DIPLOMARBEIT

Ein Genetischer Algorithmus für das Generalized Assignment Problem

ausgeführt am Institut für Computergraphik und Algorithmen
der Technischen Universität Wien

unter Anleitung von
Ass.Prof. Univ.Doz. Dipl.-Ing. Dr.techn. Günther Raidl

durch

Harald Feltl
Krottenbachstrasse 1A/23,
A-1190 Wien

Wien, im April 2003

,,There is a theory which states that if ever anyone discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable."

,,There is another theory which states that this has already happened."

– Douglas Adams, ,,The Hitchhiker's Guide to the Galaxy", The Restaurant at the End of the Universe (1980)

Kurzfassung

Im Rahmen dieser Diplomarbeit wurde ein genetischer Algorithmus (GA) zum näherungsweisen Lösen einer NP-schwierigen kombinatorischen Optimierungsaufgabe, bei der es auf die Zuordnung von Elementen ankommt, entwickelt und implementiert. Konkret geht es um das *„Generalized Assignment Problem"*, das in der Maschinenbelegungs- und Ressourcenplanung auftritt: Fertigungsaufträge sollen so auf Ressourcen verteilt werden, daß Resourcen-Beschränkungen nicht überschritten werden und die entstehenden Kosten minimal sind. Das implementierte Programm ermöglicht es die Auswirkungen verschiedener genetischer Operatoren sowie Heuristiken anhand unterschiedlicher Probleminstanzen zu untersuchen. Implementiert wurden: Rekombination: OnePoint-, TwoPoint- und Uniform-Crossover; Mutation: Random- und Swap-Mutation sowie Mutation mit Heuristik nach Martello und Toth; Initialisierung: Random-, Constraint-Ratio Initialisierung sowie Initialisierung basierend auf der LP-Lösung (CPLEX) und mittels Heuristik von Martello und Toth; Verbesserung/Reparatur: nach einer Idee von Chu und Beasley sowie mit Heuristik von Martello und Toth. Ein hybrider GA von Chu und Beasley, bestehend aus dem GA zur Lösungssuche kombiniert mit einer problemspezifischen Heuristik, wurde um zwei verschiedene, heuristische Initialisierungen erweitert, die vorwiegend nur gültige Kandidatenlösungen liefern. Weiters wurde ein Variablen-Reduktions-Schema als Vorverarbeitungsschritt eingebracht, um die Problemgröße zu verringern. Der neue genetische Algorithmus wird mit dem Ansatz von Chu und Beasley sowie exakten Verfahren verglichen. Aufgrund der gezielten Anwendung problemspezifischer Heuristiken konnten wir eine Verbesserung der bisherigen, besten Ergebnisse erzielen.

Abstract

In this diploma thesis we consider a genetic algorithm (GA) which uses heuristic knowledge to solve an NP-complete combinatorial optimization problem in which items need to be assigned. This thesis deals particularly with the *Generalized Assignment Problem* which is the problem of finding an optimal (minimum or maximum) cost assignment of a set of jobs to a set of agents subject to resource constraints. The implemented program allows to examine the effects of different genetic operators as well as heuristic strategies on the basis of different problem instances. These are the crossover operator (one-point, two-point and uniform crossover), the mutation operator (random and swap mutation as well as mutation based on a heuristic of Martello and Toth), the initialization operator (random, constraint-ratio initialization as well as initialization based on the lp solution obtained by CPLEX and based on a heuristic of Martello and Toth) and the repair/improvement operator (based on a heuristic of Chu and Beasley). We also tried to reduce problem size by introducing a variable reduction scheme. The hybrid GA from Chu und Beasley was extended by heuristic initialization procedures which follow a best-try strategy to generate only feasible candidate solutions. The new genetic algorithm is compared to the approach of Chu and Beasley as well as the Branch-and-Bound approach used by CPLEX. Comparing our GA with other existing algorithms we achieved superior results due to the problem-specific heuristics used.

Inhaltsverzeichnis

Algorithmenverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

In der Produktions- bzw. Prozeßplanung trifft man häufig auf das Problem Elemente (Aufträge, ...) auf gegebene Resourcen mit limitierten Kapazitäten möglichst optimal zu verteilen. Dieses sogenannte *Generalized Assignment Problem* (GAP) kann als kombinatorisches Optimierungsproblem interpretiert werden. Aufgrund der NP-Vollständigkeit [12] des Problems wird die Suche nach guten heuristischen Methoden, die nicht unbedingt das tatsächliche Optimum, sondern in kurzer Zeit eine möglichst gute Näherungslösung finden, interessant. Aus den bisherigen Arbeiten geht hervor, daß Ansätze mit genetischen Algorithmen gute Lösungen für das GAP liefern.

Genetische Algorithmen (GAs) sind ein spezieller Typ von evolutionären Algorithmen. *Evolutionäre Algorithmen* (EAs) orientieren sich in stark vereinfachter Weise am Vorbild des natürlichen Evolutionsprozesses. Ein wichtiger Mechanismus ist die Fortplanzung von Individuen und, damit verbunden, die Weitergabe von Erbinformationen. Im Zuge der Fortpflanzung kann es durch Faktoren wie Mutation und Rekombination zur Veränderung oder Vermischung der Erbinformation von Individuen kommen. Auf diese Weise entstehen unterschiedlich konkurrenzfähige Nachkommen. Sie stehen im Wettbewerb um Überleben und Fortpflanzung. Im Zuge natürlicher Auslese setzen sich tendenziell die unter den gegebenen Umweltbedingungen besser angepaßten Individuen gegenüber ihren Konkurrenten durch und geben wiederum ihre Erbinformationen weiter. Aus dem Wechselspiel von Variation und Selektion läßt sich dann die schrittweise Entstehung der heutigen Arten aus früheren Urformen erklären. EAs versuchen nach stark vereinfachten Prinzipien der natürlichen Evolution Optimierungsaufgaben zu lösen. Besonders sind sie für Probleme mit sehr großen, komplexen Suchräumen geeignet, wo eine Optimumsuche durch exakte Verfahren nicht mehr möglich ist. An die Stelle der Individuen treten hier Lösungskandidaten für das gegebene Problem.

Im Rahmen dieser Diplomarbeit wurde ein Programm zum Lösen des GAP, basierend auf einem GA, implementiert. In diesem werden unterschiedliche Operatoren getestet und verglichen. Über verschiedene Parameter kann die Funktionsweise des GA variiert werden.

Die vorliegende Arbeit ist wie folgt aufgebaut. Eine detailierte Beschreibung des Generalized Assignment Problem inklusive Anwendungsfälle wird im Kapitel 2 gegeben. Das Kapitel 3 beinhaltet eine Zusammenfassung der bisherigen Ansätze zum Lösen des GAP. Der prinzipielle Aufbau eines genetischen Algorithmus, sowie eine

nähere Beschreibung der einzelnen Komponenten ist in Kapitel 4 enthalten. Kapitel 5 enthält eine detailierte Beschreibung eines Ansatzes von Chu und Beasley. In Kapitel 6 wird die Anwendung des genetischen Algorithmus für das GAP, sowie alle verwendeten Heuristiken, beschrieben. Anschließend werden die Ergebnisse in Kapitel 7 von den in dieser Arbeit gewählten Ansätzen mit anderen, aus der Literatur bekannten Ansätzen verglichen und interpretiert. Die Implementation des Programms GAP:S wird im Kapitel 8 vorgestellt. Alle möglichen Aufrufparameter und Benutzerhinweise sind ebenfalls in diesem Kapitel zusammengefaßt. Im anschließenden Kapitel wird eine Zusammenfassung der vorliegenden Arbeit und Ergebnisse gebracht und im Anhang (Kapitel 10) werden wichtige Begriffserklärungen geliefert.

Kapitel 2

Generalised Assignment Problem

Unter dem *Generalised Assignment Problem* (**GAP**) versteht man das kombinatorische Optimierungsproblem, bei dem eine Menge von Aufgaben (*jobs*) auf eine Menge von Maschinen (*agents*) derart verteilt werden soll, sodaß jede Aufgabe einer Maschine zugeordnet wird und Gesamtkosten minimal sind. Dabei sind zusätzlich Resourcenbeschränkungen der einzelnen Maschinen zu berücksichtigen. Jede Maschine hat als limitierenden Faktor eine beschränkte Kapazität und mit jeder Aufgabe sind unterschiedliche Resourcenbedürfnisse und Kosten abhängig von der jeweiligen Maschine verbunden.

Das GAP wird formal wie folgt definiert:
Sei $I = \{1, 2, \ldots, m\}$ eine Menge von m Maschinen (agents), und $J = \{1, 2, \ldots, n\}$ eine Menge von n Aufgaben (jobs). Für alle $i \in I$ und $j \in J$ seien ferner gegeben:

- Kosten $c_{ij} \geq 0$ für die Zuweisung von Aufgabe j an Maschine i;
- Resourcenbedürfnisse $r_{ij} \geq 0$ von Maschine i zur Erfüllung von Aufgabe j;
- Verfügbare Kapazitäten $b_i \geq 0$ von Maschine i;
- Die Zuweisung von Aufgaben an Maschinen werden durch Variablen $x_{ij} \in \{0, 1\}$ beschrieben. $x_{ij} = 1$ bedeutet, daß die Aufgabe j von Maschine i erledigt wird; ansonsten gilt $x_{ij} = 0$.

Das GAP kann nun als ganzzahliges lineares Programm geschrieben werden:

$$\text{Minimiere} \quad s = \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij}, \tag{2.1}$$

$$\text{so daß} \quad \sum_{j=1}^{n} r_{ij} x_{ij} \leq b_i, \qquad i \in I, \tag{2.2}$$

$$\sum_{i=1}^{m} x_{ij} = 1, \qquad j \in J, \tag{2.3}$$

$$x_{ij} \in \{0, 1\}, \qquad i \in I,\ j \in J. \tag{2.4}$$

Der Ausdruck (2.1) stellt die zu minimierenden Gesamtkosten, die Zielfunktion, dar. Die Bedingungen (2.2) werden Kapazitätsbeschränkungen (*capacity constraints*) genannt und stellen sicher das der gesamte Resourcenverbrauch aller Aufgaben, die jeder Maschine zugewiesen sind, die verfügbare Kapazität dieser Maschine nicht überschreitet. Die Nebenbedingungen (2.3) werden Zuweisungsbedingungen (*assignment constraints*) genannt und stellen sicher das jede Aufgabe genau einer Maschine zugewiesen wird. Die Restriktionen (2.4) werden Ganzzahligkeitsbedingungen (*integrality constraints*) genannt und stellen sicher, daß nur ganze Aufgaben verteilt werden.

Oft tritt das GAP auch als Maximierungsproblem auf, bei dem es um die Maximierung von Profiten geht. Eine solche Maximierungsaufgabe kann durch einfache Vorzeichenumkehr der Zielfunktion jedoch in eine Minimierungsaufgabe umgewandelt werden.

Für eine piktographische Darstellung des GAP siehe Abbildung 2.1; die Pfeile repräsentieren die Zuordnung einer Aufgabe zu einer Maschine.

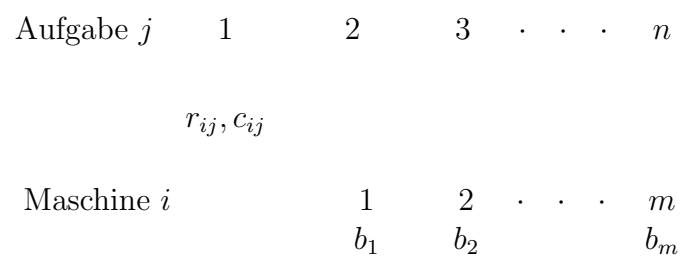

Abbildung 2.1: *Diagramm eines Generalized Assignment Problem*

Dem GAP liegen zahlreiche praktische Anwendungen zu Grunde, so z.B. das Zuweisen von Prozessen auf Computer in einem Rechnerverbund, das Zuordnen von Softwareentwicklungs-Aufgaben an Programmierer oder das Entwerfen von Telekommunikations-Netzen mit Kapazitätsbeschränkungen an den Schnittpunkten.

2.1 Suchraum

Ein Element des Suchraumes ist eine mögliche Zuordnung aller gegebenen Aufgaben auf alle gegebenen Maschinen. Die Menge der potentiellen Maschinen-Aufgaben Zuordnungen charakterisiert dabei den Suchraum.

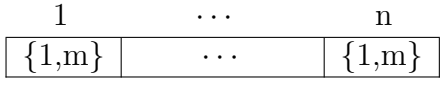

Abbildung 2.2: *Suchraum des GAP*

Das Verteilen von n Aufgaben auf m Maschinen entspricht hierbei n Ziehungen mit Wiederholung aus einer Urne mit m Elementen unter Berücksichtigung der Reihenfolge, ist also eine Kombination mit Wiederholung (mit Rangfolge). Für m Maschinen und n Aufgaben ergibt sich der Suchraum daher zu: $\mathcal{S} = \{1, \ldots, m\}^n$ und seine Größe ist daher $|\mathcal{S}| = m^n$.

Dieser Suchraum des GAP besteht aus zwei disjunkten Untermengen: einer gültigen Untermenge $\mathcal{F}$ und einer ungültigen Untermenge $\mathcal{U}$, bei der die Resourcen-Beschränkungen (*resource constraints*) nicht erfüllt sind.

2.2 Komplexität des GAP

Garey und Johnson [12] liefern den Beweis, daß das Generalised Assignment Problem NP-schwer ist. Es existiert daher mit großer Wahrscheinlichkeit kein deterministischer Algorithmus mit polynomialem Zeitaufwand. Deshalb kommt der Suche nach guten heuristischen Lösungen immer mehr Bedeutung zu. Das *GAP* kann weiters in ein *0/1 Multiple Knapsack Problem* übergeführt werden [4]; dieses ist ein klassisches NP-vollständiges Problem (Abschnitt 10.2).

Kapitel 3

Lösungsansätze

In diesem Abschnitt werden alternative Lösungsansätze besprochen. Hierbei wird in exakte und näherungsweise (heuristische) Methoden unterschieden. Zu den exakten Verfahren zählen *Branch-and-Bound* und das auf lineare Programmierung basierende *Branch-and-Cut* bzw. *Branch-and-Price*.

3.1 Lineare Programmierung

Die Simplex Methode [6] stellt das wohl verbreiteste Verfahren für die Optimierung von Problemen mit linearen Zielfunktionen und linearen Randbedingungen dar. Allerdings ist sie nur für Probleme mit kontinuierlichen Variablen anwendbar und daher nicht direkt für das GAP geeignet, da dieses ein diskretes Problem mit ganzzahligen Variablen, ein so genanntes *Integer Lineares Programm* (ILP), ist. Lineare Programmierung kann allerdings dazu verwendet werden, um eine untere Schranke für den optimalen Zielfunktionswert des ILP zu erhalten; dies geschieht mit Hilfe der LP-Relaxation wie in Abschnitt 3.1.1 beschrieben.

Bei der Simplex Methode wird die gesuchte Optimallösung nicht in einem Schritt gefunden, sondern iterativ und somit in mehreren Rechenschritten entwickelt. Das Simplex-Verfahren verwendet im Prinzip die Methode der Eckenprüfungen[1]. Die Idee ist es, von einer zulässigen Basislösung ausgehend durch geeignetes Umformen des Gleichungssystems von Eckpunkt zu Eckpunkt des Lösungspolyeders voranzuschreiten, so daß der Wert der Zielfunktion verbessert wird, solange bis das Optimum erreicht ist.

3.1.1 Das Problem mit der Ganzzahligkeit

Unter der LP-Relaxation eines ILP wie dem GAP versteht man das Problem, bei dem die Ganzzahligkeitsbedingungen weggelassen werden. In unserem Fall heißt das, daß alle $x_{ij} \in \{0, 1\},\ i = 1 \ldots m,\ j = 1 \ldots n$ durch $0 \leq x_{ij} \leq 1$ ersetzt werden. Der

[1] Bei diesem Lösungsansatz werden die Ecken des Lösungspolyeders überprüft. Das ist zumeist einfach, weil die Anzahl der Ecken durch die Zahl der aktiven Beschränkungen bestimmt wird, und es oft nur wenige Beschränkungen sind, die das Aussehen des Lösungsvieleckes tatsächlich mitbestimmen.

Zielfunktionswert s^* der optimalen Lösung dieses LP-relaxierten Problems ergibt eine untere Schranke für die optimale, ganzzahlige Lösung. Ist das tatsächliche Optimum eines ganzzahligen Optimierungsproblems unbekannt, so kann die Qualität einer beliebigen Lösung in Form des Abstands (*gap*) zum LP-Optimum angegeben werden. Der *gap* ist definiert als relative Differenz der Kosten der gegebenen Lösung zum LP-Optimum:

$$gap = \frac{|\ s - s^{LP}|}{s^{LP}} = |1 - \frac{s}{s^{LP}}|$$

mit s^{LP} = Optimale Kosten des LP-relaxierten Problems und
s = Kosten einer ganzzahligen Lösung.

3.2 Exakte Methoden

3.2.1 Algorithmus von Martello und Toth

Martello und Toth [16] verwenden einen enumerativen Algorithmus für die exakte Lösung des GAP. Dieser basiert auf einer Heuristik zum Finden einer guten Ausgangslösung, einer Reduktionsphase (*reduction phase*) und einem Branch-and-Bound Schema. Martello und Toth betrachten hierbei das GAP als Maximierungsaufgabe.

Die eigentliche Bearbeitung des GAP teilt sich hierbei in zwei Phasen. Im ersten Schritt wird mittels eines heuristischen Algorithmus eine Initiallösung generiert. Dazu definieren Martello und Toth vier verschiedene Gewichtsfaktoren, die als Distanzmaß der Heuristik dienen. Die beste Lösung der vier Gewichtsfaktoren, dient dann als Ausgangslösung für die nachfolgende Verarbeitung.

Der zweite Schritt besteht aus einem Branch-and-Bound Algorithmus mit Tiefensuche. Hierbei wird bei jedem Knoten des Entscheidungsbaums eine obere Schranke u berechnet; dies geschieht durch Lösen eines ,,relaxierten" Problems. Dabei wird das ,,relaxierte" Problem, durch Zerlegen des GAP in m unabhängige *0-1 Rucksack Probleme* (Abschnitt 10.1.6.1), durch Weglassen der Zuweisungsbedingungen (Gleichung 2.3), gewonnen. Dabei entspricht jede Maschine einem eigenen Rucksack-Problem; die Lösung dieses Rucksack-Problems wurde in [17] beschrieben.

Im Entscheidungsbaum wird immer dann verzweigt, wenn die Lösung des ,,relaxierten" Problems für das Ausgangsproblem keine gültige Lösung darstellt. Bei jedem Knoten im Entscheidungsbaum wird eine Reduktionsphase angewandt, um den Suchraum weiter einzugrenzen. Hierbei werden einzelne Aufgaben für bestimmte Maschinen ausgeschlossen (bzw. fixiert), wenn deren Belegung keine Verbesserung der derzeitigen Lösung mit sich bringt. Die derzeitige beste Lösung des Ausgangsproblems dient als untere Schranke. Sobald die obere Schranke (Profitpotenzial) nicht besser ist als die untere Schranke (bisherige, beste Lösung), wird ausgelotet, d.h. dieser Zweig wird im Entscheidungsbaum nicht mehr weiter verfolgt. Die optimale Lösung ist dann gefunden, wenn der (ausgelotete) Entscheidungsbaum durch Backtracking komplett abgearbeitet wurde.

Martello und Toth erbringen in [16] den Beweis, daß das GAP NP-vollständig ist. Daran ist aber schon der Nachteil des Branch-and-Bound Ansatzes ersichtlich, nachdem dieser nur für kleine Aufgabenstellungen in praktikabler Zeit zum Ziel führt.

3.2.1.1 Heuristik von Martello und Toth *MTH*

Martello und Toth stellen in ihrer Arbeit einen Algorithmus vor, der eine heuristische Lösung für das GAP liefert; diese dient als Ausgangslösung für einen Branch-and-Bound Algorithmus. Aufgabe der Heuristik ist es in erster Linie eine gültige Lösung für das GAP zu finden.

Im Rahmen der Heuristik werden Nützlichkeitsmaße μ_{ij} (auch Gewichtsfaktoren genannt) definiert, die das Bestreben (den Nutzen) der Zuweisung von Aufgabe i auf Maschine i charakterisieren sollen. Die Heuristik wird zunächst auf alle definierten Gewichtsfaktoren angewendet, und die beste Lösung dieser heuristischen Lösungen dient dann als initiale Lösung für den Branch-and-Bound Algorithmus.

Algorithmus 3.1 Heuristik von Martello und Toth *MTH*

Let: $\begin{cases} S[j] &= \text{the agent assigned to job } j \text{ in } S, \\ R_i &= \text{the accumulated resources assigned to agent } i \text{ in } S. \end{cases}$

1: /* Phase 1: try to generate feasible solution */
2: $\sum_{i \in I} R_i \leftarrow 0$; $F \leftarrow \{1, \ldots, n\}$;
3: **while** $F \neq \emptyset$ **do**
4: search for a job $j^* \in F$ which has the maximum difference d between the largest and the second largest μ_{ij} and fulfills $R_i + r_{ij} \leq b_i, \forall i \in I$;
5: **if** $j^* \neq nil$ **then**
6: $S[j^*] \leftarrow i^*$; /* i^* = the agent having the maximum μ_{ij^*} */
7: $R_{i^*} \leftarrow R_{i^*} + r_{i^*j^*}$;
8: $F \leftarrow F - \{j^*\}$;
9: **else**
10: exit; /* no feasible solution found */
11: **end if**
12: **end while**
13: /* Phase 2: improve quality of solution (profit) */
14: **for** $j = 1$ to n **do**
15: $i \leftarrow S[j]$;
16: $i^* \leftarrow \max_i\{p_{ij}, \text{ so that } R_i + r_{ij} \leq b_i, \forall i \in I\}$;
17: **if** $i^* \neq nil$ **then**
18: $S[j] \leftarrow i^*$;
19: $R_i \leftarrow R_i - r_{ij}$; $R_{i^*} \leftarrow R_{i^*} + r_{i^*j}$;
20: **end if**
21: **end for**

Bei dieser Heuristik werden zunächst alle noch nicht zugewiesenen Aufgaben iterativ betrachtet ($O(n)$ Operationen). Die Aufgabe j^* mit der maximalen Differenz zwischen dem größten und dem zweit-größten Gewichtsfaktor μ_{ij} über alle Maschinen wird bestimmt ($O(n \cdot m)$ Operationen). Die Aufgabe j^* wird dann der Maschine i zugewiesen, bei der das Maximum des Gewichtsfaktors μ_{ij^*} vorliegt. Der (Worst-Case) Aufwand für diesen ersten Schritt beträgt somit $O(n^2 \cdot m)$.

Hierbei kommen folgende Gewichtsfaktoren μ_{ij^*} zur Anwendung:

$$\mu_{ij} = \begin{cases} p_{ij} & (a) \\ p_{ij}/r_{ij} & (b) \\ -r_{ij} & (c) \\ -r_{ij}/a_i & (d) \quad (a_i = \text{verbliebene Restkapazität der Maschine } i) \\ -r_{ij}/b_i & (e) \quad (b_i = \text{verfügbare Gesamtkapazität der Maschine } i) \end{cases}$$

Martello und Toth haben die Gewichtsfaktoren (a - d) definiert; in der Literatur wird auch oft (e) angegeben (Testläufe ergaben, daß die Unterschiede marginal sind).

Im zweiten Schritt der Heuristik wird versucht die gefundene Lösung durch lokale Vertauschungen (*local exchange procedure*) noch weiter zu verbessern. Die Idee dahinter ist, daß durch gezielte Vertauschungen (*shift procedure*) einzelner Aufgaben j eine bessere Lösung gefunden wird. Bei der Vertauschung werden alle Aufgaben j ($O(n)$ Operationen), unter Berücksichtigung der Kapazitätsrestriktionen, betrachtet und der Maschine i zugewiesen, wo der Profit p_{ij} am höchsten ist ($O(m)$ Operationen).

Die (Worst-Case) Gesamtkomplexität für diese Heurstik ergibt sich somit zu:
$O(n^2 \cdot m + n \cdot m) = O(n^2 \cdot m)$.

