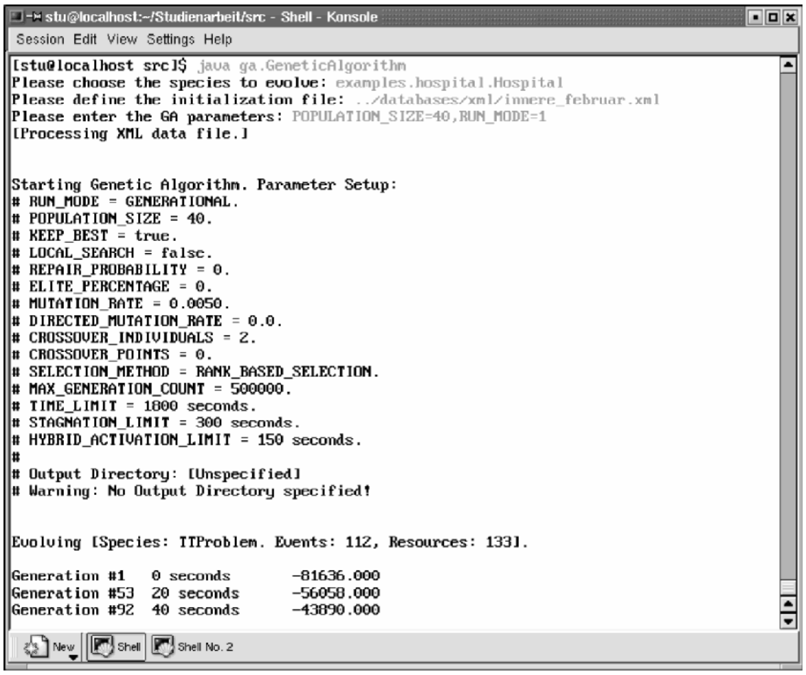


Abbildung 9.6: Starten des Genetischen Algorithmus

Algorithmus zu tätigen ist, legt schließlich die Parameter fest. Eine Beschreibung der wichtigsten Parameter des GA kann man in Abschnitt 3.9 finden.

9.6 Verwenden der Protokoll-Informationen

Um verschiedene Parameterkonfigurationen und den Einsatz von Reparaturalgorithmen vergleichend beurteilen zu können, ist es sinnvoll, die Protokollinformationen, die während eines Evolutionsvorgangs in einer Datei gespeichert werden können, auszuwerten.

Beim Starten des Genetischen Algorithmus kann man dem Parameter **OUTPUT** als Wert einen Verzeichnisnamen zuweisen. In dieses Verzeichnis werden dann während des Evolutionsvorgang in bestimmten

Abständen Protokollinformationen und Sicherungskopien der momentan besten Lösung des aktuellen Problems geschrieben. Am Ende der Evolution werden Text-, XML- und HTML-Ausgabe des besten Individuums ebenfalls in das durch **OUTPUT** festgelegte Verzeichnis geschrieben.