3.2.1.2 Reduktionsphase

Bei der Reduktionsphase wird die momentan beste Lösung des Ausgangsproblems dazu verwendet durch das exakte Fixieren von Variablen, den Suchraum weiter einzuschränken. Bestimmte Variablen x_{ij} können gezielt auf den Wert 0 bzw. 1 gesetzt werden und helfen dadurch die Größe des Problems weiter zu verringern. Hierbei werden einzelne Aufgaben für bestimmte Maschinen ausgeschlossen (bzw. fixiert), wenn deren Belegung keine Verbesserung der derzeitigen Lösung mit sich bringt. Hierbei werden folgende zwei Fälle unterschieden:

> $x_{ij} := \mathbf{0}$ (keine Zuweisung der Aufgabe j zu Maschine i)
> Mittels Exklusionsverfahren können all jene Maschinen i für die Aufgabe j ausgeschlossen werden, die keine Verbesserung der derzeitigen Lösung darstellen. Dies geschieht durch gezieltes Fixieren von Variablen x_{ij} auf den Wert 0, für all jene Variablen x_{ij}, die, durch Setzen auf den Wert 1, eine schlechtere, obere Schranke u ergäben als die derzeit gefundene, beste Lösung (untere Schranke).
> Ebenso scheiden per se all jene Maschinen i aus, deren verfügbare Kapazität b_i nicht ausreicht um die Aufgabe j abzuarbeiten.
>
> $x_{ij} := \mathbf{1}$ (Zuweisung der Aufgabe j zu Maschine i)
> Eine Aufgabe j kann genau einer von m Maschinen zugewiesen werden. Verbleibt durch Ausnullen ($x_{ij} := 0$) nur mehr eine einzige Maschine i übrig, so wird dieser Maschine i die Aufgabe j zwingend zugewiesen (einzig verbliebene Möglichkeit); die Variable x_{ij} wird somit auf den Wert 1 gesetzt.

Diese Reduktionsphase wird iterativ angewandt, solange bis keine weiteren Reduktionen mehr vorgenommen werden können.

3.2.1.3 Verzweigungsstrategie

Für die Verzweigungsstrategie wird die ,,relaxierte" Form des Problems betrachtet (durch Weglassen der Zuweisungsbedingung 2.3), d.h. es darf ein und die selbe Aufgabe auf mehrere Maschinen zugewiesen sein. Jede Maschine wird hierbei als einfaches 0/1 Rucksackproblem (*Single 0-1 Knapsack Problem*) betrachtet, d.h. es gibt genau m 0/1-Rucksack-Probleme die für die ,,relaxierte" Form der Maximierungsaufgabe gelöst werden müssen.

Formulierung des einfachen 0-1 Rucksack-Problem K_i:

$$\text{Maximiere} \quad u_i = \sum_{j=1}^{n} p_{ij} x_{ij} \tag{3.1}$$

$$\text{so daß} \quad \sum_{j=1}^{n} r_{ij} x_{ij} \leq b_i, \tag{3.2}$$

$$x_{ij} \in \{0,1\} \qquad \forall j \in \{1,\ldots,n\}; \tag{3.3}$$

Daraus folgt eine entsprechende obere Schranke u (*upper bound*) von:

$$u = \sum_{i=1}^{m} u_i. \tag{3.4}$$

Die Summe der einzelnen Zielfunktionswerte u_i, aller Rucksack Probleme K_i, stellt somit eine obere Schranke u für den Zielfunktionswert der Maximierungsaufgabe dar (maximal zu erreichender Profit). Die obere Schranke u_i wird durch Lösen des 0/1 Rucksackproblems berechnet; für die optimale Lösung dieses Problems gibt es effiziente Verfahren [17], [26].

Ist die Lösung dieses ,,relaxierten" Problems auch für das ursprüngliche Problem inklusive der Zuweisungsbedingung (Bedingung 2.3) gültig, so werden keine Nachfolgeknoten erzeugt (der Zweig wird ausgelotet). Es liegt somit eine gültige Lösung für die ursprüngliche Optimierungsaufgabe vor. Ist diese Lösung auch besser als die bisherige beste Lösung, so liegt ebenfalls eine neue untere Schranke vor.

Sobald die obere Schranke nicht besser ist, als die untere Schranke (bisherige beste Lösung des Ursprungsproblems) wird dieser Zweig nicht mehr weiter verfolgt, da dieser die aktuell beste Lösung nicht mehr übertreffen kann.

Nachdem zum Finden einer gültigen Lösung des Ursprungsproblems auch die Zuweisungsbedinung erfüllt sein muß, kann diese zur Verbesserung der oberen Schranke herangezogen werden. Die obere Schranke u kann durch Berücksichtigen der Distanz (Strafe), die zum Erreichen der Zuweisungsbedingung (2.3) notwendig ist, noch weiter verbessert werden. Die Distanz stellt ein Mindestmaß dar, das unbedingt notwendig ist, um eine ungültige Lösung in eine gültige Lösung überzuführen.

Aus allen Aufgaben $j \in \overline{J}$ (Aufgaben ohne Zuordnung zu Maschinen und Aufgaben mit mehrfacher Zuweisung auf Maschinen), die eine Verletzung der Zuweisungsbedingung mit sich bringen, wird dann die Aufgabe $\hat{j}$ ausgewählt, die die größte, unbedingte Strafe nach sich zieht. Dabei ist l_j die Distanz der Aufgabe j zum ,,Erreichen" einer gültigen Lösung (Zuweisungsbedingung 2.3 muß erfüllt sein).

$$l_{\hat{j}} = \max_{j \in \overline{J}} \{l_j\} \qquad \overline{J} = \{j | \sum_{i \in I} x_{ij} \neq 1\}$$

Dies ist auch einsichtig, da die Bedingungen für alle Aufgaben erfüllt sein müssen, also auch für die Aufgabe mit der größten Distanz (Strafe) von einer gültigen Lösung. Die verbesserte, obere Schranke u' ergibt sich daher zu:

$$u' = u - l_{\hat{j}} = \sum_{i \in I} u_i - \max_{j \in \overline{J}} \{l_j\}.$$

Im Entscheidungsbaum wird dann die Aufgabe $l_{\hat{j}}$ zum Verzweigen herangezogen. Durch Auswahl der Aufgabe $l_{\hat{j}}$ mit der größten Distanz, wird der Suchraum stärker begrenzt und andere Zweige eher ausgelotet.

Die Verzweigungsstrategie unterscheidet hierbei zwei Fehlerfälle:

1. eine Aufgabe j ist keiner Maschine zugewiesen, also $\sum_{i \in I} x_{ij} = 0$,
2. eine Aufgabe j ist mehr als einer Maschine zugewiesen, also $\sum_{i \in I} x_{ij} = \overline{m} > 1$.

Bei fehlender Zuweisung (Verletzung vom Typ 1, Abbildung 3.1), wird auf allen Maschinen eine Verzweigung generiert, d.h. m Knoten für die Zuweisung auf m Maschinen. Bei Mehrfach-Zuweisung (Verletzung vom Typ 2, Abbildung 3.2), werden die in Frage kommenden Maschinen belegt, d.h. $\overline{m}$ Verzweigungen für $\overline{m}$ Maschinen werden generiert.

$$\ldots$$

$$\begin{array}{cc} x_{1\hat{j}} = 1 & x_{m\hat{j}} = 1 \\ x_{i\hat{j}} = 0 \ \forall i \neq 1 & x_{i\hat{j}} = 0 \ \forall i \neq m \end{array}$$

Abbildung 3.1: *Verzweigungsstrategie bei fehlender Zuweisung*

$$\ldots$$

$$\begin{array}{cc} x_{i_1\hat{j}} = 1 & x_{i_{\overline{m}}\hat{j}} = 1 \\ x_{i\hat{j}} = 0 \ \forall i \neq i_1 & x_{i\hat{j}} = 0 \ \forall i \neq i_{\overline{m}} \end{array}$$

Abbildung 3.2: *Verzweigungsstrategie bei Mehrfachzuweisung*

3.2.2 Branch-and-Price

Savelsbergh [25] hat einen Algorithmus präsentiert, der auf einer *Set Partitioning* Formulierung des GAP beruht. Hierbei kommen *Column Generation* und Branch-and-Bound Techniken zur Anwendung. Nachdem die Column Generation in jedem Knoten des Entscheidungsbaums zur Anwendung kommt, wurde dieser Algorithmus von Savelsbergh *Branch-and-Price* Algorithmus getauft.

Bei einem Branch-and-Price Algorithmus, wird nur eine Teilmenge der Variablen (*columns*) mittels Column Generation bearbeitet, da die Anzahl der Variablen zu groß ist und die meisten Variablen in der optimalen Lösung sowieso nicht vorkommen. Der Column Generation Algorithmus löst nur die LP-Relaxation des *Integer Programs*. Die Lösung für das LP stellt nicht unbedingt eine gültige Lösung für das IP dar (die Ganzzahligkeitsbedingungen sind verletzt). Daher wird der Column Generation Algorithmus mit einem impliziten Aufzählungsverfahren (Branch-and-Bound) kombiniert, um eine optimale Lösung für das IP zu finden.

Column Generation ist ein Kalkulationsschema zur Lösung umfangreicher linearer Programme (LPs). Die LP Relaxation der disaggregierten Formulierung (Set Partitioning) des GAP, kann aufgrund der exponentiellen Anzahl von Variablen (columns) nicht direkt gelöst werden. Daher bedient man sich der Lösung eines beschränkten Ausgangsproblems. Ein beschränktes Problem mit einer Teilmenge der Variablen kann unter Verwendung des Simplex Verfahrens direkt gelöst werden.

Anschließend wird überprüft, ob die LP Lösung optimal ist; dies geschieht durch Lösen des sogenannten *Pricing Problem*. Hierbei wird überprüft ob durch Hinzufügen zusätzlicher Variablen, die derzeit nicht im LP enthalten sind, die Lösung weiter verbessert werden kann. Wurde eine solche Variable gefunden, d.h. ihr reduzierter Preis ist positiv, so wird in einem iterativen Prozeß diese Variable dem LP hinzugefügt, dieses reoptimiert und anschließend wieder geprüft, ob die derzeitige Lösung noch weiter verbessert werden kann. Wird die LP Relaxation durch Column Generation gelöst, so ist jedoch nicht sichergestellt, daß die Lösung auch ganzzahlig ist.

Eine Verzweigung im Entscheidungsbaum findet genau dann statt, wenn eine optimale Lösung für die LP Relaxation vorliegt (d.h., es gibt keine Variablen, die die LP Lösung noch weiter verbessern), diese allerdings für das Integer Problem keine gültige Lösung darstellt, also die Ganzzahligkeitsbedingungen nicht erfüllt sind.

3.2.2.1 Column Generation Algorithmus

Der Column Generation Algorithmus löst die LP Relaxation (*linear programming relaxation*) eines gemischten ILP (*mixed integer optimization problem*).

Der *Column Generation* Ansatz wird dann gewählt, wenn das ursprüngliche lineare Problem (*master problem*) zu viele Variablen enthält, um dieses explizit zu lösen. Daher betrachtet man zunächst nur eine kleine Teilmenge des ursprünglichen Problems (*restricted master problem*) und berechnet eine optimale Lösung für dieses Problem. Anschließend wird überprüft, ob durch das Hinzufügen von zusätzlichen Variablen (*columns*), die im derzeitigen LP nicht enthalten sind, die Lösung noch weiter verbessert werden kann.

Ob die aktuelle Lösung bereits optimal ist bzw. ob Variablen vorhanden sind, die die

Lösung noch weiter verbessern, kann durch Berechnung der sogenannten reduzierten Kosten (*reduced cost*), festgestellt werden. Die Berechnung der reduzierten Kosten wird auch *pricing* genannt. Positive reduzierte Kosten bedeuten, daß die Lösung noch weiter verbessert werden kann. Ist eine Variable mit positiven reduzierten Kosten vorhanden, so wird diese dem LP hinzugefügt, dieses reoptimiert und anschließend geprüft, ob diese neue Lösung noch weiter verbessert werden kann. Dieser Vorgang wiederholt sich solange, bis keine Variablen mehr mit positiven reduzierten Kosten vorliegen (*„all variables price out correctly"*). Diese letzte Lösung stellt auch eine Lösung für das Ausgangsproblem dar.

Das Verfahren zum Generieren neuer Variablen, die positive reduzierte Kosten haben, wird *Column-Generation*[2] genannt. Zum Lösen des LP $\max\{c^T x \mid Ax \leq b,\ x \geq 0\}$ kommt ein Column Generation Algorithmus (Algorithmus 3.2) zur Anwendung.

Algorithmus 3.2 Generischer Column Generation Algorithmus

1: Select a small subset J of the variables $\{1, \ldots, n\}$.
2: Obtain an optimum basic solution $\overline{x}_J$ of the LP /* use of simplex method */
$c_J^T \overline{x}_J = \max\{c_J^T x_J \mid A_J x_J \leq b,\ x_J \geq 0,\ x_J \in \mathbb{R}^{|J|}\}$
3: **if** *reduced cost* $r_i \leq 0$ for all variables $i \in \{1, \ldots, n\} \setminus J$ **then**
4: Stop. /* „all variables price out correctly" */
5: **end if**
6: Add a column i^* with $r_{i^*} > 0$ to J.
7: **goto** Step 2.

Die lineare Optimierungsaufgabe $\max\{c^T x \mid Ax \leq b,\ x \geq 0\}$ wird Hauptaufgabe (*master problem*) genannt, während das Lineare Programm $\max\{c_J^T x_J \mid A_J x_J \leq b,\ x_J \geq 0,\ x_J \in \mathbb{R}^{|J|}\}$, das nur aus einer Teilmenge der Spalten (Variablen) der Hauptaufgabe besteht, beschränkte Hauptaufgabe (*restricted master problem*) genannt wird.

Zur Auswahl der nächsten Variable (Spalte) können folgende Column-Generation Strategien angewandt werden.

- *best-positive*
  Dabei wird die Variable mit den höchsten reduzierten Kosten ausgewählt.
- *first-positive*
  Die erste Variable mit positiven reduzierten Kosten wird ausgewählt.
- *all-positive*
  Es werden alle Variablen mit positiven reduzierten Kosten ausgewählt.

[2] In der LP-Theorie wird eine Matrixdarstellung für Lineare Programme gewählt, dabei entsprechen die Zeilen (*row*) den einzelnen Nebenbedingungen (Restriktionen) und die Spalten (*column*) den Variablen, die in der Zielfunktion vorkommen.

3.2.2.2 Pricing Problem

Beim *Pricing Problem* geht es um die Bestimmung welche Variablen für die Lösung relevant bzw. irrelevant sind (also nicht in der Lösung vorkommen). Entsprechend der LP-Theorie geschieht dies durch Berechnen der ,,reduzierten Kosten" (*reduced cost*, Grenzertrag[3]) aller nicht aktiven Variablen. Diese werden auch oft die Ersatzkosten (*opportunity cost*) der Variable genannt. Die reduzierten Kosten einer Variable liefern eine Schätzung, um wieviel sich der Zielfunktionswert durch Hinzufügen dieser Variable ändert.

In einem linearen Programm der Form $\max\{c^T x \mid Ax \leq b,\ x \geq 0\}$ kann eine Variable mit positiven reduzierten Kosten die aktuelle Lösung noch verbessern. In diesem Fall wird die Variable dem LP hinzugefügt, das LP reoptimiert und iteriert. Wenn keine Variable positive reduzierte Kosten hat, dann stellt die aktuelle optimale Lösung auch eine Lösung für das ursprüngliche Problem dar.

3.2.2.3 Verzweigungsstrategie und Selektionsstrategie

Eine Lösung für das LP ist nicht zwangsläufig auch eine gültige Lösung für das IP. Eine Verzweigung im Entscheidungsbaums findet daher dann statt, wenn die Lösung für das LP keine gültige Lösung für das IP darstellt; also die Ganzzahligkeitsbedingungen nicht erfüllt sind.

Verzweigungsstrategien für 0–1 Lineare Programme basieren auf dem Fixieren von Variablen, entweder von einzelnen Variablen oder einer Menge von mehreren Variablen; dementsprechend *variable dichotomy* oder *GUB dichotomy* genannt[4]. Die Verzweigungsstrategie bestimmt, wie die aktuelle Menge an gültigen Lösungen in zwei oder mehrere kleinere Teilmengen aufgeteilt wird. Die Verzweigungsstrategie bestimmt allerdings nicht, welches Teilproblem als nächstes gelöst werden soll.

Dies ist Aufgabe der Selektionsstrategie. Es wird unterschieden zwischen *depth-first search* und *best-bound search*. Ziel der Tiefensuche ist es möglichst gute Lösungen möglichst rasch zu finden, um Knoten frühzeitig auszuloten und damit die Größe des Entscheidungsbaums zu reduzieren. Die Erfahrung zeigt, daß es wahrscheinlicher ist, gültige Lösungen tief im Entscheidungsbaum zu finden anstatt bei Knoten in Wurzelnähe. Die Best-Bound Suche bearbeitet den Knoten mit der besten Schranke zuerst, da dieser sowieso betrachtet werden muß, um die Optimalität zu beweisen.

[3] G. Wöhe [28] verwendet folgende Definition: ,,Als Grenzertrag bezeichnet man den Zuwachs zum Gesamtertrag, der sich durch Einsatz der jeweils letzten (unendlich kleinen) Mengeneinheit eines Produktionsfaktors ergibt."

[4] Die Dichotomie beschreibt in der Botanik eine gabelartige Verzweigung bzw. in der Philosophie eine Zweiteilung oder eine Gliederung nach zwei Gesichtspunkten. In der Naturwissenschaft wird zwischen variable dichotomy und GUB dichotomy unterschieden. Erstere wirkt nur auf eine einzelne Variable, während letztere auf eine Gruppe von mehreren Variablen (*Generalized-Upper-Bound*) angewandt wird.

3.2.2.4 Primäre Heuristik

Die Größe des Entscheidungsbaums kann durch das Vorhandensein guter, gültiger Lösungen beträchtlich eingeschränkt werden. Daher wird der Branch-and-Price Algorithmus mit einem Näherungsalgorithmus kombiniert. Die Heuristik, die hierbei zur Anwendung kommt, stellt eine Kombination der Algorithmen von Martello und Toth und von Jörnsten und Nasberg [15] dar und wird in jedem Knoten des Entscheidungsbaums angewandt.

Das Näherungsverfahren basiert auf dem Algorithmus von Martello und Toth, erweitert um Vertauschungsoperationen auf lokaler Ebene, für den Fall, daß keine gültige Lösung gefunden wurde. Der Algorithmus von Martello und Toth wird im einem eigenen Kapitel (Abschnitt 3.2.1) näher beschrieben. Als Maß für das Verlangen der Zuweisung eines Jobs zu einem Agent wird die aktuelle LP-Lösung herangezogen.

Die Vertauschungsoperationen beruhen auf dem Algorithmus von Jörnsten und Nasberg. Hierbei kommt ein Maß für die Ungeeignetheit einer Zuweisung zur Anwendung, um eine ungültige Lösung mittels Vertauschungsoperationen in eine gültige Lösung zu überführen (restriktions-orientiert). Daran anschließend werden die lokalen Vertauschungsoperationen nocheinmal angewandt; diesmal um die Qualität der gültigen Lösung noch weiter zu verbessern (zielfunktions-orientiert).

3.2.2.5 Realisierung

Die Lösung von Savelsbergh baut auf einer Set Partitioning Formulierung des GAP auf – diese stellt eine disaggregierte Form der üblichen Formulierung dar – und enthält eine exponentielle Anzahl von Variablen.

Um das Branch-and-Price Verfahren zu starten, bedarf es eines Ausgangsproblems für den Column-Generation Algorithmus. Dazu wird eine kleine Teilmenge aller möglichen Variablen ausgewählt. Savelsbergh wählte für dieses initiale, beschränkte Ausgangsproblem genau eine Variable (Aufgabe) pro Maschine.

Daran anschließend wird das Column Generation Verfahren angewendet, um eine gültige Lösung für das LP zu erhalten. Nachdem die Lösung für das LP (meist) eine ungültige Lösung für das IP darstellt, findet eine Verzweigung im Entscheidungsbaum statt. Hierbei wird auf jenen Variablen verzweigt, für die die Ganzzahligkeitsbedingungen verletzt sind, gemäß der gewählten Verzweigungsstrategie (Abschnitt 3.2.2.3). Dabei wird eine heuristische Lösung (Abschnitt 3.2.2.4) für das IP in jedem Knoten des Entscheidungsbaums generiert, um diesen so früh wie möglich auszuloten.

Gemäß der Selektionsstrategie findet die Bearbeitung im nächsten Teilbaum statt. Dabei findet in einem iterativen Prozeß wieder das Column Generation Verfahren Anwendung, um für den Knoten des aktiven Teilbaum eine neue Lösung des LP zu erhalten. Daran reiht sich wieder die Verzweigung im Entscheidungsbaum, solange bis eine gültige Lösung für das IP in den Endknoten des Baums vorliegt. Dieser Vorgang wiederholt sich in allen Teilbäumen bis mittels Backtracking der ganze Entscheidungsbaum abgearbeitet wurde.

3.3 Heuristische Methoden

Zu den heuristischen Methoden zählen alle Verfahren, die nicht garantieren, eine tatsächlich optimale Lösung zu liefern. Heuristische (informierte) Suchverfahren verwenden problemspezifische Zusatzinformationen um eine Lösung zu erzeugen. Dazu gehören unter anderem Verfahren, die auf genetischen Algorithmen aufbauen. Zu dieser Kategorie zählt die Arbeit von Chu und Beasley, welche in einem eigenen Kapitel näher besprochen (Kapitel 5) wird.

Die Heuristik von Martello und Toth [16] (Abschnitt 3.2.1.1) genauso wie die Heuristik von Jörnsten und Nasberg [15] dienen zum Finden einer möglichst guten Ausgangslösung für die anschließende Weiterverarbeitung und zählen ebenfalls zu dieser Gruppe.

Kapitel 4

Genetische Algorithmen

4.1 Allgemein

Genetische Algorithmen (GA) sind computer-basierte Problemlösungssysteme, die berechenbare Modelle von natürlichen, evolutionären Prozessen als Schlüsselelemente verwenden [11]. Die Idee stammt von Holland [14] aus den 70er Jahren und wurde unter anderem von Goldberg [13], De Jong und Michalewicz [18] weiterentwickelt.

Genetische Algorithmen basieren auf einer biologischen Metapher. Hierbei wird das Lernen als Wettbewerb zwischen sich entwickelnden Kandidatenlösungen in einer Population verstanden. Eine ,,Fitness" Funktion evaluiert jede Lösung, um zu entscheiden, ob diese an der Generierung von Nachfolgelösungen teilnimmt. Dieses Ableiten einer neuen Population von Lösungskandidaten erfolgt mittels Operationen wie Rekombination und Mutation analog dem Transfer von Genen bei natürlicher Reproduktion.

Genetische Algorithmen sind für sehr viele Arten von komplexen Optimierungsaufgaben gut geeignet:

- Keine grundsätzlichen Einschränkungen bezüglich der zu optimierenden Funktion (wie z.B. Stetigkeit, Ableitbarkeit oder Dimensionalität).
- Benötigt keine besondere Information über den Suchraum (wie z.B. Ableitungen).
- Vor allem für Probleme mit einem sehr großen, komplexen Suchraum geeignet, wo eine Optimumsuche durch Aufzählung aller möglichen Lösungen nicht mehr möglich ist.
- Globale Sichtweise des Verfahrens – es wird grundsätzlich das globale Maximum gesucht und nicht nur das nächste lokale.
- Auffinden der optimalen Lösung (bzw. einer annähernd gleichwertigen) kann aber nicht garantiert werden.

4.2 Kodierung

Das Vokabular für genetische Algorithmen wurde aus der Genetik entlehnt:

- Die **Population** besteht aus einer fixen Anzahl von Lösungen des Problems, Individuen genannt.
- Jedes **Individuum** repräsentiert einen Punkt im Suchraum und enthält die Parameter für eine potentielle Lösung in kodierter Form. Individuen werden auch, in Anlehnung an die Natur, als *Chromosomen* bezeichnet.
- Für die Kodierung eines Individuums wird ein aus **Genen** bestehender String meist fixer Länge verwendet. Jedes Gen kann einen Wert einer diskreten Wertemenge annehmen.
  Im klassischen Genetischen Algorithmus von J. H. Holland [14] werden alle zu optimierenden Variablen in binärer Form kodiert und zu einem binären String als Chromosom zusammengefügt. Nicht immer ist diese Art der Darstellung zweckmäßig, daher kommen auch Repräsentation mit Alphabeten höherer Kardinalität vor, wie in dieser Arbeit.

4.3 Aufbau eines Genetischen Algorithmus

Genetische Algorithmen sind gerichtete Zufallsverfahren die probabilistische Entscheidungen treffen. Algorithmus 4.1 zeigt den prinzipiellen Aufbau eines GAs.

Zuerst wird die Ausgangspopulation erzeugt und bewertet. Solange die Abbruchbedingung nicht erfüllt ist, wird mittels Selektion eine neue Population $P(t) = \{S_1^t, ..., S_N^t\}$ gebildet. Die Individuen S_i^t der neuen Population $P(t)$ werden nun den genetischen Operationen, Rekombination (Crossover) und Mutation, unterzogen. Durch diese Transformation entsteht eine Population neuer Lösungen, die als Eltern in der nächsten Generation fungieren. Schließlich wird die neue Population bewertet und falls die Abbruchbedingung erfüllt ist, ist eine hinlänglich gute Lösung gefunden; anderenfalls wiederholt sich dieser Vorgang von vorne.