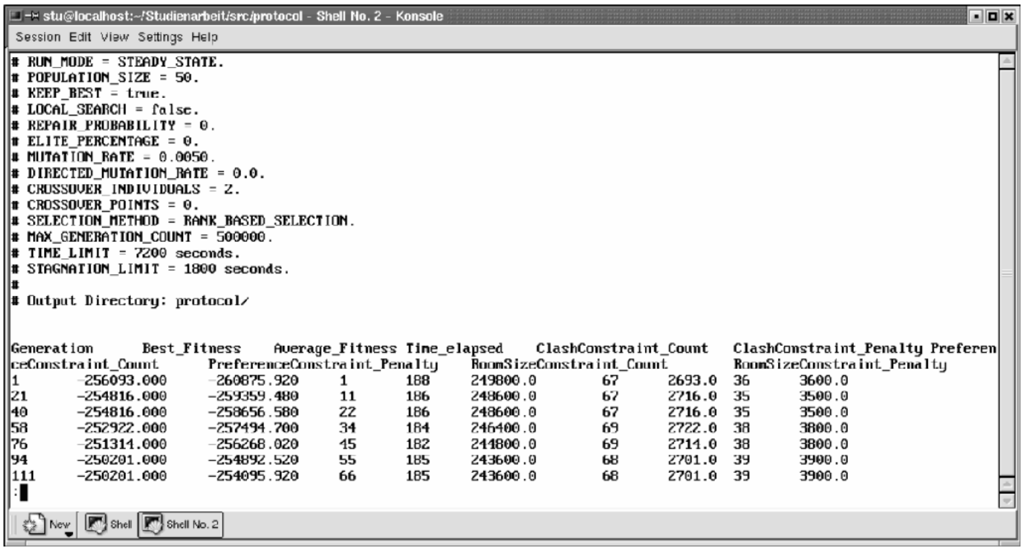


Abbildung 9.7: Die Protokolldatei des Genetischen Algorithmus

Die Protokollinformationen finden sich in der Datei „ga.log“. Ein Beispiel für eine solche Logdatei sieht man in Abb. 9.7. Alle Spalten der aufgezeichneten Protokollinformationen sind durch Tabulatoren (`'\t'`) getrennt, was eine einfache Trennung der Daten, zum Beispiel mit Hilfe der Klasse `ExtractColumns` im Paket `tools`, ermöglicht.

Kapitel 10

Der parallele Genetische Algorithmus

10.1 Arten paralleler Genetischer Algorithmen

Genetische Algorithmen simulieren den Ablauf natürlicher Evolution. Natürliche Evolution, die im Wesentlichen auf Fortpflanzung, also der Rekombination des Genmaterials von üblicherweise zwei verschiedenen Individuen basiert, verläuft offenkundig in hohem Maße parallel.

Es existieren verschiedene Techniken, diese Parallelität der Evolution auch im Rahmen Genetischer Algorithmen zu realisieren und auf diese Weise die Effizienz eines GA beziehungsweise die Güte der von ihm produzierten Ergebnisse zu verbessern:

- Eine Möglichkeit ist es, jedes Individuum auf einem eigenständigen Kleinstrechner unterzubringen und dann diese Rechner, zum Beispiel zum Crossover, miteinander kommunizieren zu lassen.
- Eine weiter verbreitete Variante ist das sog. Insel-Modell. Bei dieser Art eines parallelen Genetischen Algorithmus werden auf mehreren eigenständigen Rechnern abgeschlossene Populationen einem Evolutionsvorgang unterzogen, es läuft also auf jedem dieser Rechner ein eigenständiger GA. Der Informationsaustausch zwischen den verschiedenen Populationen (Inseln) geschieht durch Migration: Mit einer bestimmten Frequenz werden von jeder Insel Gruppen von Individuen an andere Inseln verschickt.

Eine Übersicht über die verschiedenen Arten paralleler GAs liefert [CP95].

10.2 Struktur der vorliegenden Implementierung

Der im Rahmen dieser Arbeit implementierte parallele Genetische Algorithmus basiert auf dem Insel-Modell. Damit die einzelnen Rechner, auf denen jeweils ein autonomer Genetischer Algorithmus läuft, miteinander kommunizieren können, also die Inseln, um bei dem gewählten Bild zu bleiben, Individuen auf andere Inseln schicken können, müssen die einzelnen Instanzen des GA auf besondere Weise initialisiert werden, indem bestimmten Parametern die gewünschten Werte zugewiesen werden:

ISLAND_MODE muss auf `true` gesetzt werden, um den Insel-Modus zu aktivieren,

PORT_NUMBER erhält den Wert des Ports, auf dem Neuankömmlinge von anderen Inseln empfangen werden sollen, z.B. `PORT_NUMBER=1234`,

HARBOR_FILE gibt den Namen der Datei an, in der Informationen über die anderen Inseln, denen Individuen geschickt werden können gefunden werden, und

MIGRATION_INTERVAL legt fest, in welchem Abstand (gemessen in Generationen) jeweils ein Individuum versandt werden soll.

Das Format der Datei, die die Insel-Informationen enthält, ist dabei denkbar einfach. Sie enthält Zeilen der Form: `NAME, ADRESSE, PORTNUMMER` .