Algorithmus 4.1 Prinzip eines Genetischen Algorithmus

```
1:  t ← 0;
2:  initialize(P(t));                          /* initial population */
3:  evaluate(P(t));
4:  while ( not termination-condition ) do
5:     t ← t + 1;
6:     Q_s(t) ← select(P(t − 1));              /* selection operator */
7:     Q_r(t) ← recombine(Q_s(t));             /* crossover operator */
8:     P(t) ← mutate(Q_r(t));                  /* mutation operator */
9:     evaluate(P(t));                         /* evaluate fitness */
10: end while
```

4.4 Initialisierung

Die Initialisierung der Ausgangspopulation erfolgt im allgemeinen durch Zufall. Bei vielen Problemen muß darauf geachtet werden, daß die generierten Lösungen auch zulässig sind.

4.5 Bewertung

Die **Fitnessfunktion** (*fitness function*) entscheidet, mit welcher Wahrscheinlichkeit ein Individuum an dem Prozeß der Erzeugung der Nachkommen teilnehmen darf. Die Fitnessfunktion liefert Werte, die für die Selektion geeignet sind. Die Fitness $f(S)$ eines Individuums S wird aus der Bewertungsfunktion durch Skalierung gewonnen: $f(S) = scale(g(S))$.
Die Fitnessfunktion $f(S_i)$ hat folgende Eigenschaften ($P = \{S_1, ..., S_N\}$):

- $f(S_i)$ ist für alle möglichen Individuen S_i berechenbar
- $f(S_i) \geq 0$
- aus $f(S_i) > f(S_j)$ folgt Individuum S_i ist besser als Individuum S_j

Die Fitnessfunktion $f(S)$ dient zur Bewertung aller Individuen einer Population (Evaluation). Sie entscheidet darüber, welche Lösungen in Folge „überleben“ und welche nicht. Der Wert dieser Funktion gibt also an, wie *fit* eine Lösung ist.
Die **Bewertungsfunktion** (*raw fitness*, Qualitätsfunktion) ist ein Maß für die Qualität einer Lösung (der Wertebereich ist an die Problemstellung angepaßt). Die Bewertungsfunktion mißt, wie nahe ein Individuum dem gesuchten optimalen Wert ist. Die Bewertungsfunktion $g(S)$ berechnet sich für ein Individuum S aus der Zielfunktion und der Straffunktion (falls Randbedingungen verletzt sind, hilft dieser Abschlag gültige Individuen gegenüber ungültigen ,,aufzuwerten”, Abschnitt 4.10).
Allgemeiner Aufbau der Bewertungsfunktion $g(S)$ unter Berücksichtigung der Straffunktion $p(S)$:

$$g(S) = h(S) + p(S) \qquad \text{(Maximierung: } g(S) = h(S) - p(S)\text{)}$$

Für gültige Individuen ist die Straffunktion $p(S)$ gleich null ($p(S) = 0, \forall S \in \mathcal{F}$).
Die **Zielfunktion** (*objective function*) entspricht dem Optimierungsziel des Optimierungsproblems. Die Zielfunktion $h(S)$ spezifiziert die zu optimierenden Zielkriterien; sie berechnet ein Gütemaß für die gegebenen Modellparameter.

4.6 Selektion

Die Selektion ermittelt die Zulassung von Individuen zur Reproduktion. Aus der aktuellen Population werden durch Selektion die Elternindividuen für die nächste Generation bestimmt. Die Selektion erfolgt meist zufallsgesteuert (stochastische Selektion), aber doch entsprechend dem Prinzip der natürlichen Auslese: bessere Individuen werden öfter bzw. mit größerer Wahrscheinlichkeit ausgewählt als schlechtere. Die Selektion „treibt“ im GA die Individuen in Richtung optimaler Lösung.

Je nach Optimierungsziel (Minimierung oder Maximierung) und dem Wertebereich der Fitnesswerte, muß die Fitness normalisiert werden. Verschiedene Arten der Normalisierung existieren: diese beruhen entweder direkt auf dem Fitnesswert selbst oder auf dem Rang des jeweiligen Individuums in der Population. Die Rang-basierte Normalisierung ist in speziellen Selektionsoperatoren integriert, wie der Tournament Selektion (Abschnitt 4.6.3.1). Die Fitness-basierte Normalisierung wird in eigenen Skalierungsroutinen (Abschnitt 4.6.2.1) durchgeführt, die vor der eigentlichen Selektion aufgerufen werden müssen.

4.6.1 Selektionsdruck

Ein Genetischer Algorithmus stellt eine parallele Suche in einem meist sehr großen Suchraum dar. Jedes Individuum der Population ist eine potentielle Lösung (ohne Aussage über deren Güte). Um möglichst schnell zu einem guten Ergebnis zu kommen, muß der Algorithmus einen möglichst großen Teil des Suchraumes überblicken. Vor allem in den Anfangsstadien ist es wichtig, daß die Population eine große Vielfalt aufweist, d.h. die verschiedenen Lösungen sollen möglichst unterschiedlich sein.

Der Selektionsdruck bestimmt wie sehr gute Individuen gegenüber schlechteren bevorzugt werden. Bei **zu hohem Selektionsdruck** werden gute Individuen zu sehr bevorzugt:

- rasche Vermehrung dieser (→ sogenannte *Superindividuen*)
- Vielfalt der Population wird geringer
- GA konvergiert oft vorzeitig gegen *lokales Optimum*

Bei **zu niedrigem Selektionsdruck** werden gute Individuen kaum bevorzugt:

- gute Individuen vermehren sich kaum
- schlechtere Individuen bleiben in Population
- Verfahren degeneriert zur Zufallssuche
- GA konvergiert nicht oder nur sehr langsam

Es gibt verschiedene Formen von Selektionsmechanismen; sie unterscheiden sich in ihrer Selektionswahrscheinlichkeit.

4.6.2 Fitnessproportionale Selektion

Bei dieser Methode ist die Selektionswahrscheinlichkeit für die Individuen proportional zu ihrer Fitness $f(S_i)$.

$$p_s(S_i) = \frac{f(S_i)}{\sum_{j=1}^{n} f(S_j)} \quad \text{mit} \quad \sum_{j=1}^{n} f(S_j) > 0$$

$p_s(S_i)$ ist die Wahrscheinlichkeit mit der ein Individuum S_i bei der Selektion ausgewählt wird, n entspricht der Populationsgröße. Diese Methode wird auch *Roulette-Wheel* Selektion genannt (Abbildung 4.1: Quelle: [21]).

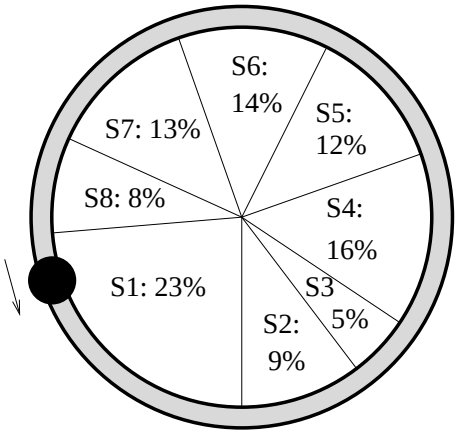

Abbildung 4.1: *Fitnessproportionale Selektion*

Jedes Individuum erhält einen Bereich, dessen Größe von der Selektionswahrscheinlichkeit $p_s(S_i)$ abhängig ist. Weiters gilt $\sum_{i=1}^{n} p_s(S_i) = 1$. Der „Kugel“ im Roulette entspricht eine zufällige Zahl im Intervall $[0, 1)$. Das Individuum, dessen Bereich die „Kugel“ überdeckt, wird selektiert.

4.6.2.1 Fitness-basierte Normalisierung

Folgende Skalierungsfunktionen werden häufig verwendet:

- **direkt:** $f(S_i) = g(S_i)$
  Es wird keine Trennung zwischen der Bewertungsfunktion und der Fitnessfunktion vorgenommen. Diese ist nur zulässig für Zielfunktionswerte die positiv sind (inkl. Null) und eignet sich daher nur zur Maximierung. In der Regel ist diese auf Fitness-proportionale Selektion nicht anwendbar, nachdem aufgrund der fehlenden Skalierung der sich ergebende Selektionsdruck ungeeignet ist. Haben alle Lösungen nahezu gleiche Fitness, werden die guten nicht entsprechend bevorzugt → Zufallssuche. Haben umgekehrt manche Individuen viel zu hohe Fitness im Vergleich zu den schlechteren, werden diese viel zu stark bevorzugt → Superindividuen.

- **linear:** $f(S_i) = a \cdot g(S_i) + b$
  Mit den beiden Konstanten $a,\ b$ ($a > 0$) kann das Verhältnis zwischen maximaler und durchschnittlicher Fitness gesteuert werden. Ist $a > 1$, so wird die Fitness von höher bewerteten Individuen stärker angehoben, d.h. der Abstand zwischen Minimal- und Maximalwert wird größer. Bei $a < 1$ wird der Abstand geringer.

- **geometrisch:** $f(S_i) = g(S_i)^p$
  Abhängig vom Problem werden höhere Fitnesswerte stärker ($p > 1$) oder schwächer gewichtet.

4.6.3 Rang-basierte Normalisierung

Die Rang-basierte Skalierung ist wie folgt definiert: $f(S_i) = a \cdot r(S_i)$. Diese Methode ist geeignet für Probleme, für die nur schwer eine konkrete Bewertungsfunktion angegeben werden kann bzw. die Verteilung der Werte der Bewertungsfunktion nicht durch einfache Skalierung gelöst werden kann. Bei der Rang-basierten Skalierung wird die Population gemäß der Zielfunktionswerte sortiert. Die Fitness die jedem Individuum zugeordnet wird, hängt ausschließlich von der Position des Rangs ab und nicht vom aktuellen Zielfunktionswert (die Selektionswahrscheinlichkeit errechnet sich aus dem Rang des Individuums). Eines der bekanntesten Verfahren, das auf diesem Ansatz aufbaut, ist die Tournament Selektion (Abschnitt 4.6.3.1).

4.6.3.1 Tournament Selektion

Turnier Selektion (*tournament selection*) ist eine sehr einfach zu implementierende und effiziente Selektionsmethode, bei der nur die Ordnung der Individuen einer Population, nicht aber ihre absoluten Fitneßwerte eine Rolle spielen. Um ein Individuum S zu selektieren, wählt man k Individuen zufällig (gleichverteilt) aus der Population aus und nimmt davon das beste (höchste Fitneß). Der Parameter k steuert somit den Selektionsdruck. Wird k erhöht, so steigt auch der Selektionsdruck.

$$p_s(S_i) = \left(1 - \left(1 - \frac{1}{n}\right)^k\right) \cdot \left(1 - \frac{i}{n}\right)^{k-1}$$

4.7 Rekombination (Crossover)

Die Rekombination ist der *primäre Operator* eines GA, der neue Informationen in die Population einbringt. Mittels Rekombination werden neue Individuen aus den selektierten Eltern erzeugt. Über die Rekombination werden Informationen (Gene) zwischen zwei potentiellen Lösungen ausgetauscht. Allgemein werden beim Crossingover Bruchstücke zwischen zwei Chromosomen ausgetauscht. Die Rekombination wird entweder für alle selektierten Individuen oder für den größten Teil davon zufallsgesteuert durchgeführt. Die Paarbildung erfolgt ebenso wie die Wahl der Crossover-Punkte (*crossover points*) durch Zufallsauswahl.

Es gibt unterschiedliche Arten von Crossover Operatoren; zwei davon zeigen Abbildung 4.2 und Abbildung 4.3 (Quelle: [21]).

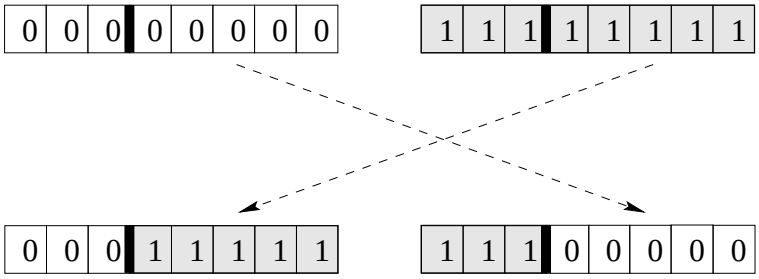

Abbildung 4.2: *1-Point Crossover*

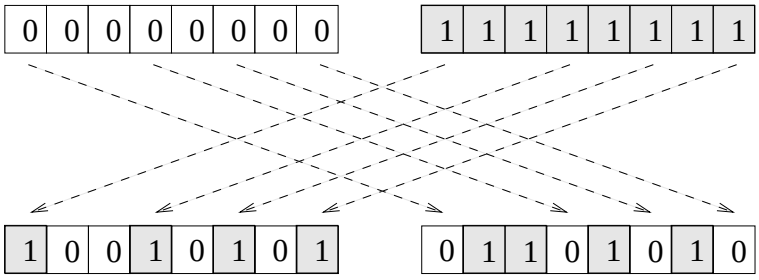

Abbildung 4.3: *Uniform Crossover*

Die Wahl des Crossover-Punkts beim *1-Point Crossover* (Abbildung 4.2) erfolgt nach dem Zufallsprinzip. Beim *2-Point Crossover* werden zwei Zufallszahlen (gleichverteilt) erzeugt. Diese indizieren jene Gene auf den Chromosomen, die das auszutauschende Bruchstück eingrenzen.

Beim *Uniform Crossover* (auch Zufalls-Schablone, Abbildung 4.3) wird für jedes Gen zufällig entschieden, von welchem Eltern-Individuum es übernommen wird. Bei dieser Form wird ein Vektor generiert, der für jede Genposition des Chromosoms eine Zufallszahl (0 oder 1) enthält. Eine 1 bedeutet, die Gene an der entsprechenden Position auf den Chromosomen werden ausgetauscht; eine 0 bedeutet, die Gene werden nicht getauscht.

4.8 Mutation

Die Mutation stellt den *sekundären Operator* dar und dient nur zum Einbringen von neuem bzw. verlorengegangenem Genmaterial (Genwerten) in die Population. Sie wird nur sehr selten und zufallsgesteuert ausgeführt. Wird sie zu oft angewandt, so degeneriert der Suchvorgang zu einer kompletten Zufallssuche.

Auch bei der Mutation gibt es verschiedene Varianten (Quelle: [21]). Die Auswahl des zu mutierenden Gens erfolgt bei der *Flipmutation* (Abbildung 4.4) zufallsgesteuert. Bei einem Bitstring wird das ausgewählte Bit (Gen) einfach negiert. Bei der *Swapmutation* (Abbildung 4.5, auch *Exchange-Based Mutation* genannt) wird der Inhalt der durch Zufall bestimmten Gene vertauscht.

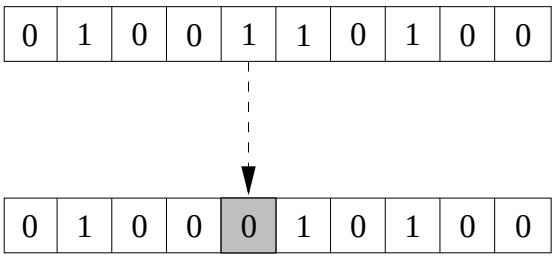

Abbildung 4.4: *Flipmutation*

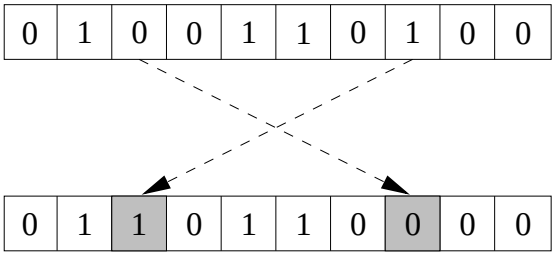

Abbildung 4.5: *Swapmutation*

4.9 Ersetzungsstrategien

4.9.1 Generational GA

Beim Generational Genetic Algorithm wird in jeder Generation die komplette Population durch eine Nachfolgegeneration ersetzt. Bei diesem Ansatz ist die Fitnessverlaufsfunktion des jeweils besten Genoms einer Generation nicht monoton (entspricht dem in Algorithmus 4.1 dargestellten).

4.9.2 Elitismus

Die Elitismusoption wurde von Goldberg [13] eingeführt, um eine gefundene beste Lösung nicht durch Zufall in der nächsten Generation zu verlieren, was durch den nicht-überlappenden Charakter der kompletten Populationsersetzung durchaus oft passieren kann. Die Elitismusoption prüft, ob das beste Individuum der aktuellen Generation einen schlechteren Fitnesswert als das beste Individuum der Elterngeneration hat. Ist dies der Fall, wird das beste Individuum der Vorgängergeneration in die aktuelle „hinübergerettet". Der Verlauf der Fitnessfunktion des jeweils besten Individuums über die Zeit ist dann monoton.

4.9.3 Steady State GA

Ein *Steady-State* GA generiert und ersetzt in jeder Generation nur jeweils ein Individuum der Population. Die am häufigsten verwendeten Methoden sind, die schlechteste Lösung in der Population zu eliminieren und dafür die neue Lösung in die Population aufzunehmen, oder ein zufällig gewähltes Individuum (unter Berücksichtigung von Elitismus) zu ersetzen.

Die Steady-State Methode hat den Vorteil, daß weniger Speicherplatz für den gesamten GA benötigt wird. Ferner können gefundene Individuen mit sehr hoher Fitneß, bereits beim Erzeugen des nächsten Individuums als Eltern verwendet werden. Genetische Algorithmen, die auf diese Strategie zurückgreifen, tendieren deshalb dazu, schneller zu konvergieren. Beim Steady-State GA können höhere Mutations- und Crossover-Wahrscheinlichkeiten verwendet werden, da gute Populationsmitglieder durch die gewählte Ersetzungsstrategie (Ersetzung der Individuen mit geringer Fitness) geschützt werden. Dadurch soll der Algorithmus einen stärker explorativen Charakter erhalten.

4.10 Randbedingungen

Randbedingungen (Restriktionen, *constraints*) stellen Beschränkungen des für die Modellparameter zulässigen Bereiches dar. Dies können einfache Schranken (Angabe von Minima und Maxima), lineare Kopplung von Parameterwerten oder nichtlineare Funktionen sein.

Ein Genetischer Algorithmus erzielt meist die besten Ergebnisse, wenn die Kodierung, die Initialisierung und die Operatoren für ein Problem so gewählt werden, daß alle möglichen Individuen auch gültige Lösungen darstellen, das heißt, daß alle Randbedingungen immer erfüllt sind. In manchen Fällen ist das aber nicht oder nur schwer möglich, sodaß ungültige Lösungen entstehen. Daher sollen diese ungültigen Individuen eine schlechtere Bewertung erhalten, damit sie sich nicht vermehren. Es sollte zwischen schwereren und leichteren Verstößen unterschieden werden, um die Schwere der Verletzung der Nebenbedingung berücksichtigen zu können. Meist wird für verletzte Randbedingungen von der Fitneß eine geeignet skalierte Strafe abgezogen [18].

Die **Straffunktion** (*penalty function*) enthält die verletzten Nebenbedingungen des Optimierungsproblems. Bei der Optimierung mit Gleichungs- und Ungleichungsnebenbedingungen, können diese über eine Straffunktion in die Zielfunktion integriert werden. Die Straffunktion $p(S)$ bestimmt einen Strafterm für das Verlassen des durch die Nebenbedingung(en) definierten zulässigen Bereichs. Diese ist innerhalb des zulässigen Bereichs gleich Null und außerhalb größer als Null (relativ zur Verletzung der Nebenbedingungen).

Anstatt die ungültigen Individuen mit einem Abschlag zu bestrafen, kann es auch sinnvoll sein, diese mittels einer problemspezifischen Reparaturfunktion (*repair*–Algorithmus) in gültige Individuen überzuführen.

4.11 Abbruchbedingung

Die Abbruchbedingung bzw. das Terminierungskriterium ist eine Regel, die besagt wann eine numerische Optimierung gestoppt werden soll. Häufig wird das Überschreiten einer maximalen Iterationszahl, die Verifikation einer gefundenen Optimallösung oder die Stagnation des Suchprozesses als Terminierungskriterium eingesetzt.

Die einfachste Variante ist das Erreichen einer vorgegebenen Anzahl von Generationen (absolute Terminierung).

Eine andere Variante ist die Berücksichtigung der Konvergenz (relative Terminierung). Dabei terminiert der GA, sobald in einer vorgegebenen Anzahl von Generationsschritten keine Verbesserung erzielt wurde.

4.12 Zusammenfassung

Genetische Algorithmen sind computer-basierte Problemlösungssysteme, die berechenbare Modelle von natürlichen, evolutionären Prozessen als Schlüsselelemente verwenden. Sie arbeiten mit einer Population von Strukturen, die sich nach den Regeln der Selektion, der Rekombination und der Mutation entwickeln. Sie sind für viele Arten von kombinatorischen Optimierungsproblemen einsetzbar. Genetische Algorithmen sind vor allem für Probleme mit sehr großen, komplexen Suchräumen geeignet, wo eine Optimumsuche durch Aufzählung aller möglichen Lösungen nicht mehr durchführbar ist. Oft stellen Genetische Algorithmen eine effiziente Suche dar, aber für das Auffinden der optimalen Lösung oder einer annähernd gleichwertigen kann nicht garantiert werden.

Kapitel 5

Lösungsansatz von Chu und Beasley

An dieser Stelle wird der genetische Algorithmus von Chu und Beasley [5] für das Generalised Assignment Problem beschrieben.

5.1 Repräsentation

Die der mathematischen Definition von Kapitel 2 entsprechende Darstellung des GAP ist die binäre Darstellung. Nachdem es $(m \cdot n)$ Entscheidungsvariablen gibt, entspricht dies einem Bit-String der Länge $m \cdot n$ (Abbildung 5.1).

(agent i, job j)	$(1,1)$	$\cdots$	$(1,n)$	$\cdots$	$(m,1)$	$\cdots$	(m,n)
$[i,j]$	1	$\cdots$	0	$\cdots$	0	$\cdots$	1

Abbildung 5.1: *Binäre Darstellung einer GAP-Lösung*

Eine naheliegende kompaktere Darstellung (Abbildung 5.2) ist ein Vektor $s \in I^n$ mit $I = \{1 \ldots m\}$. s_j $(i = 1 \ldots m)$ stellt dabei die Maschine dar, der Job j zugewiesen ist. Ein Vorteil dieser Repräsentation ist der Umstand, daß alle Zuweisungsbedingungen (2.3) implizit erfüllt sind, nachdem jeder Aufgabe genau eine Maschine zugeordnet ist. Damit werden die Standard Rekombinations- und Mutations-Operatoren angewendet ohne die Zuweisungsbedingungen zu verletzen. Weiters ist dieser Ansatz effizient, da die Länge des Chromosoms genau n ist im Gegensatz zu $m \cdot n$ bei der binären Darstellung.

job j	1	2	3	4	5	$\cdots$	$n-1$	n
$[j]$ = agent	2	1	3	m	3	$\cdots$	1	2

Abbildung 5.2: *Alternative Darstellung einer GAP-Lösung*

Jedoch garantiert auch diese Darstellung nicht, daß die Kapazitätsbedingungen (2.2) erfüllt sind. Dieser Fall kann durch Bestrafung in der Bewertungsfunktion oder durch einen Reparatur-Operator (Abschnitt 5.6) abgedeckt werden, der einzelne Aufgaben gezielt umordnet, sodaß die Kapazität nicht mehr überschritten wird. Dadurch kann

eine ungültige Lösung in eine gültige Lösung übergeführt werden. Die Wirksamkeit eines solchen Reparatur-Operators hängt stark von der Knappheit der Kapazitätsbeschränkungen ab.

5.2 Bewertung der Individuen

Chu und Beasley verwenden getrennte Fitness- und Untauglichkeits-Werte zur Bewertung eines Individuums. Die Fitnessfunktion und die Untauglichkeitsfunktion (*unfitness function*) wurden wie folgt festgelegt:

$$f(S) = \sum_{j=1}^{n} c_{S[j],j} \tag{5.1}$$

$$u(S) = \sum_{i=1}^{m} \max\left[0, \left(\sum_{j \in J, S[j]=i} r_{ij}\right) - b_i\right] \tag{5.2}$$

Der Ausdruck $S[j]$ steht für die Maschine die der Aufgabe j zugewiesen ist in der Lösung S. Die Gleichung (5.1) entspricht der ursprünglichen Zielfunktion (2.1). Die Untauglichkeitsfunktion (*unfitness*) in Gleichung (5.2) berechnet die Summe der Verletzungen der einzelnen Kapazitäts-Bedingungen (*capacity constraints*).

Wird eine gegebene Kapazitätsbeschränkung einer Maschine i überschritten (der akkumulierte Resourcenbedarf, der einer Maschine i zugeordneten Aufgaben, übersteigt somit die verfügbare Kapazität b_i der Maschine), so ist die Untauglichkeit die Differenz der Kapazität b_i und der akkumulierten Resourcen. Liegt der akkumulierte Resourcenbedarf innerhalb der Kapazitätsgrenze einer Maschine i, so hat die Untauglichkeit dieser Maschine i genau den Wert 0 (es liegt keine Verletzung der Kapazitätsbedingung vor). Die Untauglichkeitsfunktion $u(S)$ hat nur dann den Wert 0, wenn das Individuum S als ganzes gültig ist, d.h. es liegt keine einzige Kapazitätsverletzung vor.

5.3 Initiale Population

Die initiale Population wurde per gewöhnlicher Zufallsbelegung generiert.

Algorithmus 5.1 Initialisierung der ersten Generation $P(0)$

Let: $\begin{cases} N = \text{the population size (number of individuals),} \\ m = \text{the number of agents,} \\ n = \text{the number of jobs.} \end{cases}$

```
1: for i = 1 to N do
2:    for j = 1 to n do
3:       S_i[j] ← Random(1, m);                    /* random assignment */
4:    end for
5: end for
```

5.4 Selektion und Ersetzung

Als Selektion wurde die normale binäre Tournament Selektion gewählt: Sieger ist das Individuum mit dem besten (geringsten) Fitnesswert $f(S)$. Die Selektion ist somit ausschließlich fitnessbasierend und berücksichtigt in keinster Weise die Untauglichkeit (*constraint violation*) der Individuen.

Um ungültige Individuen so früh wie möglich zu eliminieren, wurde folgende Ersetzungsstrategie (*special ranking replacement strategy*) angewandt: Das Individuum mit der höchsten Untauglichkeit (höchsten Kapazitätsverletzung) $u(S)$ wird von dem neu generierten Kind als erstes ersetzt. Besteht die ganze Population nur mehr aus gültigen Individuen ($u(S_i) = 0, \forall S_i \in \mathcal{P}$), so wird das Individuum mit dem schlechtesten (höchsten) Fitnesswert $f(S_i)$ ersetzt. Duplikate (doppelte Individuen) werden nicht zugelassen – diese werden nicht in die Population aufgenommen.

5.5 Crossover und Mutation

Zur Rekombination wurde der One-Point Crossover Operator (Abbildung 4.2) gewählt ($O(n)$).

Als Mutation kam die sogenannte Exchange-Based Mutation (Abbildung 4.5) zur Anwendung. Hierbei werden die Aufgaben zweier zufällig gewählter Positionen ausgetauscht, z.B. $S[i] \rightleftharpoons S[j]$, wobei i und j aus Random$(1, n)$ stammen. Der Vorteil dieser Mutation liegt darin, daß die Wahrscheinlichkeit einer Verschlechterung des momentan Zustand (gültig oder ungültig) in Hinblick auf die Kapazitätsbeschränkung weniger wahrscheinlich ist als bei reiner Zufalls-Mutation.

5.6 Heuristic Improvement Operator

Um die Qualität der einzelnen Individuen zu erhöhen, führten Chu und Beasley einen heuristischen Reparatur-Operator ein. Dieser dient zum einen dazu, ungültige Individuen in gültige überzuführen (Reparatur) und zum anderen die Fitneß der Individuen zu verbessern (Optimierung). Dieser wissensbasierte Algorithmus wird von den Autoren ,,heuristic improvement operator" (Algorithmus 5.2) genannt und wird anschließend an die Operatoren Crossover und Mutation angewandt.

Der eigentliche Algorithmus teilt sich hierbei in zwei Teile: **Phase 1** dient dazu die Gültigkeit (untauglichkeits-basierend) einer Lösung zu verbessern. Zuerst werden alle ungültigen Beschränkungen (constraints) identifiziert ($O(m)$ Operationen). Anschließend wird auf jede verletzte Beschränkung eine Heuristik angewandt, die versucht, Aufgaben von einer Maschine zu einer anderen Maschine zu übertragen, sobald die Untauglichkeit (unfitness) reduziert wird. Im Detail bedeutet dies, daß alle zugewiesenen Aufgaben auf mögliche Umordnung untersucht werden ($O(n)$ Operationen); in Frage kommen hierbei die anderen verfügbaren Maschinen ($O(m)$ Operationen). Die Belegung auf eine neue Maschine kommt hierbei nur in Frage, wenn diese ausreichende Kapazität hat (*greedy heuristic*). Der (Worst-Case) Aufwand beträgt somit $O(n \cdot m^2)$ – dies stellt gleichzeitig die ,,teuerste" Komponente im GA

dar. **Phase 2** dient dazu die Kosten (fitness-basierend) der Lösung zu verbessern. Hierbei wird versucht Aufgaben anderen Maschinen zuzuordnen, sodaß die Kosten optimiert werden (z.B. für ein Minimierungsproblem gesenkt) unter Berücksichtigung der Kapazitätsbeschränkungen. Der (Worst-Case) Aufwand beträgt $O(n \cdot m)$.

Algorithmus 5.2 Heuristic Improvement Operator

Let: $\begin{cases} S[j] & = & \text{the agent assigned to job } j \text{ in } S, \\ R_i & = & \text{the accumulated resources assigned to agent } i \text{ in } S. \end{cases}$

1: compute $R_i \leftarrow \sum_{j \in J, S[j]=i} r_{ij}, \forall i \in I$;
2: /* Phase 1: improve feasibility (unfitness) */
3: **for** $i = 1$ to m **do**
4: **if** $R_i > b_i$ **then**
5: $T \leftarrow \{j | S[j] = i\}$;
6: **repeat**
7: randomly select a $j \in T$; $T \leftarrow T - \{j\}$;
8: search for an agent $i^* \in I$ for which $r_{i^*j} \leq b_{i^*} - R_{i^*}$;
9: **if** $i^* \neq nil$ **then**
10: $S[j] \leftarrow i^*$;
11: $R_i \leftarrow R_i - r_{ij}$; $R_{i^*} \leftarrow R_{i^*} + r_{i^*j}$;
12: **end if**
13: **until** $(T = \emptyset)$ or $(R_i \leq b_i, \forall i \in I)$;
14: **end if**
15: **end for**
16: /* Phase 2: improve cost (fitness) */
17: **for** $j = 1$ to n **do**
18: $i \leftarrow S[j]$;
19: search for $\min_{i^*}\{c_{i^*j},\ s.t.\ c_{i^*j} < c_{ij}$ and $r_{i^*j} \leq b_{i^*} - R_{i^*}, \forall i^* \in I\}$;
20: **if** $i^* \neq nil$ **then**
21: $S[j] \leftarrow i^*$;
22: $R_i \leftarrow R_i - r_{ij}$; $R_{i^*} \leftarrow R_{i^*} + r_{i^*j}$;
23: **end if**
24: **end for**

Abbildung 5.3 demonstriert die Funktionsweise des ,,heuristic improvement operator" anhand eines Beispiels. Die eingerahmten Einträge in der (r_{ij}, c_{ij}) Matrix kennzeichnen die Lösung vor Anwendung des ,,heuristic improvement operator". In der Phase 1 wird eine verletzte Bedingung in Reihe 2 entdeckt, woraufhin Aufgabe 5 von Maschine 2 nach Maschine 3 umgeordnet wird (ohne Verletzung der Kapazizätsbedingung in Reihe 3). Dies ergibt eine gültige Lösung. In Phase 2 wird Aufgabe 2 von Maschine 4 nach Maschine 1 umgereiht, da $c_{12} < c_{42}$ und der akkumulierte Resourcenbedarf $r_{12} + r_{14} \leq b_1$.

5.7 Ein Genetischer Algorithmus für das GAP

Der folgende Algorithmus für das GAP wurde übernommen aus der Arbeit von Chu und Beasley [5]. Die Duplikatprüfung in Zeile 11 hat einen (Worst-Case) Aufwand von $O(N \cdot n)$ für eine Generation. Nachdem der heuristische Reparatur-Operator von

Maschine i	Aufgabe j							
$r_{ij}(c_{ij})$	1	2	3	4	5	6		b_i
1	15 (8)	**11 (4)**	28 (1)	19 (6)	25 (7)	20 (6)	$\leq$	30
2	29 (5)	23 (9)	22 (5)	24 (8)	14 (3)	11 (9)	$\leq$	33 $\leftarrow$
3	23 (7)	16 (5)	28 (2)	30 (5)	**13 (5)**	15 (3)	$\leq$	29
4	20 (2)	14 (7)	25 (4)	22 (1)	17 (7)	30 (3)	$\leq$	35

	1	*2*	*3*	*4*	*5*	*6*	
$S[j]$	4	4	2	1	2	3	$f(S) = 26, u(S) = 3$
Phase 1					$\downarrow$		
	4	4	2	1	3	3	$f(S) = 28, u(S) = 0$
Phase 2		$\downarrow$					
	4	1	2	1	3	3	$f(S) = 25, u(S) = 0$

Abbildung 5.3: *Beispiel für den Heuristik Improvement Operator*

Chu und Beasley die ,,teuerste" Komponente im GA darstellt, beträgt die gesamte Worst-Case Zeit-Komplexität $O(m^2 \cdot n + m \cdot n) = O(m^2 \cdot n)$.

Algorithmus 5.3 Prinzip des GA für das GAP

Let: $\begin{cases} f(C) = \text{the fitness of the chromosome } C, \\ u(C) = \text{the unfitness of the chromosome } C, \\ P(t) = \text{the population at time (generation) } t. \end{cases}$

1: $t \leftarrow 0$; $t' \leftarrow 0$;
2: initialise $P(t) \leftarrow \{S_1, \ldots, S_N\}, S_i \in \{1, 2, \ldots, m\}^n$;
3: evaluate $P(t) : \{f(S_1), u(S_1)\}, \ldots, \{f(S_N), u(S_N)\}$;
4: find $\min_{S \in P(t), u(S)=0}\{f(S)\}$ or $\min_{S \in P(t), u(S)>0}\{u(S)\}$;
5: $S^* \leftarrow S$;
6: **while** $(t - t' < t_{max})$ **do**
7: select $\{P_1, P_2\} \leftarrow \Phi(P(t))$; /* Φ = tournament selection based on fitness */
8: crossover $C \leftarrow \Omega_f(P_1, P_2)$; /* Ω_f = one-point crossover operator */
9: mutate $C \leftarrow \Omega_m(C)$; /* Ω_m = exchange-based mutation operator */
10: $C \leftarrow \Omega_{improve}(C)$; /* $\Omega_{improve}$ = heuristic improvement operator */
11: **if** $C \equiv any\ S_i \in P(t)$ **then**
12: discard C; **continue**; /* C is redundant */
13: **end if**
14: evaluate $f(C)$ and $u(C)$;
15: find a $S' \in P(t)$ based on special ranking replacement strategy;
16: replace $S' \leftarrow C$;
17: **if** $(u(C) = 0$ and $u(S^*) = 0$ and $f(C) < f(S^*))$ or
$(u(S^*) > 0$ and $u(C) < u(S^*))$ **then**
18: $S^* \leftarrow C$;
19: $t' \leftarrow t$;
20: **end if**
21: $t \leftarrow t + 1$;
22: **end while**
23: **return** S^*;

Kapitel 6

Neue Lösungsansätze

Der in diesem Abschnitt präsentierte neue Lösungsansatz basiert auf dem im vorigen Kapitel beschriebenen GA (Algorithmus 5.3) von Chu und Beasley.

6.1 Allgemeines

Für die Repräsentation von Kandidatenlösungen (Abschnitt 5.1) wurde die gleiche kompakte Vektordarstellung wie von Chu und Beasley gewählt. In Abschnitt 6.2 wird eine neue Preprocessing-Strategie beschrieben, die dazu beitragen soll die Problemgröße zu verringern. Diese wird VRS (Variablen Reduktions Schema) genannt. Der erste Schritt im GA stellt die Initialisierung der Ausgangspopulation dar, diese wird in Abschnitt 6.3 näher beschrieben. Daran schließt sich die Evaluierung von Individuen an, für die unterschiedliche Varianten in Abschnitt 6.4 präsentiert werden. Neue Selektions- bzw. Ersetzungsstrategien werden in Abschnitt 6.6 und 6.7 beschrieben. Auf Rekombinations- und Mutationsoperatoren wird in Abschnitt 6.9 und 6.8 eingegangen. Danach wird die Reparaturroutine (*local improvement*) von Chu und Beasley wie in Abschnitt 5.6 beschrieben, angewendet. Die Duplikatprüfung wird analog zu Chu und Beasley angewandt. Die Schritte angefangen von der Selektion bis zur Ersetzung wiederholen sich so lange, bis die aktuelle Population dem Abbruchkriterium genüge tut.

Das beste Individuum ergibt sich aus der Bewertungsfunktion gemäß folgender Regel: ,,einmal gültig, immer gültig" (,,once valid, always valid"). Dieser Gültigkeitsmechanismus stellt sicher, daß eine einmal gültige, beste Kandidatenlösung auch immer gültig bleibt. Gibt es für die beste Kandidatenlösung mehrere Alternativen, d.h. es kommen mehrere Kandidatenlösungen mit gleicher Fitness in Frage, so wird das ökonomischte ausgewählt (sprich die Kandidatenlösung mit dem meisten Resourcen-Spielraum – größte Toleranz zwischen Resourcenverbrauch und Kapazitätsgrenze – wird ausgewählt).

6.2 Variablen Reduktions Schema VRS

In diesem Abschnitt wird das Variablen Reduktions Schema (VRS, *variable reduction scheme*) näher erläutert. Dieses stellt einen Vorverarbeitungsschritt vor dem eigentlichen GA dar und setzt direkt auf den Testdaten auf. Dieses stellt den Versuch dar, die Problemgröße zu verringern (und damit den Suchraum).

6.2.1 Beschreibung

Der Hintergedanke des VRS ist, daß durch die frühzeitige Reduzierung von freien Aufgaben (Variablen) der Suchraum verkleinert wird und damit die Performance des GA steigt ohne beträchtlichen Einfluß auf die Qualität der Endlösung zu haben. Daher werden bestimmte Variablen apriori fixiert und nehmen in der Optimierung durch den GA nicht teil – die Belegung dieser Variablen ändert sich nicht mehr.

6.2.2 Algorithmus

Der VRS-Algorithmus (Algorithmus 6.1) sucht für jeden Job $j = 1 \ldots n$ jene Maschine i_r mit geringstem Resourcenverbrauch r_{ij} und jene Maschine i_c mit niedrigsten Kosten c_{ij}. Falls die beiden ermittelten Maschinen (i_c und i_r) identisch und eindeutig sind, wird die Maschine i_r mit der Aufgabe j fix belegt bzw. umgekehrt. Die Belegung der Aufgabe j ist somit apriori bekannt und ändert sich während des gesamten GA nicht mehr.

Algorithmus 6.1 Prinzip des VRS-Algorithmus für eine Minimierungsaufgabe

Let: $\begin{cases} r_{ij} = \text{the resource consumption of the assignment from job } j \text{ to agent } i, \\ c_{ij} = \text{the cost (profit) of the assignment from job } j \text{ to agent } i, \\ S[j] = \text{the agent assigned to job } j \text{ in } S. \end{cases}$

1: **for** $j = 1$ to n **do**
2: find $\min_{i_r}\{r_{ij}, \forall i \in I\}$;
3: find $\min_{i_c}\{c_{ij}, \forall i \in I\}$; /* maximisation: find $\max_{i_p}\{p_{ij}, \forall i \in I\}$ */
4: **if** $i_r = i_c$ **then**
5: $S[j] \leftarrow i_r$; /* job j is assigned to agent i_r throughout the entire GA*/
6: **end if**
7: **end for**

6.2.3 Gegenbeispiel

Durch den Einsatz des VRS kann es selten, aber doch, zu suboptimalen bzw. ungültigen Lösungen kommen. Suboptimale oder ungültige Lösungen können auftreten, weil der VRS Ansatz nur eine lokale Betrachtungsweise der Kapazitätsbeschränkungen hat und die globalen Abhängigkeiten nicht berücksichtigt werden, d.h. es wird immer nur die Kapazität einer Maschine gleichzeitig betrachtet und nicht die ,,globalen" Abhängigkeiten zu den anderen Maschinen.

Ein Beispiel für eine suboptimale Lösung ist in Abbildung 6.1 gegeben, ein Beispiel für eine ungültige Lösung wird in Abbildung 6.2 gegeben. Bei den von uns durchgeführten Testläufen kam es zwar zu suboptimalen Lösungen, jedoch traten keine ungültigen Lösungen auf.

Aufgabe j

$r_{ij}(c_{ij})$	1	2	3		b_i
1	**1 (1)**	2 (8)	3 (2)	$\leq$	4
2	4 (3)	5 (2)	**2 (1)**	$\leq$	6

Suboptimale Lösung

	1	*2*	*3*	
$S[j]$	1	1	2	$c(S) = 10, r(S) = 5, u(S) = 0$

Optimale Lösung

	1	*2*	*3*	
$S[j]$	1	2	1	$c(S) = 5, r(S) = 9, u(S) = 0$

Abbildung 6.1: *Beispiel für suboptimale Anwendung des VRS*

Beim Durchlaufen des VRS (Abbildung 6.1) werden die Belegungen „[1/1]" und „[2/3]" ([agent/job]) ermittelt, d.h. $S[1] = 1$ und $S[3] = 2$. Diese Belegungen scheiden daher apriori aus und sind somit während des gesamten GA fixiert. Die damit in weiterer Folge erzielbare Lösung ist leider nur suboptimal, wie im Vergleich mit der optimalen Lösung darunter ersichtlich ist. Die Kombination mehrerer lokal, nicht optimaler Aufgaben kann im Gesamtergebnis zu einer optimalen Lösung führen.

Aufgabe j

$r_{ij}(c_{ij})$	1	2	3		b_i
1	**1 (1)**	4 (8)	2 (2)	$\leq$	3
2	4 (3)	5 (2)	**1 (1)**	$\leq$	5

Ungültige Lösungen

	1	*2*	*3*	
$S[j]$	1	1	2	$c(S) = 10, r(S) = 6, u(S) = 2$

	1	*2*	*3*	
$S[j]$	1	2	2	$c(S) = 4, r(S) = 7, u(S) = 1$

Gültige Lösung

	1	*2*	*3*	
$S[j]$	1	2	1	$c(S) = 5, r(S) = 8, u(S) = 0$

Abbildung 6.2: *Beispiel für ungültige Anwendung des VRS*

Weiters kann es dazu kommen, daß überhaupt keine gültige Lösung gefunden wird. Nach Durchlaufen des VRS (Abbildung 6.2) werden die Belegungen ,,[1/1]" und ,,[2/3]" fixiert. Wie leicht ersichtlich ist, steht für die Belegung der noch offenen Aufgabe (Nummer 2), allerdings keine Maschine mehr mit ausreichender Kapazität zur Verfügung. Die einzig gültige und somit auch optimale Lösung $S = [1|2|1]$ kann mittels des GA nicht mehr erreicht werden. In diesem Fall empfiehlt sich die Anwendung des GA nochmals ohne vorherige Anwendung des VRS-Algorithmus. Nach unseren Tests gehen wir aber davon aus, daß diese Situation sehr selten auftritt – bei den uns vorliegenden Testfällen ist dieser Fall kein einziges Mal aufgetreten.

Auch die Einschränkung, daß das globale Optimum dann mitunter nicht mehr gefunden werden kann, ist in der Praxis wie die Tests in Abschnitt 7.4 zeigen äußerst unwahrscheinlich.

6.3 Initialisierung der Ausgangspopulation

In diesem Abschnitt werden verschiedene alternative Initialisierungsroutinen erläutert, die dazu dienen die Ausgangspopulation (Generation 0) zu erzeugen. Zu den hier vorgestellten Routinen zählen, die Initialisierung basierend auf

- der Zufallsbelegung (*RND*),
- der Heuristik nach Martello und Toth (*MTH*) sowie
- der Constraint-Ratio-Heuristik (*CRH*).

Zunächst folgt eine detailierte Beschreibung der einzelnen Initialisierungsverfahren. Die erzielten Ergebnisse werden anschließend in Abschnitt 7.3.1 wiedergegeben.

6.3.1 Initialisierung durch Zufallsbelegung

Die Initialisierung nach der Zufallsbelegung funktioniert analog der Beschreibung in Abschnitt 5.3. Dabei wird jede Aufgabe genau einer Maschine durch reine Zufallsauswahl zugeteilt (bzw. umgekehrt). Der (Worst-Case) Aufwand des Algorithmus beträgt $O(N \cdot n)$.

6.3.2 Initialisierung mittels Heuristik von Martello und Toth

Diese Initialisierung greift auf die Heuristik von Martello und Toth (siehe Abschnitt 3.2.1) zurück. Die ursprüngliche Heuristik liefert genau eine einzige Lösung und nur für den Fall, daß diese gültig ist. Daher wurden folgende Ergänzungen eingebracht: Wird keine gültige Belegung gefunden, so wird jene mit der geringsten Kapazitätsverletzung ausgewählt, anderenfalls wird wie gewohnt nach den Gewichtsfaktoren vorgegangen. Damit soll erreicht werden, daß die Verletzung insgesamt reduziert wird.

Weiters wird der MTH-Algorithmus nicht auf das ganze Chromosom (Kandidatenlösung) angewandt – dies führt bekanntlich nur zu einer einzigen Lösung – sondern dieses wird in Teile geteilt, auf die das MTH-Verfahren separat angewandt wird. In einem iterativen Prozeß werden die noch freien Aufgaben in Gruppen bestimmter Größe geteilt und darauf die Reparaturstufe (Gültigkeit der Kapazitätsverletzungen herstellen) des MTH angewandt. Daran anschließend wird auf das gesamte Chromosom die Optimierungsstufe (Verbesserung des Zielfunktionswerts) angewandt. Der (Worst-Case) Aufwand des Algorithmus beträgt $O(N \cdot n^2 \cdot m)$.

6.3.3 Initialisierung mittels Constraint-Ratio-Heuristik

Bei dieser Initialisierung kommen mehrere Heuristiken zur Anwendung, nämlich die Constraint-Heuristik und die Ratio-Heuristik. Es werden deshalb mehrere Heuristiken eingesetzt, um die Vielfalt der Lösungen zu erhöhen, aber auch die Wahrscheinlichkeit, daß eine gültige Lösung gefunden wird. Die Kombination dieser beiden Routinen führt zu Lösungen, die besser sind als jene der einzelnen Routinen alleine.

Der generische Initialisierungsalgorithmus 6.2 kommt hierbei zur Anwendung. Dieser ruft abwechselnd die alternativen, heuristischen Initialisierungsoperatoren auf (dadurch wird keine der Routinen bevorzugt) und erstellt damit die einzelnen Kandidatenlösungen in der Population. Für den Fall, daß keine gültige Lösung gefunden wird, iteriert dieser Ansatz über alle möglichen Initialisierungsroutinen und nimmt davon das beste, ungültige Ergebnis. Der (Worst-Case) Aufwand des Algorithmus beträgt $O(N \cdot n \cdot m)$.

Algorithmus 6.2 Heuristikbasierter Initialisierungsalgorithmus

Let:
N = the number of chromosomes in the population P,
S_i = the i-th chromosome string S of the population P,
Ω_{init} = the init operators i.e. constraint and ratio heuristic.

```
 1: k ← 1;                                   /* loop counter for initializer methods */
 2: for i = 1 to N do
 3:    C ← null;
 4:    for l = 1 to |Ω_init| do
 5:       initialise C_Ω ← Ω_init[k];              /* call of initializer method */
 6:       if C = null or C.isBetter(C_Ω) then
 7:          C ← C_Ω;
 8:       end if
 9:       k ← mod(k, |Ω_init|) + 1;          /* loop based on modulo operator */
10:       if C.isValid() then
11:          break;
12:       end if
13:    end for
14:    S_i ← C;
15: end for
```

6.3.3.1 Constraint-Heuristik

Bei der Constraint-Heuristik handelt es sich um eine intelligentere Zufallsbelegung, bei der die Kapazitätsbeschränkung der einzelnen Maschinen berücksichtigt wird. In einem iterativen Prozeß wird für jede Aufgabe eine Maschine gesucht, bei der die Kapazitätsbeschränkung dieser Maschine erfüllt ist (Algorithmus 6.3). Bei der Zuweisung wird die Maschine ausgewählt, die die Kapazitätsbedingung als Erste erfüllt. Es wird keine lokale Suche nach der am Besten passenden Maschinen-Aufgaben-Zuweisung durchgeführt, weil sonst immer nur die gleiche Belegung des Chromosomes generiert werden würde. Die Suche nach einer gültigen Maschine startet mit einer zufällig gewählten Maschine, um bestimmte Maschinen nicht zu bevorzugen und um die Streuung der enthaltenen Lösungen zu erhöhen.