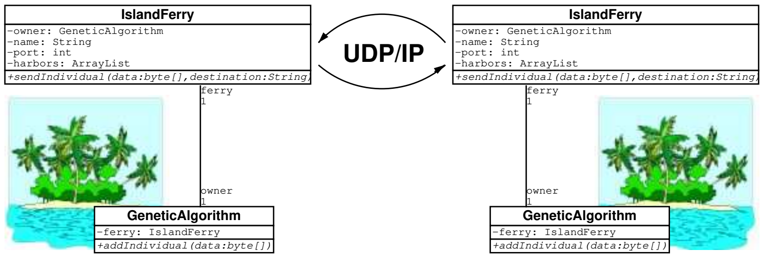


Abbildung 10.1: Kommunikation beim verteilten GA

Das Versenden und Empfangen der Individuen übernimmt dabei die Klasse `IslandFerry` im Paket `ga`. Die Fähre versendet einfach die Bitvektor-Darstellungen der Klasse `TTChromosome` in komprimierter Form als UDP-Paket an andere Rechner bzw. lauscht auf dem angegebenen Port auf neue UDP-Pakete, die Individuen von anderen Inseln enthalten und leitet sie an den zuständigen GA weiter, damit der sie in die aktuelle Population aufnimmt.

10.3 Ergebnisse

Es ist interessant, zu untersuchen, wie weit der im vorhergehenden Abschnitt vorgestellte Insel-GA in der Lage ist, die Ergebnisse bei den verschiedenen Testproblemen zu verbessern. Von besonderem Interesse ist hier das Schulstundenplan-Problem, das sich durch seine enorme Komplexität von den beiden anderen Problemen klar unterscheidet.

Zu diesem Zweck wurde auf 4 verschiedenen, durch ein Netzwerk verbundenen Rechnern mit je 1200 MHz je ein GA-Prozess im Steady-State-Modus mit einer Populationsgröße von 100 Individuen gestartet. Die Migrationsgeschwindigkeit wurde auf 1 Individuum pro Generation, d.h. pro 100 untersuchten Zeitplänen, eingestellt. Die Migration verläuft dabei so, dass ein nach bestimmten Kriterien selektiertes Individuum an eine zufällig ausgewählte Insel verschifft wird und dort das momentan schwächste Individuum ersetzt.

Den Verlauf der Evolution sieht man in Abb. 10.2. Zum Vergleich wurde neben dem Evolutionsvorgang des parallelen GA (schwarz) auch noch der beste auf einem Einzelrechner erreichte Prozess (grau) mit in das Diagramm aufgenommen.

Es zeigt sich, dass der einfache parallele GA ohne Reparatur-Operatoren bereits in dieser Konfiguration mit lediglich 4 verschiedenen Rechnern nah an die Ergebnisse des Hybrid-GA mit Reparatur und lokaler Suche herankommt: Am Ende des Evolutionsvorgangs wird beinhaltet der Stundenplan nur noch eine einzige Kollision.

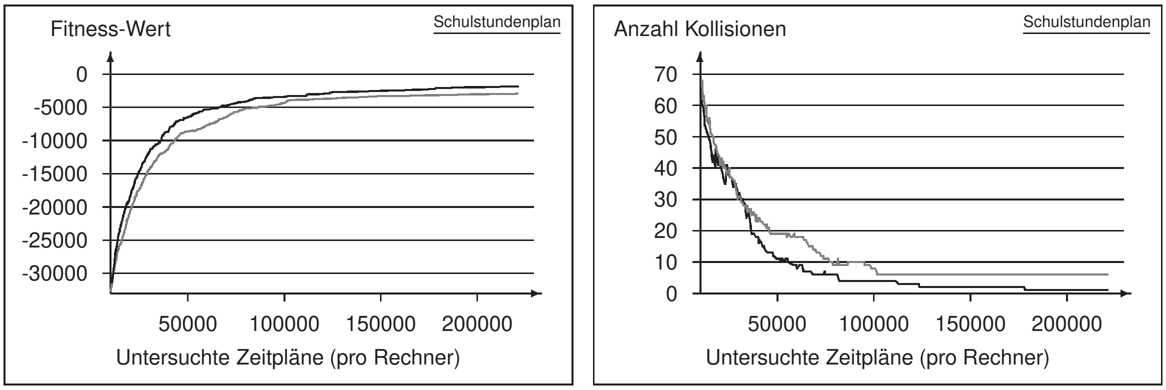


Abbildung 10.2: Vergleich von normalem und parallelem GA

Die folgende Tabelle gibt genaueren Aufschluss über die Qualität der einzelnen Pläne nach fünf Stunden Laufzeit und verdeutlicht noch einmal die Verhältnisse zwischen den verschiedenen Varianten des Genetischen Algorithmus:

Größe	**Steady-State**	**Steady-State (parallel)**	**Hybrid Steady-State**
Fitness-Wert	-2.670	-1.857	-1.360
Anzahl Kollisionen	5	1	1

Ein weiterer Testlauf des parallelen GA im Hybrid-Modus konnte den Fitness-Wert des besten Individuums nach 5 Stunden noch einmal auf -1.282 anheben. Dies bedeutet jedoch gegenüber dem Ergebnis des einfachen hybriden GA auf einem Rechner nur eine marginale Verbesserung, was die Vermutung nahelegt, dass man sich hier bereits, was die Anzahl der Constraint-Verletzungen betrifft, in der Nähe des globalen Optimums befindet.

Kapitel 11

Zusätzliche Hilfsprogramme

In diesem Kapitel finden sich zwei kleine Hilfsprogramme für den Genetischen Algorithmus, die nicht Bestandteil der Arbeit sind, die aber an dieser Stelle eine kurze Erwähnung finden sollen, um zu zeigen, in welche Richtung eine weitere Entwicklung des Programmpakets gehen könnte.

11.1 Ein grafisches Frontend für den GA

Im Paket `tools.frontend` findet man ein grafisches Frontend für den Genetischen Algorithmus.

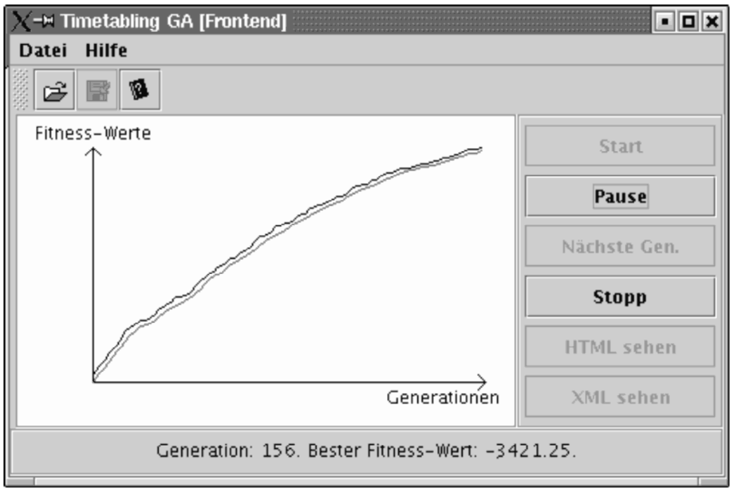


Abbildung 11.1: Evolutionsvorgang im GA-Frontend

Mit ihm kann man einen Evolutionsvorgang starten, anhalten, die Evolution Schritt für Schritt (generationsweise) ablaufen lassen und sich die Zwischenergebnisse anzeigen lassen.

11.2 Ein Timetable-Explorer

Der Timetable-Explorer (im Paket `tools.ttexplorer`) ist ein praktisches Hilfsmittel zum Untersuchen der vom Genetischen Algorithmus oder einer anderen `TTSolver`-Implementierung erstellten Lösungen eines bzw. Zeitplänen zu einem Timetabling-Problem(s).