Algorithmus 6.3 Constraint Initializer

Let: $\begin{cases} S[j] & = & \text{the agent assigned to job } j \text{ in } S, \\ R_i & = & \text{the accumulated resources assigned to agent } i \text{ in } S. \end{cases}$

1: $\sum_{i \in I} R_i \leftarrow 0$; $F \leftarrow \{1, \ldots, n\}$;
2: **while** $F \neq \emptyset$ **do**
3: $j \leftarrow \text{Random}(F)$; /* take a random job */
4: search for an agent i^* which fulfills $R_i + r_{ij} \leq b_i, \forall i \in I$;
5: **if** $i^* = nil$ **then**
6: $i^* \leftarrow \text{Random}(1,m)$;
7: **end if**
8: $S[j] \leftarrow i^*$;
9: $R_{i^*} \leftarrow R_{i^*} + r_{i^*j}$;
10: $F \leftarrow F - \{j\}$;
11: **end while**

In Abbildung 6.3 ist ein Beispiel für die Initialisierung gemäß Constraint-Heuristik angegeben. In diesem wird die verbliebene Restkapazität (*available capacity*) einer Maschine i mit dem Ausdruck a_i bezeichnet (für eine gültige Maschine gilt: $a_i \geq 0$). Die Angaben für die Indizes i und j stellen Zufallszahlen dar und stellen sicher, daß einzelne Maschinen bzw. Aufgaben nicht bevorzugt werden. Hat die Maschine i, die momentan an der Reihe ist, keine freien Kapazitäten für die Aufnahme der Aufgabe j mehr frei, so werden iterativ die anderen Maschinen nach freien Kapazitäten abgesucht. Dies ist im Beispiel angedeutet durch den Pfeil nach rechts ($\rightarrow$).

Problemmatrix:

Maschine i	Aufgabe j							
$r_{ij}(c_{ij})$	1	2	3	4	5	6		b_i
1	15 (8)	11 (4)	28 (1)	19 (6)	25 (7)	20 (6)	$\leq$	30
2	29 (5)	23 (9)	22 (5)	24 (8)	14 (3)	11 (9)	$\leq$	33
3	23 (7)	16 (5)	28 (2)	30 (5)	13 (5)	15 (3)	$\leq$	29
4	20 (2)	14 (7)	25 (4)	22 (1)	17 (7)	30 (3)	$\leq$	35

Initialisierungsvorgang:

Schritt 1: $j = 2, i = 3$

	1	*2*	*3*	*4*	*5*	*6*	
$S[j]$	/	**3**	/	/	/	/	$a_3 = 13$

Schritt 2: $j = 4, i = 1$

	1	*2*	*3*	*4*	*5*	*6*	
$S[j]$	/	3	/	**1**	/	/	$a_1 = 11$

Schritt 3: $j = 3, i = 1 \rightarrow 2$

	1	*2*	*3*	*4*	*5*	*6*	
$S[j]$	/	3	**2**	1	/	/	$a_2 = 11$

Schritt 4: $j = 1, i = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$

	1	*2*	*3*	*4*	*5*	*6*	
$S[j]$	**4**	3	2	1	/	/	$a_4 = 15$

Schritt 5: $j = 5, i = 2 \rightarrow 3$

	1	*2*	*3*	*4*	*5*	*6*	
$S[j]$	4	3	2	1	**3**	/	$a_3 = 0$

Schritt 6: $j = 6, i = 4 \rightarrow 1 \rightarrow 2$

	1	*2*	*3*	*4*	*5*	*6*	
$S[j]$	4	3	2	1	3	**2**	$a_2 = 0$

$$f(S) = 32, u(S) = 0$$

Abbildung 6.3: *Beispiel für Constraint-Initialisierung*

6.3.3.2 Ratio-Heuristik

Die Ratio-Heuristik berücksichtigt den relativen Kosten-Resourcen-Index (*relative cost-resource index*) cri_{ij} der Zuweisung der Aufgabe j auf die Maschine i und stellt einen kosten-basierenden Ansatz dar. Gesucht wird jene Maschine, die für eine gegebene Aufgabe das beste ,,Kosten-Nutzen-Verhältnis" aufweist, unter Berücksichtigung der Kapazität der jeweiligen Maschine. Wird keine Maschine mit erfüllbarer Kapazitätsbeschränkung gefunden, so wird eine Zufallszuweisung durchgeführt. Der Index cri_{ij} wird bestimmt durch den relativen Resourcenverbrauch $r^*_{ij} = \frac{r_{ij}}{b_i}$.
Der Kosten-Resourcen-Index berechnet sich daraus zu:

$$cri_{ij} = c_{ij} \cdot r^*_{ij} \qquad \text{(Maximierung: } cri_{ij} = \frac{p_{ij}}{r^*_{ij}}\text{)}$$

Grundsätzlich gilt, daß Belegungen bevorzugt werden, die wenige Resourcen bei gleichzeitig hoher Kapazität der involvierten Maschine verbrauchen. Bei Minimierung werden daher Belegungen bevorzugt, die niedrige Kosten mit sich bringen, bei gleichzeitig niedrigem relativen Resourcenverbrauch der jeweiligen Maschine. Bei Maximierung wird analog dazu darauf geachtet, daß der Profit möglichst hoch ausfällt.

Algorithmus 6.4 Ratio Initializer

Let: $\begin{cases} S[j] & = & \text{the agent assigned to job } j \text{ in } S, \\ R_i & = & \text{the accumulated resources assigned to agent } i \text{ in } S. \end{cases}$

1: $\sum_{i \in I} R_i \leftarrow 0$; $F \leftarrow \{1, \ldots, n\}$;
2: **while** $F \neq \emptyset$ **do**
3: $j \leftarrow \text{Random}(F)$; /* take a random job */
4: search for an agent i^* which has the best cost-resource index cri_{ij} and fulfills $R_i + r_{ij} \leq b_i, \forall i \in I$;
5: **if** $i^* = nil$ **then**
6: $i^* \leftarrow \text{Random}(1,m)$;
7: **end if**
8: $S[j] \leftarrow i^*$;
9: $R_{i^*} \leftarrow R_{i^*} + r_{i^*j}$;
10: $F \leftarrow F - \{j\}$;
11: **end while**

In Abbildung 6.4 ist ein Beispiel für die Initialisierung mittels Ratio-Heuristik angegeben. Es gilt das Gleiche, wie für das vorherige Beispiel. In der Problemmatrix ist, zusätzlich zum Resourcenverbrauch bzw. den Kosten, der Rang des Kosten-Resourcen-Index angegeben (Reihung nach cri_{ij}). Die Reihenfolge, in der die Aufgaben j verteilt werden sollen, wurde analog zum Beispiel für die Constraint-Initialisierung gewählt. Die jeweilige Maschine i ergibt sich dabei aus dem Rang (Reihenfolge) des Kosten-Resourcen-Index cri_{ij}.

Problemmatrix:

Maschine i	Aufgabe j							
$r_{ij}(c_{ij})$	1	2	3	4	5	6		b_i
1	15 (8) [2.]	11 (4) [1.]	28 (1) [1.]	19 (6) [2.]	25 (7) [4.]	20 (6) [4.]	$\leq$	30
2	29 (5) [3.]	23 (9) [4.]	22 (5) [4.]	24 (8) [4.]	14 (3) [1.]	11 (9) [3.]	$\leq$	33
3	23 (7) [4.]	16 (5) [2.]	28 (2) [2.]	30 (5) [3.]	13 (5) [2.]	15 (3) [1.]	$\leq$	29
4	20 (2) [1.]	14 (7) [3.]	25 (4) [3.]	22 (1) [1.]	17 (7) [3.]	30 (3) [2.]	$\leq$	35

Initialisierungsvorgang:

Schritt 1: $j = 2 \rightarrow i = 1$

	1	*2*	*3*	*4*	*5*	*6*	
$S[j]$	/	**1**	/	/	/	/	$a_1 = 19$

Schritt 2: $j = 4 \rightarrow i = 4$

	1	*2*	*3*	*4*	*5*	*6*	
$S[j]$	/	1	/	**4**	/	/	$a_4 = 13$

Schritt 3: $j = 3 \rightarrow i = 1 \rightarrow 3$

	1	*2*	*3*	*4*	*5*	*6*	
$S[j]$	/	1	**3**	4	/	/	$a_3 = 1$

Schritt 4: $j = 1 \rightarrow i = 4 \rightarrow 1$

	1	*2*	*3*	*4*	*5*	*6*	
$S[j]$	**1**	1	3	4	/	/	$a_1 = 4$

Schritt 5: $j = 5 \rightarrow i = 2$

	1	*2*	*3*	*4*	*5*	*6*	
$S[j]$	1	1	3	4	**2**	/	$a_2 = 19$

Schritt 6: $j = 6 \rightarrow i = 3 \rightarrow 4 \rightarrow 2$

	1	*2*	*3*	*4*	*5*	*6*	
$S[j]$	1	1	3	4	2	**2**	$a_2 = 8$

$$f(S) = 27 \quad u(S) = 0$$

Abbildung 6.4: *Beispiel für Ratio-Initialisierung*

6.4 Evaluierung der Individuen

Die Fitnessfunktion $f(S)$ dient zur Evaluation der Individuen und ist bei der Selektion von entscheidender Rolle (Abschnitt 4.6). Die Fitnessfunktion wird, wie in Abschnitt 4.5 dargestellt, mittels Skalierung aus der Bewertungsfunktion $g(S)$ berechnet.

Folgende Bewertungsfunktionen werden betrachtet:

- **Fitness Evaluation**
  Diese kommt bei Chu und Beasley zur Anwendung und berücksichtigt ausschließlich die Fitness eines Individuums, d.h. die Kapazitätsverletzungen werden bei der Selektion überhaupt nicht beachtet, sondern nur bei der Ersetzung.

- **Lack Evaluation**
  Die *lack evaluation* (Mangel) berücksichtigt bei der Bewertung eines ungültigen Individuums ausschließlich den Mangel (den Überhang an Resourcenverbrauch gegenüber der vorhandenen Kapazität) ohne Berücksichtigung der zugrunde liegenden Fitness des Individuums. Liegt ein gültiges Individuum (ohne Kapazitätsverletzung) vor, so wird nach der Fitness des Individuums verfahren.

- **Condition Evaluation**
  Die *condition evaluation* (Kondition, Beschaffenheit) berücksichtigt immer beide Aspekte eines Individuums, nämlich die Fitness (den Zielfunktionswert) als auch die Untauglichkeit.

6.4.1 Kennzahlen

Der Zielfunktionswert $h(S)$ der Optimierungsaufgabe errechnet sich zu:

$$h(S) = \sum_{j=1}^{n} c_{S[j],j}$$

Der maximale Zielfunktionswert H_{max} stellt eine obere Schranke dar und errechnet sich hierbei wie folgt:

$$H_{max} = \left(\sum_{j=1}^{n} \max_{i \in I} \{c_{ij}\} \right) \tag{6.1}$$

Anders ausgedrückt, werden für jede Aufgabe j die Maschine i, die am teuersten kommt, ausgewählt und die so entstandenen Kosten c_{ij} aufsummiert (ohne Rücksicht auf Kapazitätsgrenzen). Der so gewonnene Wert, H_{max}, dient dazu die Kandidatenlösungen in zwei disjunkte Mengen von gültigen und ungültigen Individuen zu trennen.

Der folgende Ausdruck gibt die relative Kapazitätsüberschreitung (*relative overconsumption*) einer Kandidatenlösung an:

$$oc(S) = \frac{\sum_{i=1}^{m} \max \left[0, \frac{\left(\sum_{j \in J, S[j]=i} r_{ij} \right) - b_i}{b_i} \right]}{m} \tag{6.2}$$

Diese dient dazu, ungültige Lösungen mit einem Abschlag, entsprechend der Verletzung der Bedingungen (*constraint violation*), zu versehen. Sie ist genau null für gültige Lösungen (somit keine Überschreitung) ansonsten größer null (d.h. mehr als die 100% Gesamtkapazität aller Maschinen ist in Verwendung).

In dem Ausdruck (6.2) kommt die relative Kapazitätsüberschreitung einer Maschine zur Anwendung – diese hilft die Auswirkung einer konkreten Aufgaben-Maschinen Belegung über alle Maschinen hinweg vergleichbar zu machen (durch Normierung mit der Kapazität einer Maschine b_i). Weiters hat der relative Abschlag den großen Vorteil, daß dieser direkt in die Bewertung einer Kandidatenlösung einfließen kann. Die unterschiedlichen Einheiten Kosten/Resourcen spielen keine Rolle – es ist somit keine Skalierung notwendig.

6.5 Bewertungsfunktion

6.5.1 Fitness Evaluation

Bei der Fitness Evaluation wird die direkte Skalierung $f(S) = g(S)$ angewandt, wobei in diesem Fall $g(S)$ dem Zielfunktionswert $h(S)$ entspricht – der Penaltyfunktionswert wird zur Bewertung nicht herangezogen.

6.5.2 Lack Evaluation

Bei der Lack Evaluation wird ebenfalls nicht zwischen der Bewertungsfunktion und der Fitnessfunktion unterschieden. Sie berücksichtigt bei ungültigen Individuen ausschließlich die Untauglichkeit.

$$g(S) = \begin{cases} h(S) & oc(S) = 0, \\ (H_{max} + 1) \cdot (1 + oc(S)) & oc(S) > 0. \end{cases}$$

Somit ist bei einer Minimierungsaufgabe sichergestellt, daß alle gültigen Belegungen innerhalb dieser Schranke zu liegen kommen, während sich alle ungültigen Belegungen oberhalb dieser Schranke befinden – die beiden Bereiche (durch die Schranke H_{max} voneinander getrennt) sind somit disjunkt.

6.5.3 Condition Evaluation

Bei diesem Ansatz dürfen sich gültige und ungültige Individuen "vermischen". Die Fitness- bzw. Bewertungsfunktion ergibt sich dabei zu:

$$g(S) = h(S) \cdot (1 + oc(S)) \qquad oc(S) \geq 0.$$

Die Idee dahinter ist, daß gültige Individuen mit ungültigen in Konkurrenz stehen – der Funktionswert $oc(S)$ fungiert hierbei als (relative) Entfernung zu einer gültigen Lösung. Das Ziel der Condition Evaluation ist, ein ausgewogeneres Bild der Wertigkeit eines Individuums abzugeben.

6.6 Selektionsstrategie

Zur Selektion dient die binäre Tournament Selektion, d.h. bei jedem Durchgang („Turnier") werden zwei Individuen zufällig aus der Population ausgewählt und jenes mit dem besseren Fitnesswert ist das Selektierte. Der (Worst-Case) Aufwand mit $O(2)$ ist somit konstant.

Der Ansatz von Chu und Beasley ist ausschließlich fitnessorientiert, die Untauglichkeit (unfitness) wird nicht berücksichtigt. Laut Chu und Beasley besteht die Population sehr bald aus fast ausschließlich gültigen Individuen, daher konzentriert sich die Selektion ausschließlich auf Individuen mit hoher Fitness (siehe Fitness Evaluation, Abschnitt 6.4 bzw. Abschnitt 6.5).

Bei unserem Ansatz wird der Schwerpunkt auf möglichst gültige Individuen gelegt, daher wird für die Selektion neben der Fitness auch die Untauglichkeit herangezogen (Standard-Implementierung der EA-Bibliothek von Raidl [22]). Gerade zu Beginn des GA ist es von großer Bedeutung möglichst rasch zu gültigen Individuen in der Population zu kommen (siehe Lack- und Condition-Evaluation, Abschnitt 6.4 bzw. Abschnitt 6.5).

Für die Lack-Evaluation spricht, daß sie ohne spezieller Vorkehrungen auskommt, nachdem die ungültigen Individuen immer schlechter bewertet werden als das schlechteste gültige. Bei der Condition-Evaluation ist zu beachten, daß sich ohne geeignete Gegenmaßnahmen, ungültige Individuen in der Population durchsetzen können (Individuen mit geringer Verletzung der Kapazität, aber gutem Fitnesswert). Eine dieser Gegenmaßnahmen ist die Anwendung von *SAW-ing*.

Unter SAW-ing wird die Adaptierung der Fitnessfunktion zur Behandlung von Problemen mit Nebenbedingungen (constraint problem) verstanden. Dies funktioniert hierbei wie in Algorithmus 6.5 angegeben. Der Algorithmus wurde der Arbeit 'SAW-ing EAs' von Eiben und Hemert [10] entnommen.

Algorithmus 6.5 Prinzip des SAW-ing Algorithmus

Let: $\begin{cases} w = \text{initial weight}, \\ T_p = \text{the update period}, \\ \Delta_w = \text{the level of weight increase}. \end{cases}$

```
1: set intial weights (thus fitness function f);
2: while not termination do
3:    for the next T_p fitness evaluations do
4:       let the GA go with this f;
5:    end for
6:    redefine f and recalculate fitness of individuals;
7: end while
```

Den Nebenbedingungen werden Gewichtsfaktoren (w) zugeordnet – diese Gewichtsfaktoren fungieren als Abschlag für verletzte Beschränkungen. Diese werden in bestimmten Intervallen (T_p) adaptiv geändert, daher der Name *Stepwise Adaption of Weights* (SAW). Solange die beste Kandidatenlösung ungültig ist, werden die Gewichtsfaktoren der verletzten Nebenbedingungen erhöht (Δ_w). Anschließend muß die Bewertung aller Individuen in der Population unter den nun gegebenen Umständen

(neue Fitnessfunktion) erneuert werden. Auf diesem Wege werden die ungültigen Individuen sukzessive aus der Population gedrängt. Durch diese adaptive Fitnessfunktion gleicht die Fitness einer Sägezahn-Kurve.

6.7 Ersetzungsstrategie

Das Ziel der Ersetzungsstrategie (Abschnitt 4.9.3, 5.4) ist es, ungültige bzw. schlechte Individuen so rasch wie möglich zu eliminieren.

Zunächst wird das Individuum mit der höchsten Untauglichkeit durch ein Nachkommen ersetzt. Besteht die Population zur Gänze aus gültigen Individuen, so wird jenes mit der geringsten Fitness ersetzt. Es werden also gültige gegenüber fiteren Individuen favorisiert – damit wird die Gültigkeit der Population vorangetrieben. Der (Worst-Case) Aufwand beträgt hierbei $O(N)$.

6.8 Rekombination

In dieser Arbeit wurden die Auswirkungen unterschiedlicher Rekombinationsoperatoren untersucht. Aufgrund der alternativen Darstellung der Individuen verletzen diese Operatoren die Zuweisungsbedingungen (Nebenbedingung 2.3) nicht. Für eine genaue Erklärung der Rekombinations Operatoren siehe Abschnitt 4.7. Die erzielten Ergebnisse decken sich mit den Erkenntnissen von Chu und Baisley [5]; diese sind in Abschnitt 7.7 ersichtlich.

6.9 Mutation

In diesem Abschnitt werden die verschiedenen Mutationsroutinen erläutert, die dazu dienen, neue Informationen in die Population einzubringen. In dieser Arbeit werden verglichen:

- der Zufallsbelegung einer Maschine (*flip mutation*),
- dem Austausch der Aufgaben zweier Maschinen (*swap mutation, exchange-based mutation*) sowie
- der Heuristik von Martello und Toth (MTH).

Allen drei Mutationsarten ist gemeinsam, daß sie das Auftreten von ungültigen Lösungen nicht verhindern. Der (Worst-Case) Aufwand ist für die Zufalls-Mutation als auch die Swap-Mutation konstant ($O(c)$); für die Mutation basierend auf MTH gilt $O(c) + O(c^2 \cdot m) = O(c^2 \cdot m)$. Zunächst folgt eine detailierte Beschreibung der einzelnen Mutationsverfahren. Dabei gilt jeweils:

$$\begin{aligned} n_{mut} &= \text{die Anzahl der Mutationen,} \\ S[j] &= \text{die Maschine } i \text{ die der Aufgabe } j \text{ in } S \text{ zugeordnet ist,} \\ m &= \text{die Anzahl der Maschinen (agents),} \\ n &= \text{die Anzahl der Aufgaben (jobs).} \end{aligned}$$

6.9.1 Mutation durch Zufallsbelegung

Die Mutation durch Zufallsbelegung, auch *Flip Mutation* genannt (Abschnitt 4.8), funktioniert durch einfache Zufallsbelegung einer Aufgabe zu einer Maschine (die aktuelle Zuweisung der Aufgabe zu einer Maschine wird aufgelöst). Diese stellt die einfachste Art der Mutation dar und berücksichtigt die Nebenbedingungen (Kapazitätsbeschränkungen) in keiner Weise (Algorithmus 6.6).

Algorithmus 6.6 Flip Mutation

1: **for** $k = 1$ to n_{mut} **do**
2: $S[\text{Random}(1, n)] \leftarrow \text{Random}(1, m)$; /* random assignment */
3: **end for**

6.9.2 Mutation durch Austausch

Bei der Mutation durch Austausch, siehe Abschnitt 4.8 - *Swap Mutation*, werden die Aufgaben zweier Maschinen vertauscht. Hierbei werden zwar auch nicht die Kapazitätsbeschränkungen der einzelnen Maschinen berücksichtigt, aber die Wahrscheinlichkeit und die Auswirkungen einer Kapazitätsverletzung sind geringer als beim reinen Zufallsansatz (Algorithmus 6.7).

Algorithmus 6.7 Swap Mutation

1: **for** $k = 1$ to n_{mut} **do**
2: $S[\text{Random}(1, n)] \rightleftharpoons S[\text{Random}(1, n)]$; /* swap assignments */
3: **end for**

6.9.3 Mutation durch Heuristik von Martello und Toth

Durch die Mutation nach der Heuristik von Martello und Toth (MTH) werden die Kapazitätsbeschränkungen explizit berücksichtigt. Allerdings garantiert dieser Ansatz dennoch nicht, immer eine gültige Lösung zu finden. Da der Originalansatz von Martello und Toth möglicherweise keine Lösung liefert, wurde er dahingehend adaptiert, daß in so einem Fall die Lösung mit der geringsten Kapazitätsverletzung geliefert wird. Für eine genauere Erklärung dieses Verfahrens siehe Abschnitt 6.3.2 bzw. Abschnitt 3.2.1.

Bei der Mutation nach MTH werden zunächst die Aufgaben festgelegt, die im anschließenden Schritt durch die Heuristik von Martello und Toth neu vergeben werden (Algorithmus 6.8). Dort wird versucht, aufgrund der Heuristik, den optimalen Kandidaten (Maschine) für diese Aufgabe zu finden.

Algorithmus 6.8 MTH Mutation

1: $F \leftarrow \emptyset$;
2: **for** $k = 1$ to n_{mut} **do**
3: $F \leftarrow F + \{\text{Random}(1, n)\}$; /* collect jobs for mutation */
4: **end for**
5: $S \leftarrow \Omega_{MTH}(S, F)$; /* Ω_{MTH} = heuristic improvement operator */

6.10 GA basierend auf LP-Lösung

Der bisherige Ansatz wurde H3-GA getauft, aufgrund der Anwendung der drei Heuristiken für Initialisierung, Mutation und Reparatur. Basierend auf dem H3-GA Ansatz wurde versucht die Lösungen dadurch weiter zu verbessern, daß für die Initialisierung der Ausgangspopulation die Lösung der LP-Relaxation des GAP herangezogen wird. Diese wurde mit Hilfe des Programmpakets CPLEX [7] ermittelt – CPLEX ist ein Programm zum Lösen von linearen Optimierungsaufgaben (LP); siehe Anhang 10.5. CPLEX kann auch dazu verwendet werden, die ganzzahlige Lösung (IP) eines Optimierungsproblems zu liefern. Die so erhaltenen Ergebnisse dienen gleichzeitig als Maßstab für unseren GA-basierenden Ansatz – siehe Ergebnisse in Kapitel 7.

Der GA basierend auf der LP-Lösung funktioniert analog unserem zuvor besprochenen Ansatz, mit dem einzigen Unterschied, daß als Ausgangsbasis für die Initiallösungen (Generation 0) die LP-Lösung des GAP herangezogen wird. Die LP-Initialisierung gliedert sich hierbei in zwei Teile:

- Zunächst wird die vorhandene LP-Lösung (von CPLEX) in eine integrale Lösung umgewandelt (Basislösung).
- Aufbauend auf der so gewonnenen Basislösung werden so viele Initiallösungen wie nötig kreiert.

6.10.1 Erzeugen der Basislösung

Der Schwerpunkt dieses Algorithmus liegt im Bestimmen welche Teilbelegungen (*fraktionale Belegungen*) der LP-Lösung in Exklusivbelegungen (*integrale Belegungen*) umgewandelt werden. Folgende Vorbemerkungen dazu.

Die integralen (100-prozentigen) Belegungen werden von unserem Algorithmus nicht verändert, weil davon ausgegangen wird, daß die meisten Variablen in der optimalen, ganzzahligen Lösung den gleichen Wert haben werden wie in der LP-Lösung. Dies wird aber kaum auf alle zutreffen. Theoretisch und praktisch gibt es jedoch den Fall, daß integrale Belegungen in der LP-Lösung in der optimalen, ganzzahligen Lösung nicht vorkommen, weil sie anderen fraktionalen Belegungen ,,Platz" machen mußten; also aufgrund der Konstellation der Nebenbedingungen zueinander. Diese Aussage wurde durch unsere Tests erhärtet. Es kann nicht garantiert werden, daß die Teilbelegungen einer konkreten Aufgabe immer in eine gültige, integrale Belegung der involvierten Maschinen übergeführt werden können. Dies tritt genau dann auf, wenn die Teilbelegungen einer Aufgabe bei keiner der beteiligten Maschinen zu

einer gültigen, integralen Belegung führen (weil in jedem Fall die Kapazitätsgrenze überschritten sind). Dieser Situation sollte der GA aufgrund seines explorativen Charakters Rechnung tragen.