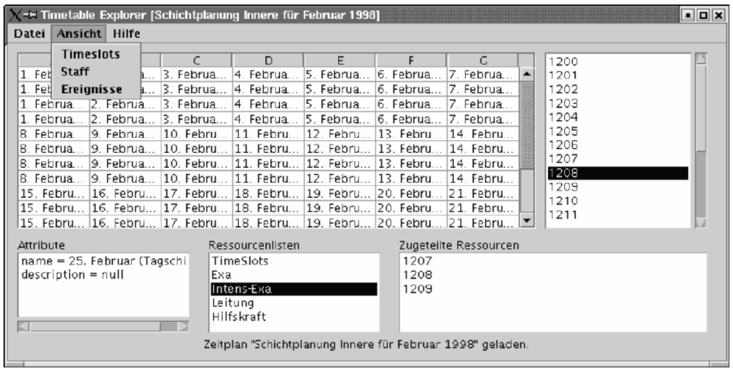


Abbildung 11.2: Der Timetable-Explorer im Einsatz

Er liest die XML-Beschreibung eines Zeitplans ein und erlaubt es dem Benutzer, zwischen den Sichten der einzelnen Ereignisse und Ressourcen zu wechseln.

[]

Literaturverzeichnis

[Bak85] J. Baker. Adaptive selection methods for genetic algorithms. *Proceedings of the First International Conference on Genetic Algorithms and their Application*, pages 101–111, 1985.

[BC97] E. Burke and M. Carter, editors. *Springer Lecture Notes in Computer Science*, volume 1408. Springer-Verlag, 1997.

[BC02] E. Burke and P. De Causmaecker, editors. *Proceedings of the 4th International Conference on the Practice and Theory of Automated Timetabling*. KaHo St.-Lieven, Department of Industrial Engineering, Gent, Belgium, August 2002.

[BE00] E. Burke and W. Erben, editors. *Springer Lecture Notes in Computer Science*, volume 2079. Springer-Verlag, 2000.

[BR95] E. Burke and P. Ross, editors. *Springer Lecture Notes in Computer Science*, volume 1153. Springer-Verlag, 1995.

[Bri81] A. Brindle. *Genetic Algorithms for Function Optimization*. University of Alberta, 1981.

[BT95a] T. Blickle and L. Thiele. A comparison of selection schemes used in genetic algorithms. *TIK-Report*, 11, 1995.

[BT95b] T. Blickle and L. Thiele. A mathematical analysis of tournament selection. *Genetic Algorithms: Proceedings of the 6th International Conference*, 1995.

[CGH$^+$02] St. Cagnoni, J. Gottlieb, E. Hart, M. Middendorf, and G. R. Raidl, editors. *Springer Lecture Notes in Computer Science*, volume 2279. Springer-Verlag, April 2002.

[Cos94] D. Costa. A tabu search algorithm for computing an operational timetable. *European Journal of Operational Research*, 76:98–100, 1994.

[CP95] E. Cantu-Paz. A summary of research on parallel genetic algorithms. *IlliGAL Report No. 95007*, 1995.

[CRF94] D. Corne, P. Ross, and H.-L. Fang. Evolutionary timetabling: Practice, prospects and work in progress. In Prosser [Pro94].

[Dar59] C. Darwin. *The Origin of Species*. John Murray, 1859.

[ea99] W. Banzhaf et. al., editor. *Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann, 1999.

[ea01a] E. J. W. Boers et. al., editor. *Springer Lecture Notes in Computer Science*, volume 2037. Springer-Verlag, April 2001.

[ea01b] L. Spector et. al., editor. *Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann, July 2001.

[EMG01] F. Espinoza, B. Misker, and D. Goldberg. A self adaptive genetic algorithm. In et. al. [ea01b], pages 75–80.

[Fan94] H.-L. Fang. *Genetic Algorithms in Timetabling and Scheduling*. PhD thesis, Department of Artificial Intelligence, University of Edinburgh, September 1994.

[FCMR99] C. Fernandes, J. Caldeira, F. Melicio, and A. Rosa. Evolutionary algorithm for school timetabling. In et. al. [ea99], page 1777.

[GD91] D. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. *Foundations of Genetic Algorithms*, pages 69–93, 1991.

[GJ79] M. Garey and D. Johnson. *Computers and Intractibility: a Guide to the Theory of NP-Completeness*. Freeman, 1979.

[Gol89] D. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison Wesley, 1989.

[GW01] M. Gröbner and P. Wilke. Optimizing employee schedules by a hybrid genetic algorithm. In et. al. [ea01a], pages 463–472.

[GW02] M. Gröbner and P. Wilke. A general view on timetabling problems. In Burke and Causmaecker [BC02], pages 221–227.

[Hol75] J. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975.

[Hor95] J. Horn. *The Nature of Niching: Genetic Algorithms and the Evolution of optimal, cooperative Populations*. University of Illinois Press, 1995.

[Kin00] J. Kingston. Modelling timetabling problems with sttl. In Burke and Erben [BE00], pages 433–445.

[KW92] L. Kang and G. White. A logic approach to the resolution of constraints in timetabling. *European Journal of Operational Research*, 61:306–317, 1992.

[Mic] Sun Microsystems. Java technology homepage, http://java.sun.com/.

[Ost01] N. Oster. Stundenplanerstellung für schulen mit evolutionären verfahren. Master's thesis, Universität Erlangen-Nürnberg, Lehrstuhl für Programmiersprachen und Programmiermethodik, Martensstraße 3, 91058 Erlangen, July 2001.

[Pro94] P. Prosser, editor. *Proceedings of the UK Planning and Scheduling SIG Workshop*, September 1994.

[RCF94] P. Ross, D. Corne, and H.-L. Fang. Improving evolutionary timetabling with delta evaluation and directed mutation. In Y. Davidor, H.-P. Schwefel, and R. Männer, editors, *Parallel Problem Solving from Nature – PPSN III*, pages 556–565, Berlin, 1994. Springer.

[Sch95] A. Schaerf. A survey of automated timetabling. Technical report, CWI National Research Institute for Mathematics, Computer Science / Department of Software Technology, 1995.

[Sch97] U. Schöning. *Theoretische Informatik - kurzgefaßt*. Spektrum Akademischer Verlag, 3 edition, 1997.

[Sys89] G. Syswerda. Uniform crossover in genetic algorithms. *Proceedings of the Third International Conference on Genetic Algorithms and their Application*, pages 2–9, 1989.

[Sys91] G. Syswerda. A study of reproduction in generational and steady-state genetic algorithms. *Foundations of Genetic Algorithms*, pages 94–101, 1991.

[TR98] A. Tuson and P. Ross. Adapting operator settings in genetic algorithms. *Evolutionary Computation*, 6(2):161–184, 1998.

[Whi89] D. Whitley. The genitor algorithm and selection pressure. *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121, 1989.

Index

Allele, 19
Arbeitsmodus, 24

Bitvektor, 18

Chromosomen, 19
Clash Constraint, 5
Constraint, 14
Constraints, 2, 14
ConstraintViolation, 14
Crossover, 19, 21, 28

Definitionen, 3

Ereignis, 11
Event, 9, 11

Färbbarkeit, 5
Fitness-Funktion, 18

GA
- Generational, 24, 34
- Starten des, 61
- Steady-State, 24, 34

Gene, 19
GeneticAlgorithm, 23
GeneticIndividual, 23
Genetische Algorithmen
- Vorteile von, 18

Goldberg, D., 17
Graph Colouring, 5
GTL, 6

Hard Constraints, 3
Hill-Climbing, 24, 31
Holland, J., 18
HybridMode, 31

Individuum, 19
Insel-Modell, 64
IslandFerry, 64

Kollisionen, 9
Komplexitat, 4

Lösung, 4
- optimale, 4

Logische Programmierung, 6
lokale Suche, 24, 31

Mutation, 19, 23, 26
- gerichtete, 31

Mutationsoperator, 23
Mutationsrate, 23

NP-vollstandig, 5

Operatoren
- hybride, 24, 30
- klassische, 24

Parameter, 33
Parametrisierung, 33
Penalty Points, 3
PlanningClass, 9

Rahmenbedingungen, 14
Rekombination, 19
repair-Methode, 30
Reparatur, 32
Reparatur-Operator, 24, 30
Resource, 9
ResourceType, 9
Ressourcenlisten, 12

Scheduling-Problem, 3
Selektion, 20
- fitness-proportionale, 21
- Rang-basierte, 21
- Turnier-, 21

Selektionsverfahren, 21
Soft Constraints, 3
Stundenplan, 4
Suche
- lokale, 32

Tabu-Suche, 6
Timetabling-Problem, 3
Timetabling-Vorgang, 4
Travelling Salesperson, 18
TTChromosome, 33
TTProblem, 9

Wissen
- problemspezifisches, 15