Eine weitere Tatsache ist, daß fraktionale Aufgaben immer mindestens zwei oder mehr Maschinen zugeordnet sind. Unsere empirischen Tests ergaben bei der Belegung der Maschinen folgende ,,Rangordnung": zumeist war eine Maschine klar dominant, sodaß diese eindeutig mehr Anteil an der Belegung hatte als die restlichen Maschinen.

Wie soll nun eine fraktionale Lösung in eine integrale übergeführt werden?

Die Antwort vor dem Hintergrund des bisher gesagten läßt folgende Möglichkeiten zu: durch Zufallsbelegung oder mittels Heuristik.

Unter dem Aspekt, daß nicht garantiert werden kann, daß überhaupt eine gültige Belegung einer fraktionalen Aufgabe existiert, ist die Zufallsbelegung eine valide Option. Außerdem sollte diese für mehr Diversität sorgen. Die Heuristik sollte andererseits sicherstellen, daß die ganzzahlige Lösung der Natur der LP-Lösung am nächsten kommt. Deshalb haben wir uns für letztere Variante entschieden.

Die gewichtigste Rolle spielt die Zuweisung mit der höchsten Belegung, daher wird diese zur Generierung der ganzzahligen Initiallösung herangezogen. Falls sich diese Entscheidung als nicht optimal herausstellen sollte, so wurde es dem GA überlassen solche Konflikte aufzulösen. Der Algorithmus zum Erzeugen einer Basislösung wird nur einmal aufgerufen – siehe Algorithmus 6.9.

Algorithmus 6.9 LP Initialisierung - Basislösung

Let: $\begin{cases} (i,j,r)_{lp} &= \text{assignment of job } j \text{ to agent } i \text{ with ratio } r \text{ in lp-solution,} \\ A_{lp} &= \text{all assignments } (i,j,r)_{lp} \text{ of lp-solution.} \end{cases}$

1: remove all fractional assignments (i.e. ratio $r > 0$ and ratio $r < 1$) of every (fractional) job except for its assignment with highest ratio r;

2: make all remaining fractional assignments of A_{lp} integral, i.e. ratio $r \leftarrow 1$;

Zuerst werden alle Teilbelegungen einer fraktionalen Aufgabe gelöscht, bis auf jene Teilbelegung mit dem höchsten Anteil der Zuteilung zu einer Maschine. Jede fraktionale Aufgabe ist nun genau einer Maschine zugeordnet – jener mit dem höchsten Belegungsanteil. Es wird davon ausgegangen, daß die Maschinen mit einem höheren Belegungsanteil einen gewichtigeren Anteil bei der Lösung spielen.

Anschließend werden alle fraktionalen Belegungen in eine integrale Belegung umgewandelt, d.h. die Maschine übernimmt die Abarbeitung dieser Aufgabe zur Gänze. Dies wird meistens zu einer Überschreitung der Resourcen führen.

Daher wird in einem anschließenden Schritt (siehe nächster Abschnitt) versucht die Gültigkeit dieser integralen, aber ungültigen Lösung wieder herzustellen.

6.10.2 Herstellen gültiger Kandidatenlösungen

Dieser Schritt dient dazu, um aus einer ungültigen Basislösung (beliebig viele) gültige Kandidatenlösungen zu generieren. Es kann aber nicht garantiert werden, daß in jedem Fall eine solche gefunden wird (best-try effort). Für den Fall, daß die Basislösung selber bereits gültig ist, wird es einem anschließenden Optimierungsschritt in Form des Heuristic Improvement Operators von Chu und Beasley (siehe Abschnitt 5.6) überlassen, verschiedene Lösungen zu generieren.

Das Hauptkriterium unseres Algorithmus ist die Reihenfolge in der die verletzten Bedingungen aufgelöst werden. Folgende Varianten sind hierbei denkbar:

- fraktionale Belegungen zuerst (*fractional first*)
- alle Belegungen gleich behandeln (*no respecter of assignments*)

Unsere empirischen Tests ergaben, daß es sinnvoll ist, alle Belegungen der überkapazitiven Maschinen gleich zu behandeln. Die Vorteile sind, daß die Lösungen diversifizierter sind, weiters führt dieser Ansatz öfter zu gültigen Lösungen. Auf der anderen Seite führt dieser Ansatz nicht immer zu den (besten) initialen Lösungen.

Bei der Variante fractional-first werden zuerst die fraktionalen Belegungen betrachtet, dann erst die restlichen, ganzzahligen Belegungen einer überkapazitiven Maschine. Eine Spezialisierung dieser Variante ist, sich zuallererst auf jene fraktionalen Aufgaben zu konzentrieren, die die Kapazität einer Maschine überschreiten. Die Vorteile dieses Ansatzes sind teilweise bessere Lösungen als bei der Gleichbehandlung der Belegungen, auf der anderen Seite sind nicht immer gültige Lösungen erreichbar, da die Nebenbedingungen zu strikt (*tight*) angenähert werden (Erschwernis für Reparatur). Weiters war die Anzahl der eindeutigen Lösungen bei diesem Ansatz geringer.

Weiters kann der Algorithmus unterschieden werden, nach dem Verhalten ,,Ganz oder gar nicht" (*complete*) versus ,,teilweise Verbesserung" (*partial*). Der Complete-Ansatz funktioniert nach dem Prinzip *all-or-nothing*, d.h. eine Belegung wird nur dann einer anderen Maschine zugeordnet, wenn diese die Aufgabe zur Gänze aufnehmen kann (innerhalb der verfügbaren Kapazität). Beim Partial-Ansatz wird im Prinzip nach einem Tauschpartner gesucht, der die Kapazitätsverletzung möglicherweise nicht restlos behebt, aber eine Verbesserung hin zu einer gültigen Lösung bedeutet (die Kapazitätsüberschreitung sinkt insgesamt). Der Complete-Ansatz hat gegenüber dem anderen Ansatz den Vorteil, daß die Lösungsstruktur eher erhalten bleibt – beim Partial-Ansatz kommt es zu wesentlich mehr Vertauschungen und damit ist dieser wesentlich destruktiver.

Das bisher Gesagte führt uns zu einem Algorithmus, der alle Belegungen einer ungültigen Maschine gleich behandelt und nach dem Complete-Ansatz funktioniert. Dies entspricht exakt dem Reparatur-Algorithmus von Chu und Beasley (Phase 1). Algorithmus 6.10 beinhaltet daher die Reparatur-Phase des Heuristic Improvement Operators von Chu und Beasley.

Dieser Teil der LP-Initialisierung wird für jedes Chromosome extra aufgerufen. Zuerst wird die aktuelle Kandidatenlösung mit der Basislösung belegt, um anschließend in eine gültige Lösung übergeführt zu werden (a'la Chu und Beasley).

Algorithmus 6.10 LP Initialisierung - Reparatur

Let: $\begin{cases} S[j] & = & \text{the agent assigned to job } j \text{ in } S, \\ R_i & = & \text{the accumulated resources assigned to agent } i \text{ in } S. \end{cases}$

1: /* init chromosome based on lp-solution */
2: **for** $j = 1$ to n **do**
3: get assignment $a \leftarrow (i, j, r)_{lp}$ of A_{lp} at position j;
4: $S[j_a] \leftarrow i_a$;
5: **end for**
6: compute $R_i \leftarrow \sum_{j \in J, S[j]=i} r_{ij}, \forall i \in I$;
7: /* repair chromosome (improve feasibility) */
8: **for** $i = 1$ to m **do**
9: **if** $R_i > b_i$ **then**
10: $T \leftarrow \{j | S[j] = i\}$;
11: **repeat**
12: randomly select a $j \in T$; $T \leftarrow T - \{j\}$;
13: search for an agent $i^* \in I$ for which $(R_{i^*} + r_{i^*j}) \leq b_{i^*}$;
14: **if** $i^* \neq nil$ **then**
15: $S[j] \leftarrow i^*$;
16: $R_i \leftarrow R_i - r_{ij}$; $R_{i^*} \leftarrow R_{i^*} + r_{i^*j}$;
17: **end if**
18: **until** $(T = \emptyset)$ or $(R_i \leq b_i, \forall i \in I)$;
19: **end if**
20: **end for**

6.10.3 Ein Beispiel für die LP-Initialisierung

In Abbildung 6.5 ist ein Beispiel für die LP-Initialisierung angegeben. Hierbei ist zunächst die Problemmatrix angegeben, die das eigentliche GAP beschreibt; anschließend ist das LP- und das IP-Optimum für dieses Problem angegeben.

Dabei ist eindeutig ersichtlich, daß:

- die integralen Belegungen aus der LP-Lösung auch in der IP-Lösung vorkommen können (Aufgaben vier bis sechs),
- die ,,dominante" Belegung einer fraktionalen Belegung in der LP-Lösung erhalten bleiben kann (Aufgabe drei) und
- die Belegungen (fraktional oder integral) der LP-Lösung nicht unbedingt in der IP-Lösung vorkommen müssen (Aufgaben eins bis zwei).

Die Basislösung wird nun wie folgt generiert. Ausgehend aus der LP-Lösung werden alle integralen Belegungen eins-zu-eins übernommen, bei den fraktionalen Belegungen wird jeweils die dominanteste zu 100% übernommen. Dies führt uns zu der im Beispiel angegeben (ungültigen) Basislösung – diese fungiert als Ausgangsbasis für alle weiteren Kandidatenlösungen. Die Basislösung ist in der Problemmatrix abgebildet – zu erkennen an dem Rahmen um die jeweilige Belegung.

Im Rahmen der Reparaturphase der LP-Initialisierung wird nun die Basislösung schrittweise in eine (möglichst gültige) Kandidatenlösung übergeführt. Dabei werden

in einem iterativen Prozeß alle überschrittenen Maschinen betrachtet und versucht alternative, gültige Belegungen zu finden. Im ersten Schritt wandert Aufgabe zwei von Maschine eins nach Maschine zwei. Dadurch verbessert sich die Untauglichkeit der Lösung, gleichzeitig geht damit eine Verschlechterung des Zielfunktionswerts einher. Im nächsten Schritt wird Maschine drei mit Aufgabe eins betraut (vormals Maschine vier). In diesem Beispiel führt uns dies zu einer gültigen Kandidatenlösung – dies muß keinesfalls immer der Fall sein. Durch weitere Optimierungen, wie z.B. durch Anwendung des Heuristischen Reparaturoperators von Chu und Beasley, besteht die Möglichkeit, daß diese Kandidatenlösung noch weiter verbessert werden kann. In unserem konkreten Fall liegt bereits die optimale Lösung vor.

Problemmatrix:

Maschine i	Aufgabe j							
$r_{ij}(c_{ij})$	1	2	3	4	5	6		b_i
1	15 (8)	11 (4)	28 (1)	19 (6)	25 (7)	20 (6)	$\leq$	30 $\leftarrow$
2	29 (5)	**19 (9)**	22 (5)	24 (8)	14 (3)	11 (9)	$\leq$	33
3	**14 (7)**	16 (5)	28 (2)	30 (5)	13 (5)	15 (3)	$\leq$	29
4	20 (2)	14 (7)	25 (4)	22 (1)	17 (7)	30 (3)	$\leq$	35 $\leftarrow$

Optima:

	1	*2*	*3*	*4*	*5*	*6*
$S_{LP}[j]$	2: 35% 4: 65%	1: 100%	1: 68% 3: 32%	4: 100%	2: 100%	3: 100%

$f(S_{LP}) = 15.4$, $u(S_{LP}) = 0$

	1	*2*	*3*	*4*	*5*	*6*
$S_{IP}[j]$	3: 100%	2: 100%	1: 100%	4: 100%	2: 100%	3: 100%

$f(S_{IP}) = 24$, $u(S_{IP}) = 0$

Basislösung:

		↓				
	1	*2*	*3*	*4*	*5*	*6*
$S[j]$	4: 100%	1: 100%	1: 100%	4: 100%	2: 100%	3: 100%

$f(S) = 14$, $u(S) = 16$

Kandidatenlösung:

	↓					
	1	*2*	*3*	*4*	*5*	*6*
$S[j]$	4: 100%	2: 100%	1: 100%	4: 100%	2: 100%	3: 100%

$f(S) = 19$, $u(S) = 7$

	1	*2*	*3*	*4*	*5*	*6*
$S[j]$	3: 100%	2: 100%	1: 100%	4: 100%	2: 100%	3: 100%

$f(S) = 24$, $u(S) = 0$

Abbildung 6.5: *Beispiel für die LP-Initialisierung*

Kapitel 7

Experimente und Ergebnisse

In diesem Kapitel liefern wir eine Beschreibung der durchgeführten Experimente zur Evaluierung unseres Lösungsansatzes sowie der erhaltenen Ergebnisse und daraus resultierender Erkenntnisse.

Zunächst wird der Aufbau der einzelnen Testinstanzen beschrieben (Abschnitt 7.3). Daran anschließend folgen die Ergebnisse und Schlussfolgerungen der jeweiligen Experimente. Für den Vergleich der erhaltenen Lösungen wird der Lösungsansatz von Chu und Beasley herangezogen (nachdem dieser als Ausgangspunkt unserer Arbeit diente) als auch die Lösungen des kommerziellen Programmpakets CPLEX (siehe Anhang 10.5). Mit diesem wurden die LP-Optima sowie IP-Lösungen der einzelnen GAP-Instanzen ermittelt. Die Lösungen von Chu und Beasley dienen uns als Referenzlösungen, nachdem diese gleichzeitig die derzeit besten, bekannten Lösungen darstellen. Die Arbeit von M. Yagiura, T. Ibaraki und F. Glover [29] ist parallel zu unseren eigenen Arbeit entstanden – auf diese sind wir erst Anfang 2003 aufmerksam geworden – daher konnten die Erkenntnisse dieser Arbeit in unsere eigene nicht mehr einbezogen werden.

Eine vollständige Implementierungsbeschreibung wird im nächsten Kapitel (Kapitel 8) geliefert.

7.1 GA Parameter

Die Populationsgröße wurde analog zu der Referenzstrategie (von Chu und Beasley) mit 100 Individuen pro Population gewählt. Wurde die Populationsgröße verändert, d.h. halbiert oder verdoppelt, so konnten die gleichen Effekte wie bei Chu und Beasley beobachtet werden: Eine Halbierung der Populationsgröße von $N = 100$ auf $N = 50$ verschlechterte die Qualität der Lösungen merklich, während eine Verdoppelung der Populationsgröße auf $N = 200$ keine signifikante Verbesserung der Lösungen brachte, bei gleichzeitig, erheblich gestiegener Laufzeit.

Das Abbruchkriterium (*termination criteria*, Abschnitt 4.11) wurde wie folgt realisiert. Jeder Testlauf wird nach $t_{max} = 500000$ eindeutigen Nachkommen abgebrochen (doppelte Individuen in der jeweils aktuellen Population werden nicht gezählt), die keine Verbesserung der besten Lösung bringen. Durch dieses Konvergenzkriterium wird sichergestellt, daß die Qualität der Lösungen über eine weite Reihe von

verschiedenen Instanzen konsistenter ist. Dieser Ansatz ist aufgrund der variablen Anzahl von Generationen problem-unabhängiger, im Gegensatz zu einer fix vorgegebenen Anzahl von Generationen. So fällt die gesamte Laufzeit für einfache Probleme geringer aus (d.h. die beste Lösung ist rasch gefunden), während sie für komplexe Probleme länger ausfällt (diese brauchen mehr Generationen um zu konvergieren). Generell interessieren uns hier in erster Linie möglichst gute Lösungen und nur sekundär die Laufzeit des Algorithmus. Bei unserem GA, basierend auf der Bibliothek EALib [22], wird jede Generation mitgezählt, d.h. auch jene Individuen die bereits in der Population enthalten sind.

Um möglichst akkurate Ergebnisse zu erhalten, wurden jeweils zehn Testläufe des GA für jeden Testfall durchgeführt und die Resultate gemittelt. In weiterer Folge wird der Ansatz basierend auf den drei Heuristiken für Initialisierung, Mutation und Reparatur ,,H3-GA" genannt und der Ansatz, der auf der LP-Lösung des GAP aufbaut ,,LP-GA".

7.2 CPLEX

Um die Qualität unseres Lösungsansatzes zu bewerten, berechnen wir die relative Abweichung zum tatsächlichen Optimalwert (IP-Optimum), falls dieses bekannt ist, oder zum Lösungswert der LP-Relaxation, der immer eine untere Schranke darstellt. Diese relative Abweichung bezeichnen wir als *gap*.

Das LP- bzw. IP-Optimum wurde hierbei mit Hilfe des kommerziellen Programmpakets *CPLEX 8.0* [7] ermittelt. CPLEX ist ein Programm zum Lösen von linearen Optimierungsaufgaben – für nähere Informationen siehe Abschnitt 10.5. Nachdem CPLEX für die vorhandenen Testinstanzen sehr gute Ergebnisse lieferte (unter teilweise massiven Speicherverbrauch), wurden von uns zusätzliche Probleminstanzen generiert (siehe Abschnitt 7.3).

7.3 Testdaten

Zum Testen wurden zwei verschiedene Kategorien von Testdaten verwendet. Die aus der Literatur stammenden Testfälle reichen von 5 Maschinen/15 Aufgaben bis 20 Maschinen/200 Aufgaben und wurden in der Vergangenheit schon von einigen Autoren verwendet (siehe [1], [3], [19], [24]). Diese Testdaten sind unter [2], der OR-Library, zugänglich. Nachdem diese Testfälle für heutige Verhältnisse zu einfach/zu klein sind, da sie mit CPLEX in relativ kurzer Zeit optimal gelöst werden können, wurden neue Testinstanzen kreiert – diese reichen bis 80 Maschinen zu 400 Aufgaben.

Die erste Kategorie von Testdaten sind 60 ,,kleinere" Maximierungsaufgaben und wurden *gap1* bis *gap12* getauft. Sie wurden u.a. in [3] und [19] verwendet. Die optimalen Lösungen dieser Testfälle sind bekannt (siehe [3]). Diese haben folgende Charakteristika:

- Die Anzahl der Maschinen beträgt 5, 8 und 10. Die Anzahl der Aufgaben ergibt sich aus dem Verhältnis ρ von Aufgaben zu Maschinen ($\rho = \frac{n}{m}$) und beträgt 3, 4, 5 und 6.
- Der Resourcenbedarf r_{ij} sind Ganzzahlen aus dem Wertebereich Random$(5, 25)$, der Kosten-Koeffizient c_{ij} sind Ganzzahlen aus dem Wertebereich Random$(15, 25)$ und die Kapazität der Maschinen ist $b_i = 0.8 \sum_{j \in J} \frac{r_{ij}}{m}$.

Beim Testen kamen die insgesamt zwölf Problemklassen zur Anwendung, mit jeweils fünf unterschiedlichen Ausprägungen pro Problemklasse – somit wurde summa summarum mit 60 Testinstanzen getestet. Pro Instanz wurden jeweils zehn Testdurchläufe durchgeführt.

Die zweite Kategorie dient zum Minimieren und besteht aus 24 ,,großen" Problemen – *gapA* bis *gapD* genannt – diese wurden u.a. in [1], [24] und [25] verwendet. Für jede Problemart wird ein Testfall pro Maschinen/Aufgaben Kombination gebildet ($m = 5, 10, 20$ und $n = 100, 200$). Um größere Anforderungen an den Lösungsalgorithmus zu stellen, wurden zusätzlich 39 neue Testklassen generiert. Diese gliedern sich auf in, die Erweiterung der Klasse gapD (Instanzen mit jeweils 40 und 80 Maschinen sowie Instanzen mit 400 Aufgaben) sowie die Testklassen *gapE* und *gapF* (Maschinenanzahl: 5, 10, 20, 40 und 80 zu Aufgaben mit 100, 200 und 400).

Typ A, B und C haben einfache Annahmen bezüglich der Kosten/Resource Beziehung. Typ B und C sind schwieriger als Typ A - Probleme, weil die Kapazitätsbeschränkungen enger sind (die ,,kleinen" Probleme gap1 bis gap12 sind vom Typ C). Typ D - Probleme haben eine höhere Korrelation zwischen der Kosten/Resource Beziehung bei engen Kapazitätsbeschränkungen und stellen die schwierigste Problemkategorie dar. Die Charakteristik dieser Testklasse kann wie folgt beschrieben werden: hoher Resourcenbedarf wird niedrigen Kosten gegenübergestellt und vize versa, d.h. es gilt einen Mittelweg zu finden zwischen geringen Kosten und hohem Resourcenverbrauch. Typ E und F - Probleme entsprechen in ihrer Struktur Typ D mit dem Unterschied, daß die Kapazität der Maschinen variiert wurde (Typ E verringert bzw. Typ F angehoben). Für die ,,großen" Testfälle sind die optimalen Lösungen für die Kategorie A bis C bekannt, ab D nur mehr teilweise.

Typ A: r_{ij} sind positive, ganze Zahlen aus dem Intervall $[5, 25]$; c_{ij} sind positive, ganze Zahlen aus dem Intervall $[10, 50]$;

$$b_i = 9 \left(\frac{n}{m}\right) + 0.4 \max_{i \in I} \sum_{j \in J, I_j = i} r_{ij},$$

mit $I_j = \min[i | c_{ij} \leq c_{kj}, \forall k \in I]$.

Typ B: r_{ij} und c_{ij} sind analog Typ A; b_i wird auf 70% des Wertes von Typ A gesetzt.

Typ C: r_{ij} und c_{ij} sind analog Typ A;

$$b_i = \left(\frac{0.8}{m}\right) \sum_{j \in J} r_{ij}.$$

Typ D: r_{ij} sind positive, ganze Zahlen aus dem Intervall $[1, 100]$;

$$c_{ij} = 111 - r_{ij} + e,$$

wobei gilt, e sind ganze Zahlen aus dem Intervall $[-10, 10]$;

$$b_i = \left(\frac{0.8}{m}\right) \sum_{j \in J} r_{ij}.$$

Typ E: r_{ij} und c_{ij} sind analog Typ D; b_i wird auf 70% des Wertes von Typ D gesetzt.

Typ F: r_{ij} und c_{ij} sind analog Typ D; b_i wird auf 200% des Wertes von Typ D gesetzt.

7.3.1 Aufbau der Instanzen

Das Dateiformat der GAP-Instanzen ist wie folgt aufgebaut:

- die Anzahl der Maschinen (m), die Anzahl der Aufgaben (n)
- für jede Maschine i $(i = 1, ..., m)$:

  die Kosten der Zuweisung von Aufgabe j auf Maschine i $(j = 1, ..., n)$
- für jede Maschine i $(i = 1, ..., m)$:

  der Resourcenbedarf der Zuweisung von Aufgabe j auf Maschine i $(j = 1, ..., n)$
- die Resourcenkapazität der Maschine i $(i = 1, ..., m)$

7.4 Variablen Reduktions Schema VRS

Das Variablen-Reduktions-Schema VRS konnte für die Testklassen 01 bis 12 bzw. A bis C angewandt werden – für die Testklasse D ergab sich keine Reduktion aufgrund der Charakteristik dieser Testklasse (die Korrelation von Kosten und Resourcen ist am höchsten, weiters sind die Kapazitätsbeschränkungen sehr strikt). Für die Testklassen 01 bis 12 erhielten wir keine hinreichend aussagekräftigen Ergebnisse, nachdem diese sehr einfach sind. Im weiteren werden daher nur die Testklassen A bis C betrachtet.

Die Ergebnisse der empirischen Tests waren recht uneinheitlich – teilweise lieferte der Ansatz unter Einbeziehung des VRS bessere Ergebnisse (jeweils bezogen auf den Durchschnitt einer Testinstanz) als der Ansatz ohne, aber auch das umgekehrte Verhalten war festzustellen. Dies ist möglicherweise in dem gewählten, naiven Reduktionsschema begründet, das mittels lokaler Suche nur die einfachsten Abhängigkeiten kristallisiert und die globalen Zusammenhänge völlig außer Acht läßt. In Tabelle 7.1 sind die Ergebnisse im Vergleich mit und ohne Anwendung des VRS ersichtlich. (Für die Ergebnisse wurde auf die Evaluierung mittels Condition Evaluation zurückgegriffen – dieses hat für die Wertigkeit der Ergebnisse keine Bedeutung.) In der Tabelle ist die Relation VRS-Ansatz zum Standard-Ansatz in Prozent ersichtlich, weiters ist die relative Abweichung zum ganzzahligen Optimum angegeben sowie die relative Variablenreduktion.

Problem	Größe		IP	% Ratio		STD	VRS	
Instanz	m	n	Optimum	Gen	Zeit	% gap_{avg}	% gap_{avg}	% Δ
A	5	100	1698	45.5	74.0	0.00	0.12	17.0
		200	3235	349.5	70.4	0.00	0.00	17.0
	10	100	1360	1212.9	131.6	0.00	0.00	5.0
		200	2623	27.1	78.9	0.00	0.00	6.0
	20	100	1158	71.5	86.4	0.00	0.00	7.0
		200	2339	19.4	94.5	0.00	0.00	2.5
B	5	100	1843	82.6	63.8	0.05	0.00	27.0
		200	3552	89.3	77.6	0.12	0.14	18.0
	10	100	1407	214.0	130.4	0.00	0.00	16.0
		200	2827	104.5	97.9	0.26	0.41	9.5
	20	100	1166	377.3	204.4	0.16	0.07	7.0
		200	2339	96.8	91.0	0.12	0.15	8.5
C	5	100	1931	86.4	76.4	0.15	0.20	15.0
		200	3456	126.6	105.5	0.10	0.09	21.0
	10	100	1402	77.5	77.0	0.11	0.19	7.0
		200	2806	75.6	71.7	0.33	0.37	10.5
	20	100	1243	262.2	215.0	0.27	0.25	5.0
		200	2391	129.5	125.8	0.53	0.50	4.0

Tabelle 7.1: *Vergleich des GA mit und ohne VRS*

Die durchschnittliche Variablenreduktion über die Testklassen (A-C; bei der Klasse D war keine Reduktion möglich) betrug $11,3\%$ (die Variablenreduktion rangiert zwischen 2,5% und 27%). Für die leichten Testklassen (01 bis 12) war dieser Wert etwas höher, mit durchschnittlich 14,6%. Von insgesamt 18 Fällen war die durchschnittliche Generationsanzahl bei Anwendung von VRS in zehn Fällen besser und in acht Fällen schlechter, trotzdem waren die Ergebnisse beim zeitlichen Aufwand in zwei von drei Fällen besser; aber insgesamt betrachtet war die Laufzeit sogar um ca. vier Prozent schlechter als ohne Verwendung des VRS-Ansatzes (gesamtdurchschnittlich gesehen gibt es somit keine Zeitersparnis). Dies dürfte damit begründet sein, daß VRS nur triviale Abhängigkeiten berücksichtigt, die vom GA ohnehin rasch fixiert werden. In Punkto auf die Qualität der Endlösungen waren die Ergebnisse ziemlich ebenbürtig (fünfmal besser, sechsmal gleich und siebenmal schlechter). Die durchschnittliche Abweichung über alle drei Testklassen beträgt ohne VRS $0,12\%$ im Vergleich zu 0,14% beim Ansatz mit VRS.

Weiters war es nicht möglich eine Schlußfolgerung aus der durchschnittlichen Maschinenbelegung (Aufgaben-Maschinen-Relation) oder der relativen Variablenreduktion herzuleiten. Obwohl suboptimale Lösungen auftraten (siehe Testklasse A), kam es bei unseren empirischen Tests zu keinen ungültigen Lösungen. Aufgrund der durchwachsenen Ergebnisse und dem Bestreben möglichst optimale Lösungen zu finden, wurde in weiterer Folge auf das Variablen-Reduktions-Schema verzichtet.

7.5 Initialisierung des GA

Um die Initialisierungsroutinen möglichst gut beurteilen zu können, wurde eine Analyse im Rahmen der Testklassen A und D durchgeführt. Die Klasse A stellt hierbei die einfachste Testinstanz dar: es sollte für den Initialisierungsalgorithmus ein leichtes sein gültige Kandidatenlösungen zu generieren. Die Klasse D widerum stellt die komplexeste Testklasse dar und dient dazu die Qualität der Initialisierungsroutinen abzugrenzen.

Bei unseren empirischen Tests zeigte sich, daß die Initialisierung basierend auf der Heuristik von Martello und Toth meist zu ungültigen Lösungen führte (Testklasse A bis D). Nicht einmal für die einfachste Testklasse (A) wurden in allen Fällen gültige Kandidatenlösungen generiert ebenso wurden für die Testklasse D kaum noch gültige Lösungen generiert. Zwar lieferte der MTH-Ansatz die besten Lösungen (sofern überhaupt gültige Lösungen vorlagen), jedoch ist dies im Optimierungsschritt begründet, den MTH inkorporiert. Andererseits war dadurch die Diversität der Kandidatenlösungen mit Abstand am geringsten (viele Duplikate).

Nachdem der Ansatz von Chu und Beasley allein durch das Vorhandensein gültiger Kandidatenlösungen in der initialen Population verbessert werden konnte, wurde in weiterer Folge auf die Initialisierung nach MTH verzichtet.

In der Tabelle 7.2 werden die Zufallsbelegung und der CRH-Ansatz miteinander verglichen. Dabei dient bei der relativen Bewertung jeweils die Zufallsbelegung (RND) als Referenzlösung (100%). Diese Normierung hilft die unterschiedlichen Probleminstanzen einer Problemklasse zu vergleichen. Die Daten wurden jeweils über zehn

Testläufe pro Testinstanz ermittelt. Beim ausschließlichen Vorhandensein ungültiger Lösungen sind in den Tabellen keine Werte angegeben, diese Stellen sind durch das Zeichen '–' gekennzeichnet. In der Tabelle sind folgende Werte angegeben:

- Initiale Lösung
  - die Anzahl der gültigen Individuen in der Population in Prozent und
  - die mittlere, relative Abweichung vom LP-Optimum (% gap_{avg})
- Endlösung
  - die absolute und relative Anzahl der Generationen (bis zur Lösung)

	Größe			Initial		Lösung	
LP-Optimum	m	n	Art	% gültig	% gap_{avg}	Gen	% Gen
6345.4	5	100	RND	0.0	–	927445	100.0
			CRH	53.6	10.1	956531	103.1
12736.2	5	200	RND	0.0	–	1626017	100.0
			CRH	82.5	8.8	1738041	106.9
6323.4	10	100	RND	18.4	14.1	920968	100.0
			CRH	100.0	12.9	1231151	133.7
12418.3	10	200	RND	0.5	–	1659379	100.0
			CRH	100.0	12.5	2193522	132.2
6142.5	20	100	RND	90.2	15.4	1466699	100.0
			CRH	100.0	14.6	1464001	99.8
12217.7	20	200	RND	36.7	15.1	1766882	100.0
			CRH	100.0	13.6	2925268	165.6
Durchschnitt			RND	24.3	14.9	–	100.0
			CRH	89.4	12.1	–	123.6

Legende	
RND:	Initialisierung nach dem Zufallsprinzip (Referenz)
CRH:	Initialisierung mittels Constraint-Ratio-Heuristik
– :	nicht anwendbar

Tabelle 7.2: *Vergleich unterschiedlicher Initialisierungsroutinen – Instanz D*

Wie in Tabelle 7.2 ersichtlich ist, liefert der CRH-Ansatz, mit Abstand, die meisten gültigen Kandidatenlösungen in der Ausgangspopulation. Die mittlere Abweichung vom LP-Optimum ist ebenfalls geringer als bei der Zufallsbelegung, d.h. die Qualität der initialen Lösungen ist besser. (Wird für die Durchschnittsbildung der relativen Abweichung nur die Instanzen herangezogen bei denen der Zufallsansatz zu ebenfalls gültigen Individuen führt, so liegt der CRH-Ansatz immer noch mit 13,7% vorne). Die Anzahl der Generationen bis zur Endlösung liegt beim CRH-Ansatz höher bei gleichzeitig besseren Ergebnissen – ein weiteres Indiz dafür, daß die Qualität der Initiallösungen von entscheidender Bedeutung ist, damit der GA nicht vorzeitig konvergiert. Im übrigen lieferten sowohl die Zufallsbelegung als auch unser Ansatz eindeutige Kandidatenlösungen in der initialen Population. Die Streuung der Endlösungen über mehrere Testläufe einer Testinstanz war beim CRH-Ansatz geringer, d.h. die gelieferten Ergebnisse sind somit stabiler.

7.5.1 Erkenntnisse

Für die Funktionsweise des GA ist von entscheidender Bedeutung, daß möglichst früh nur mehr gültige Lösungen vorhanden sind (analog der Beobachtung von Chu und Beasley). Der GA soll sich auf die Optimierung des Zielfunktionswerts konzentrieren und nicht auf die Herstellung der Gültigkeit. Chu und Beasley gehen davon aus, daß der GA aufgrund der Selektion und Ersetzung früher oder später (nur mehr) aus gültigen Kandidatenlösungen besteht. Durch die Anwendung des CRH-Ansatzes liegen in den meisten Fällen bereits von Anfang an gültige Lösungen vor (aufgrund der beiden unterschiedlichen Heuristiken, die die Kapazitätsgrenzen berücksichtigen). Durch das Vorhandensein gültiger Lösungen in der Ausgangspopulation erhöht sich dadurch die Wahrscheinlichkeit in den daran anschließenden Verarbeitungsschritten des GA widerum gültige Lösungen zu erhalten. Der CRH-Ansatz bietet somit das Potential bessere Endlösungen im GA zu finden.

Weiters wurde die initiale Population vor dem eigentlichen Ablauf des GA zur Gänze optimiert – mittels des Heuristik Improvement Operators von Chu und Beasley (kommt ebenfalls bei dem Kind-Chromosom zur Anwendung). Dadurch wurde nicht nur die Anzahl der gültigen Kandidatenlösungen erhöht, sondern auch die Qualität in Hinblick auf das Optimierungsziel (Zielfunktionswert).

In Abbildung 7.1 wird der GA mit unterschiedlichen Intialisierungsarten gegenübergestellt (Problemklasse $D-20\cdot200$). Im allgemeinen lieferte der CRH-Ansatz bessere Endlösungen, bei gleich guten Ergebnissen benötigte der CRH-Ansatz im allgemeinen weniger Generationen.

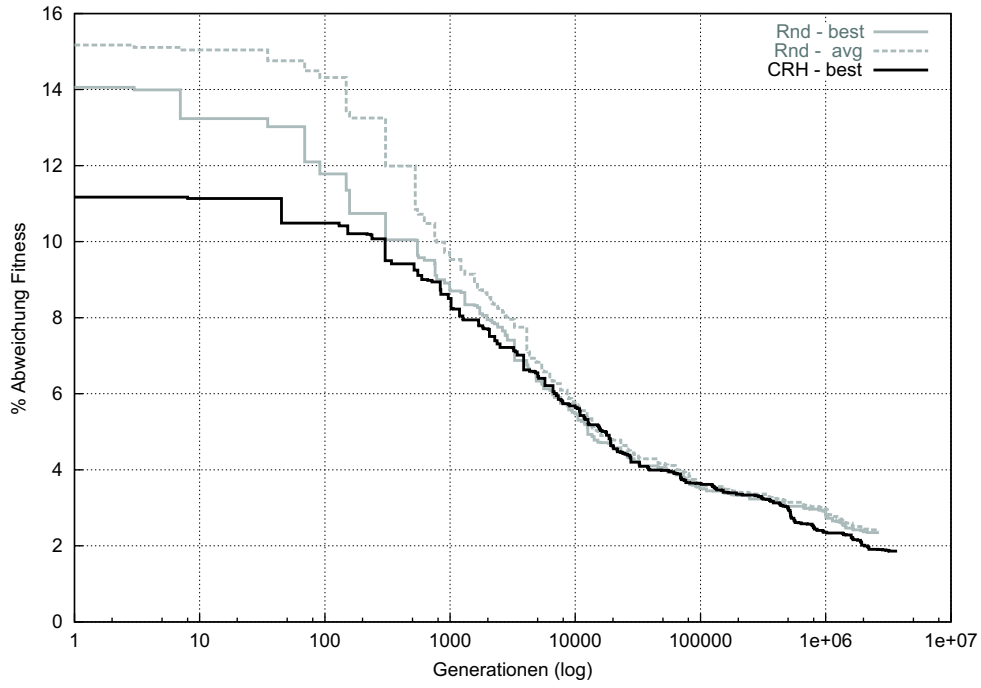

Abbildung 7.1: *Vergleich unterschiedlicher Initialisierungsarten – Instanz D*

7.6 Selektions- und Ersetzungsstrategie

Die Selektions- und Ersetzungsstrategie basieren auf der Hypothese, daß ,,gültige Individuen stärker in der Lage sind gültige bzw. fast gültige Nachkommen zu erzeugen". Daher ist es sinnvoll sich zunächst auf gültige Nachkommen zu konzentrieren und dann erst auf die Qualität der Nachkommen (Fitness). Die Population soll so früh wie möglich aus gültigen Individuen bestehen, das Hauptaugenmerk des GA soll im weiteren Verlauf auf dem Verbessern der Fitness (Qualität) liegen. Diese Annahme wurde durch unsere empirischen Tests bestätigt. Daher ist unser Ansatz mit der Bewertung, die neben der Fitness auch die Untauglichkeit berücksichtigt – sei es in Form der Lack-Evaluation oder des Condition-Evaluation dem Ansatz, der nur rein die Fitness betrachtet (wie bei Chu und Beasley), überlegen. In Abbildung 7.2 ist ein Vergleich der beiden Evaluierungsarten (Fitness-Evaluation und Unfitness- bzw. Lack-Evaluation) gegeben.

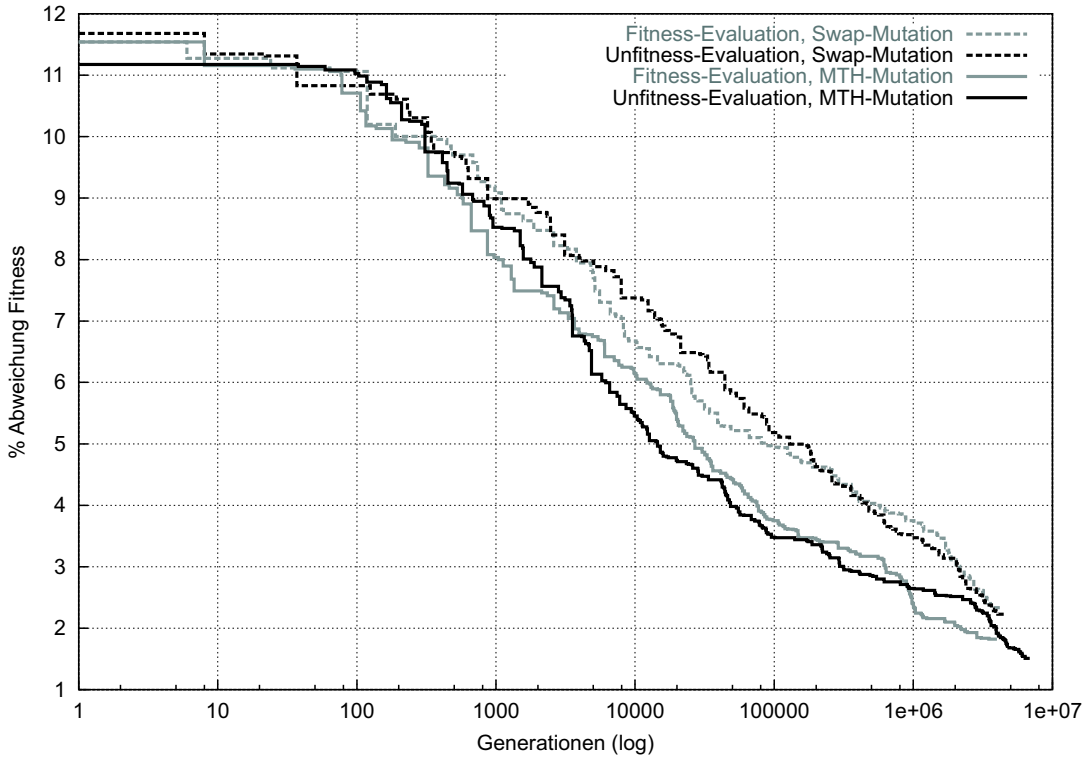

Abbildung 7.2: *Vergleich unterschiedlicher GA-Ansätze (Evaluierung, Mutation)*

Vergleichen wir die Condition-Evaluation mit der Lack-Evaluation, so schließt die Condition-Evaluation ohne dem SAW-ing Mechanismus eindeutig schlechter ab (siehe Tabelle 7.3). Die Anzahl der Generationen war zwar im Gesamtdurchschnitt kürzer als bei der Lack-Evaluation, aber nur weil der genetische Algorihmus früher konvergiert ist. Ohne SAW-ing kann es vorkommen, daß die gültigen Individuen schrittweise aus der Population gedrängt werden, weil ihr Zielfunktionswert schlechter ist als der von ungültigen. Nur durch den Mechanismus ,,once valid – always valid" (Gültigkeitskriterium) ist sichergestellt, daß ein einmal gültiges, bestes Indi-

viduum erhalten bleibt – der GA verebbt aber dennoch frühzeitig. Daher waren die Ergebnisse auch meistens schlechter.

Wird SAW-ing angewandt bei dem nur die beste Kandidatenlösung betrachtet wird, so führt dies, im Zusammenhang mit unserem Gültigkeitskriterium, auch zu keiner Änderung der Situation: Meistens war die Population (dank der CRH-Initialisierung) von Beginn an gültig, die beste, gültige Kandidatenlösung wurde ,,konserviert", während die restlichen, gültigen Individuen schrittweise aus der Population gedrängt wurden. Deshalb wurde folgende Strategie angewandt: Zur Überprüfung der Untauglichkeit in der Population wurde nicht das beste sondern das schlechteste Individuum herangezogen. Dies soll sicherstellen, daß über kurz oder lang alle ungültigen Individuen aus der Population gedrängt werden. Alternativ dazu kämen z.B. auch folgende Ansätze in Betracht: Die beste ungültige Lösung wird herangezogen, oder es werden immer mehrere Lösungen betrachtet (bestimmter Prozentsatz der Population).

Probeweise wurde dieses Verhalten für die kleinste und die größte Instanz der Problemklasse F durchgeführt. Die relative Abweichung vom IP- bzw. LP-Optimum war sowohl im Mittel als auch bestens zumindest nicht schlechter als bei der Lack-Evaluation; meistens war dieser Ansatz sogar eine Spur besser. Nach unseren Erfahrungen dürfte dieser Ansatz aber mehr Generationen benötigen, nachdem ungültige Individuen länger in der Population verbleiben dürfen. Daher wurde in weiterer Folge die Lack-Evaluation (getrennte Behandlung von Untauglichkeit und Fitness) angewandt.

Größe		IP \ LP					
m	n	Optimum	Art	% gap_{avg}	% gap_{best}	Gen_{avg}	Gen_{best}
5	100	2755	Lack	0.41	0.00	507714	64840
			Cond	2.29	1.27	138294	8787
	200	5294	Lack	0.35	0.02	647177	102861
			Cond	0.75	0.30	357269	26271
	400	10745	Lack	0.25	0.11	1235605	262845
			Cond	0.55	0.12	550825	193171
10	100	2276.8	Lack	3.94	3.39	1019383	579469
			Cond	9.64	6.64	131855	10782
	200	4644.6	Lack	3.12	2.70	1565513	331675
			Cond	6.00	3.60	266125	29704
	400	9372.7	Lack	2.78	1.96	2897607	1322219
			Cond	6.42	3.72	582852	153454
20	100	2145.1	Lack	8.38	6.57	1514784	107817
			Cond	11.69	9.04	483329	56309
	200	4310.1	Lack	6.55	4.99	2227943	565490
			Cond	10.32	8.65	379069	61070
	400	8479.4	Lack	7.15	5.49	3984825	1450552
			Cond	9.52	6.84	2002344	576744
40	100	2110.1	Lack	18.26	16.39	1191719	428656
			Cond	19.32	18.15	862888	173384
	200	4086.5	Lack	12.85	10.80	2461344	640107
			Cond	14.31	13.03	1072131	498400
	400	8274.3	Lack	10.36	9.53	3676777	2288735
			Cond	11.10	10.12	2643971	1506653
80	100	2064.4	Lack	26.76	24.64	1384520	379596
			Cond	26.79	24.64	1330063	379596
	200	4123.4	Lack	17.33	15.29	1738801	175157
			Cond	17.16	15.29	2000505	175157
	400	8167.1	Lack	12.52	11.61	2661273	1480308
			Cond	11.94	11.68	2665459	1480308
Gesamtdurchschnitt			Lack	8.73	7.57	1911932	678688
			Cond	10.52	8.87	1031132	355319

Legende
Lack: Evaluierung gemäß Lack-Evaluation
Cond: Evaluierung gemäß Condition-Evaluation ohne SAW-ing

Tabelle 7.3: *Vergleich unterschiedlicher Evaluierungsroutinen – Instanz F*

7.7 Rekombination

Um den Einfluß des Rekombinations-Operators besser feststellen zu können, wurde der Verbesserungs-Operator für diese Testläufe deaktiviert. Zur Anwendung kamen hierbei die Varianten One Point Crossover, Two Point Crossover und Uniform Crossover. Die Unterschiede zwischen diesen Operatoren sind sehr gering, wird der Verbesserungsoperator hinzugezogen so werden diese noch geringer. Festzustellen war aber, daß der 1-Point bzw. 2-Point Crossover-Operator im Schnitt 25% mehr gültige Nachkommen als der Uniform-Operator erzeugt. Der Uniform-Operator wirkt eher destruktiv als die beiden anderen Operatoren, dafür ist die Variation (die Anzahl unterschiedlicher Lösungen) höher. Dadurch ist aber auch die Wahrscheinlichkeit größer, Lösungen zu erzeugen, die die Kapazitätsbedingungen verletzen.

Der 1-Point-Crossover und 2-Point-Crossover erzielten sehr ähnliche Ergebnisse. Dies ist auch nicht weiter verwunderlich, nachdem aufgrund der alternativen Repräsentation der Individuen (siehe Abschnitt 5.1) die Auswirkungen als gleichwertig zu erachten sind. Es lagen mehr gültige Lösungen vor, als beim Ansatz mit Uniform-Crossover. Dies läßt sich damit begründen, daß durch das Uniform-Crossover die Maschinen-Aufgaben Zuweisungen eher auseinander gerissen werden. Umso mehr die einzelnen Gene (Maschinen) von einander getrennt werden, desto höher ist die Wahrscheinlichkeit, daß die Kapazitätsbedingungen der einzelnen Maschinen verletzt werden. Beim 1/2-Point-Crossover bleibt die Struktur der Genome eher erhalten, dafür sind diese weniger erfolgreich beim Erzeugen neuer Lösungsstrukturen.

Daher haben wir uns bei der Rekombination für den 1-Point Crossover Operator entschieden, analog zu Chu und Beasley.

7.8 Mutation

Die Mutation dient zum Generieren neuer, verloren gegangener Informationen in der Population (sekundärer Operator) und wurde insgesamt zweimal pro Kind-Chromosome angewendet. Abhängig von der Anzahl der Aufgaben entspricht das 2 bis 0,5% der Chromosomegröße. Bei der Mutation wurde zwischen der Zufallsmutation (*bit flip*), der Vertauschungsmutation (*swap mutation, exchange-based mutation*) und der Mutation nach der Heuristik von Martello und Toth unterschieden.

Die schlechtesten Ergebnisse liefert die Mutation mit Zufallsbelegung (siehe 6.9), nachdem diese die Kapazitätsbeschränkungen der einzelnen Maschinen in keinster Weise berücksichtigt. Besser schließt die Swap-Mutation ab, weil bei dieser durch den Austausch der Aufgaben zweier Maschinen, die Kapazitätsgrenzen der Maschinen eher gewahrt werden. Die Mutation basierend auf MTH liegt eine Spur vor der Swap-Mutation. Dies liegt darin begründet, daß versucht wird eine Belegung innerhalb der Kapazitätsgrenzen zu finden bzw. jene mit der geringsten Verletzung. Außerdem inkorporiert sie einen eigenen Optimierungsschritt (entspricht Heuristic Improvement Operator, Phase-2), der sich ebenfalls positiv auswirken kann (siehe Abbildung 7.2).

In weiterer Folge wurde die Mutation basierend auf der Heuristik nach Martello und Toth angewandt.

7.9 Reparatur-Operator

Wird der wissensbasierte Operator angewendet, so stieg die Qualität der Lösungen merklich an – in Hinblick auf das Optimierungsziel – als auch die Anzahl der generierten gültigen Lösungen. So haben schon Chu und Beasley in Ihrer Arbeit gezeigt, daß der GA in Kombination mit dem Reparatur-Operator, bessere Lösungen liefert – die Laufzeit nimmt allerdings um ca. 30% zu. Da das erklärte Ziel dieser Arbeit ist, die besten bekannten Lösungen zu übertreffen, ergo den Ansatz von Chu und Beasley, interessiert uns in weiterer Folge nur mehr der GA mit aktivierten Reparatur-Operator.

Wie aus Abbildung 7.3 hervorgeht, schließt unser Ansatz besser ab als der Ansatz von Chu und Beasley. Zum Vergleich wurde der GA (basierend auf unserem Ansatz) ohne Reparaturoperatur abgebildet – hierzu ist zu betonen, daß die Endergebnisse stärker gestreut waren. Dabei ist ersichtlich welchen Einfluß die Reparatur/Optimierung auf unseren GA hat.

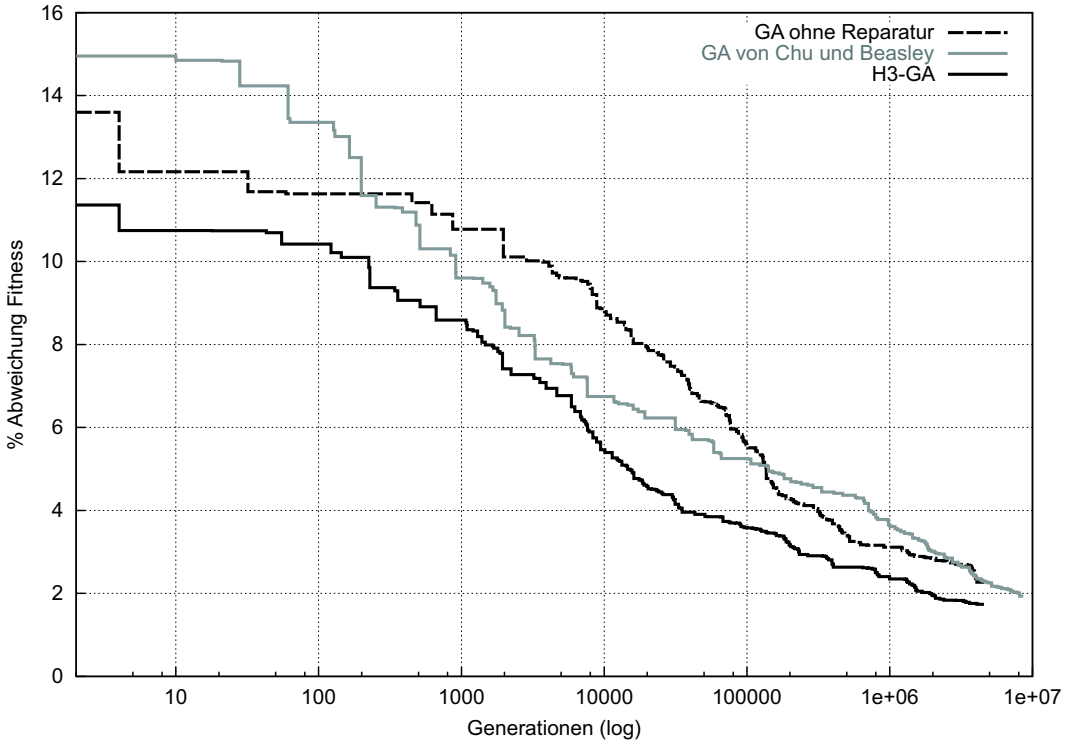

Abbildung 7.3: *Vergleich verschiedener GA-Ansätze (Klasse D-20·200)*

7.10 Zusammenfassung

Es ergeben sich folgende Empfehlungen:

- Variablen-Reduktions-Schema nicht aktiv,
- Initialisierung mittels Constraint-Ratio-Heuristik / LP-Lösung,
- Optimierung der initialen Population,
- Bewertung mittels Lack-Evaluation,
- binäre Tournament-Selektion,
- Rekombination mittels 1-Point Crossover,
- Mutation basierend auf Heuristik von Martello und Toth,
- Reparaturoperatur von Chu und Beasley,
- Duplikatprüfung,
- Ersetzung aufgrund Untauglichkeit (bzw. schlechtestes Individuum) und
- Abbruchkriterium aufgrund relativer Generationsanzahl.

In Abbildung 7.4 wird der Ansatz von Chu und Beasley mit unserem eigenen (Heuristik-GA, H3-GA) verglichen. Dabei werden jeweils die Abweichungen der besten Lösung vom IP- bzw. LP-Optimum (von jeweils zehn Testläufen) über alle Probleminstanzen gegenübergestellt.

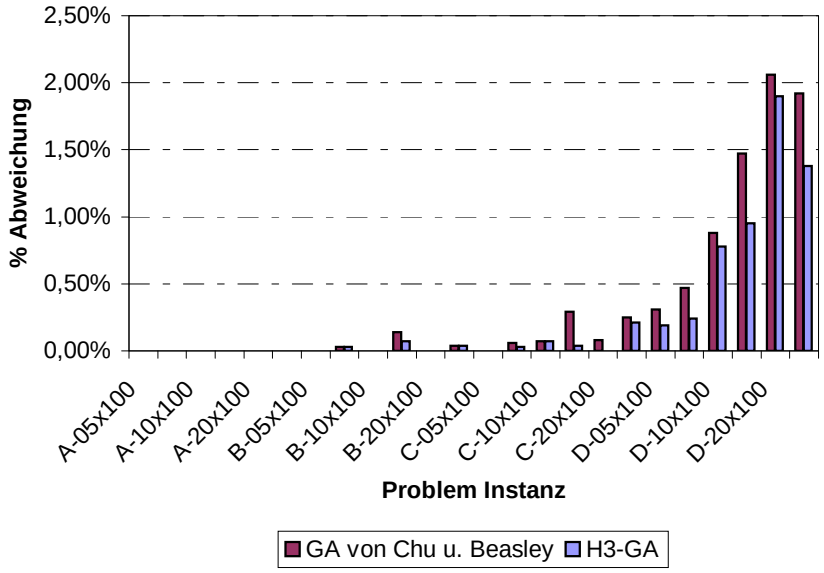

Abbildung 7.4: *Vergleich der Fitness von unterschiedlichen GA-Ansätzen*

Für die Klassen A bis C, bei denen Chu und Beasley bereits Lösungen liefern, die nahe dem Optimum (*near-optimal*) sind, war unser Ansatz mindestens gleich gut unter zumeist erheblich weniger benötigten Generationen. Bei der komplexesten Problemklasse (D) waren die Ergebnisse eindeutig besser, widerum bei gleichzeitig weniger benötigten Generationen. In Abbildung 7.5 ist die durchschnittliche Laufzeit, bis zur Lösung, angegeben.

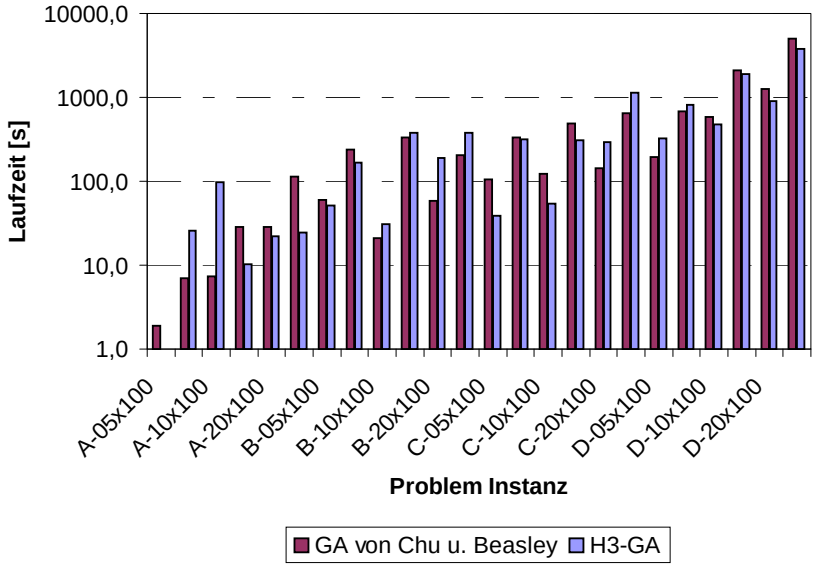

Abbildung 7.5: *Vergleich der Laufzeit von unterschiedlichen GA-Ansätzen*

Tabelle 7.4 zeigt den Vergleich zwischen der besten Lösung von Chu und Beasley und unserem Ansatz relativ zum IP-/LP-Optimum. Die Werte für die Generationsanzahl und die Laufzeit bis zur Endlösung wurden gemittelt über alle zehn Testläufe. Die Werte von dem Ansatz von Chu und Beasley wurden aus [5] übernommen. Damit die Werte mit unseren vergleichbar sind – hinsichtlich Implementierung, zugrundeliegender Hardware, etc. – wurden die Testläufe von Chu und Beasley auf unseren Systemen wiederholt. Die Zahlen bzgl. Generationsanzahl sowie die Zeit bis zur Lösung stellen somit angenäherte (simulierte) Werte dar und wurden den Ergebnissen von Chu und Beasley gegenübergestellt. Wie wir meinen stellt dies trotzdem eine valide Vorgangsweise dar, um die Leistungen der unterschiedlichen Ansätze besser vergleichen zu können.

Weiters ist in der Tabelle die Wahrscheinlichkeit p des Zweistichproben t-Test (*Student t-test*)[1] angegeben. Bei unseren Tests kommt die einseitige Variante (*directional*) zur Anwendung (bei einseitiger Fragestellung halbieren sich die Signifikanzniveaus), da wir davon ausgehen, daß die erzielten Ergebnisse besser sind als die Referenz; weiters wird ungleiche Varianz zu Grunde gelegt. Das eben gesagte bezieht sich auch auf alle folgenden Tabellen in denen der Zweistichproben t-Test (p) angeführt ist.

[1]Der Zweistichproben t-Test dient zum Vergleichen von (nur) zwei Gruppen von ungepaarten Daten. Mit t-Tests kann geprüft werden, ob die Mittelwerte zweier Stichproben gleich sind.

	Größe		IP \ LP		GA von Chu and Beasley					H3-GA					
Typ	m	n	Optimum	% IG	% gap_{avg}	% SD	% gap_{best}	* gen_{sol}	* t_{sol}	% gap_{avg}	% SD	% gap_{best}	gen_{sol}	t_{sol}	% p
A	5	100	1698	0.02	∘	0.00	∘	1469	1.9	∘	0.00	∘	524	1.0	50.00
		200	3235	0.01	∘	0.00	∘	173	7.0	∘	0.00	∘	181	25.9	50.00
	10	100	1360	0.11	∘	0.00	∘	356	7.3	∘	0.00	∘	9046	97.1	50.00
		200	2623	0.00	∘	0.00	∘	1447	28.5	∘	0.00	∘	2531	10.2	50.00
	20	100	1158	0.09	∘	0.00	∘	404	28.5	∘	0.00	∘	688	22.2	50.00
		200	2339	0.07	∘	0.00	∘	444	112.7	∘	0.00	∘	712	24.2	50.00
B	5	100	1843	0.64	0.35	0.31	∘	309088	59.3	0.03	0.08	∘	210424	51.2	0.61
		200	3552	0.13	0.30	0.15	0.03	780172	240.9	0.12	0.05	0.03	362588	165.4	0.22
	10	100	1407	0.46	0.07	0.07	∘	76341	20.9	∘	0.00	∘	42048	30.9	0.75
		200	2827	0.43	0.34	0.13	0.14	807095	335.0	0.26	0.14	0.07	509780	381.8	10.20
	20	100	1166	0.94	0.07	0.06	∘	101031	58.2	0.14	0.21	∘	120872	189.3	18.89
		200	2339	0.34	0.10	0.05	0.04	167451	204.0	0.12	0.08	0.04	212798	378.5	25.50
C	5	100	1931	0.37	0.38	0.21	∘	612830	104.1	0.15	0.18	∘	165421	39.2	1.25
		200	3456	0.15	0.23	0.11	0.06	1063783	335.8	0.07	0.03	0.03	707290	320.0	0.06
	10	100	1402	1.08	0.29	0.26	0.07	556707	123.4	0.11	0.09	0.07	137447	54.1	4.14
		200	2806	0.38	0.48	0.13	0.29	1203979	492.4	0.22	0.16	0.04	408855	304.6	0.06
	20	100	1243	1.98	0.51	0.26	0.08	363620	142.7	0.26	0.13	∘	342068	289.6	0.99
		200	2391	0.59	0.62	0.19	0.25	930809	649.6	0.56	0.18	0.21	827683	1140.1	23.79
D	5	100	6353	0.12	0.66	0.21	0.31	989664	195.3	0.31	0.06	0.19	1248014	327.4	0.03
		200	12736.2	–	0.71	0.15	0.47	1894286	676.4	0.39	0.16	0.24	1635381	814.7	0.02
	10	100	6323.4	–	1.65	0.38	0.88	2016763	591.4	1.24	0.22	0.78	1083907	472.6	0.74
		200	12418.3	–	1.68	0.15	1.47	3816928	2085.8	1.17	0.18	0.95	2259536	1909.7	< 0.01
	20	100	6142.5	–	2.70	0.37	2.06	2491521	1272.0	2.36	0.31	1.90	1126251	902.3	2.41
		200	12217.7	–	2.54	0.30	1.92	5294138	5039.3	2.00	0.33	1.38	2457659	3825.7	0.09

∘ = integer optimal value
* = values approximated

Tabelle 7.4: *Ergebnisse der GAP Testdaten A-D*

7.11 GA-LP

In den folgenden zwei Tabellen (7.5 und 7.6) werden unsere Ansätze: H3-GA und der GA basierend auf der LP-Lösung, in weiterer Folge LP-GA genannt, mit den Ergebnissen des Programms CPLEX verglichen. Wie bereits erwähnt, beziehen sich die Abweichungen % gap (bestens bzw. im Mittel) auf das jeweilige ganzzahlige Optimum sofern bekannt, anderenfalls auf das LP-Optimum. Das gleiche gilt für die relative Standardabweichung (% *SD*).

In den Tabellen sind die Endlösungen von CPLEX ersichtlich (% gap_{sol}) – als Abbruchkriterium für die CPLEX-Lösungen wurden jeweils 3 Stunden herangezogen. In vielen Fällen war jedoch bereits vorher der Speicher (ca. 1 GB) erschöpft, sodaß die Ergebnisse zu diesem Zeitpunkt angeführt sind. Wurden die Rechenläufe von CPLEX zeitlich nicht beschränkt (sondern nur durch den vorhandenen, physikalischen Speicher), so waren die Ergebnisse nur unwesentlich besser (maximale Differenz 0,44% Prozentpunkte, im Schnitt um $0,04$% Prozentpunkte).

Wie aus den Tabellen ersichtlich ist, liefern sowohl CPLEX als auch unsere beiden, GA-basierenden Ansätze für die Testklasse E die besten Ergebnisse. Am anderen Ende der Reihung befindet sich die Testklasse F, dazwischen ist die Testklasse D. Die Ansätze profitieren offensichtlich von den Testklassen mit engen Kapazitätsbeschränkungen (2.2). Zur Erinnerung, Testklasse E entspricht Testklasse D mit dem Unterschied, daß die Kapazitätsbeschränkungen auf 70% reduziert sind, bei Testklasse F widerum ist die Kapazität auf 200% von Klasse D angehoben. Die Arbeit des Branch-and-Bound Ansatzes wird durch vorzeitiges Ausloten bei strengeren Beschränkungen erleichtert. Der GA-Ansatz auf der anderen Seite dürfte bei weiten Beschränkungen zu früh konvergieren (als Abbruchkriterium für alle Testinstanzen wurden einheitlich 500.000 Generationen gewählt).

Im direkten Vergleich der Ansätze CPLEX und H3-GA schneidet die CPLEX-Variante in vielen Fällen besser ab. Der H3-GA konnte wenn überhaupt nur dort punkten, wo die durchschnittliche Maschinenbelegung (Relation $\frac{n}{m}$) sehr gering ist (≤ 5). Beim Vergleich CPLEX zu LP-GA, schneidet der LP-basierende GA in den meisten Fällen besser ab. Im Mittel ist die beste, gefundene Lösung des H3-GA Ansatzes um nicht mehr als 0,25% Prozentpunkte schlechter als der CPLEX-Ansatz. Andererseits schließt die beste, gefundene Lösung des LP-GA im Mittel um 2,47% Prozentpunkte besser ab als beim CPLEX-Ansatz.

Es hat ganz den Anschein, als wäre es für die Ansätze schwerer bei kleinem Verhältnis Aufgaben zu Maschinen die optimale Lösung zu finden bzw. anzunähern. Dies dürfte darin begründet sein, daß bei geringer, durchschnittlicher Maschinenbelegung der Anteil der fraktionalen Belegungen im Verhältnis zu den integralen Belegungen einer LP-Lösung höher ist.

	Größe		IP \ LP		CPLEX	H3-GA			
Typ	m	n	Optimum	% IG	% gap_{sol}	% gap_{avg}	% gap_{best}	gen_{sol}	t_{sol}
D	5	100	6353	0.12	0.00	0.31	0.19	1248014	327.4
		200	12736.2	–	0.09	0.39	0.24	1635381	814.7
		400	25670	0.02	0.02	0.44	0.28	3495381	3609.7
	10	100	6323.5	–	0.72	1.24	0.78	1083907	472.6
		200	12418.4	–	0.30	1.17	0.95	2259536	1909.7
		400	25274.8	–	0.18	1.31	1.08	4061815	7042.7
	20	100	6142.5	–	2.37	2.36	1.90	1126251	902.3
		200	12217.7	–	1.03	2.00	1.38	2457659	3825.7
		400	24546.8	–	0.51	1.97	1.82	4041905	13049.6
	40	100	6092.0	–	3.96	3.72	3.40	1361463	2072.9
		200	12244.9	–	2.22	3.06	2.68	2357294	7116.2
		400	24371.8	–	1.10	2.81	2.43	3574368	23374.1
	80	100	6110.5	–	6.60	7.01	6.59	846320	2878.6
		200	12132.3	–	2.87	3.76	3.18	2001149	13075.6
		400	24177.0	–	2.00	3.02	2.68	3195382	55945.1
E	5	100	7757	0.17	0.00	0.24	0.13	552212	147.1
		200	15611	0.04	0.00	0.23	0.14	1281834	669.8
		400	30794	0.02	0.01	0.28	0.20	2744401	3026.7
	10	100	7387.8	–	0.61	0.91	0.67	1005960	447.3
		200	15039.8	–	0.25	0.96	0.70	1807355	1557.2
		400	29977.9	–	0.09	0.94	0.72	3390020	6136.3
	20	100	7348.2	–	1.32	1.74	1.37	1458153	1171.4
		200	14765.2	–	0.89	1.70	1.32	1955937	3119.4
		400	29500.3	–	0.34	1.60	1.39	4255476	14462.3
	40	100	7316.1	–	3.32	3.11	2.69	976572	1573.5
		200	14630.4	–	1.85	2.20	1.88	1889586	6061.5
		400	29186.6	–	0.69	2.14	1.80	2997713	20536.8
	80	100	7650	5.0	0.00	0.78	0.60	737647	2614.1
		200	14566.7	–	2.17	2.93	2.25	1584730	11078.3
		400	29161.3	–	1.57	2.49	2.06	3063944	56250.8
F	5	100	2755	0.53	0.00	0.41	0.00	507714	123.0
		200	5294	0.18	0.00	0.35	0.02	647177	293.2
		400	10745	0.07	0.00	0.25	0.11	1235605	1130.0
	10	100	2276.8	–	1.99	3.95	3.39	1019383	386.4
		200	4644.6	–	1.13	3.12	2.70	1565513	1127.4
		400	9372.7	–	0.46	2.78	1.96	2897607	4207.9
	20	100	2145.1	–	8.15	8.38	6.57	1514783	1010.3
		200	4310.1	–	4.73	6.55	4.99	2227943	2878.4
		400	8479.4	–	2.38	7.15	5.49	3984825	10390.0
	40	100	2110.1	–	21.28	18.27	16.39	1191719	1572.9
		200	4086.5	–	10.14	12.86	10.81	2461344	6228.2
		400	8274.3	–	4.05	10.36	9.53	3676777	19192.9
	80	100	2064.4	–	31.37	26.77	24.64	1620919	3640.1
		200	4123.4	–	19.05	17.33	15.29	1738801	9828.6
		400	8167.1	–	9.18	12.55	11.61	2661273	32432.1

Tabelle 7.5: *Vergleich CPLEX versus H3-GA*

	Größe		IP \ LP		CPLEX	LP-GA			
Typ	m	n	Optimum	% IG	% gap_{sol}	% gap_{avg}	% gap_{best}	gen_{sol}	t_{sol}
D	5	100	6353	0.12	0.00	0.08	0.05	308338	82.9
		200	12736.2	–	0.09	0.15	0.11	424045	223.1
		400	25670	0.02	0.02	0.03	0.02	241022	268.2
	10	100	6323.5	–	0.72	0.75	0.58	299368	132.2
		200	12418.4	–	0.30	0.26	0.21	588684	511.9
		400	25274.8	–	0.18	0.16	0.13	633166	1139.7
	20	100	6142.5	–	2.37	1.67	1.23	656233	515.2
		200	12217.7	–	1.03	0.81	0.71	633267	1038.3
		400	24546.8	–	0.51	0.44	0.38	876090	2988.9
	40	100	6092.0	–	3.96	3.37	2.99	784784	1206.7
		200	12244.9	–	2.22	1.63	1.30	902377	3034.4
		400	24371.8	–	1.10	0.86	0.74	1325198	9049.5
	80	100	6110.5	–	6.60	7.01	6.44	948131	3047.5
		200	12132.3	–	2.87	3.05	2.63	1377981	9664.6
		400	24177.0	–	2.00	1.57	1.44	2370939	41258.3
E	5	100	7757	0.17	0.00	0.07	0.04	232404	65.3
		200	15611	0.04	0.00	0.04	0.03	211499	120.6
		400	30794	0.02	0.01	0.03	0.01	192115	231.9
	10	100	7387.8	–	0.61	0.60	0.48	359258	170.3
		200	15039.8	–	0.25	0.24	0.20	437348	408.9
		400	29977.9	–	0.09	0.11	0.09	591807	1126.7
	20	100	7348.2	–	1.32	0.97	0.79	891059	746.0
		200	14765.2	–	0.89	0.61	0.49	1041321	1742.7
		400	29500.3	–	0.34	0.33	0.24	787977	2892.4
	40	100	7316.1	–	3.32	2.86	2.66	976760	1600.8
		200	14630.4	–	1.85	1.32	1.19	1164883	3899.2
		400	29186.6	–	0.69	0.63	0.56	1099475	8387.9
	80	100	7650	5.0	0.00	0.57	0.43	806626	2862.6
		200	14566.7	–	2.17	2.43	2.21	1360348	9879
		400	29161.3	–	1.57	1.20	1.04	2063900	34618.8
F	5	100	2755	0.53	0.00	0.18	0.18	90928	21.6
		200	5294	0.18	0.00	0.06	0.02	307978	137.0
		400	10745	0.07	0.00	0.05	0.02	199224	196.7
	10	100	2276.8	–	1.99	2.88	2.47	365538	135.9
		200	4644.6	–	1.13	1.22	1.02	702420	505.4
		400	9372.7	–	0.46	0.65	0.47	1154144	1663.1
	20	100	2145.1	–	8.15	5.64	4.89	536732	362.1
		200	4310.1	–	4.73	2.96	2.43	760128	1021.5
		400	8479.4	–	2.38	1.51	1.20	1090608	3034.6
	40	100	2110.1	–	21.28	16.94	14.88	1090204	1395.7
		200	4086.5	–	10.14	7.02	6.08	788030	2213.0
		400	8274.3	–	4.05	3.80	3.43	1202311	7212.2
	80	100	2064.4	–	31.37	26.57	25.66	1092314	2932.0
		200	4123.4	–	19.05	15.67	14.27	2624327	13613.2
		400	8167.1	–	9.18	7.12	6.14	1457916	22542.5

Tabelle 7.6: *Vergleich CPLEX versus LP-GA*

In den Tabellen 7.7, 7.8 und 7.9 wird der Vergleich zwischen dem H3-GA und dem LP-GA über die Testklassen D bis F gemacht. Dabei ist ersichtlich, daß der GA-Ansatz basierend auf der LP-Lösung (Abschnitt 6.10) eindeutig besser abschließt als der Ansatz, der auf der Initialisierung mittels Constraint-Ratio-Heuristik (Abschnitt 6.3.3) aufbaut. Bei der LP-initialisierung sind die Endergebnisse nicht nur besser, sondern auch stabiler (die Streuung der einzelnen Ergebnisse ist geringer). Gleichzeitig ist die Anzahl der benötigten Generationen geringer und damit die Laufzeit.

Die einzige Ausnahme tritt für den Fall auf, daß die durchschnittliche Maschinenbelegung sehr gering ist ($m \cdot n = 80 \cdot 100$). In diesem Fall besteht die LP-Lösung aus vielen fraktionalen Belegungen – der Anteil der integralen Belegungen wird mit abnehmenden Verhältnis Aufgaben zu Maschinen ebenfalls geringer (im schlimmsten Fall lagen nur noch zu 41% integrale Belegungen vor, der Rest bestand aus fraktionalen Belegungen). Anders ausgedrückt, bei größerer, durchschnittlicher Maschinenbelegung war auch der Anteil der kompletten (ganzen) Maschinenbelegungen in der LP-Lösung höher.

Es ist zu beachten, daß sehr wohl beide Initialisierungsarten oft zu gültigen Kandidatenlösungen führen. Die CRH-Initialisierung liefert für die Testklasse D meist, bei der Testklasse E in zwei Drittel der Fälle und für die Testklasse F immer zu gültigen Ausgangslösungen. Die LP-Initialisierung hingegen liefert im Schnitt nur zu ca. 11% ungültigen Kandidatenlösungen; daß bedeutet im einzelnen: D immer gültig, E in 80% und F in $86,6\%$ der Fälle.

Der Vorteil der LP-Initialisierung liegt darin, daß die Ausgangslösungen einen besseren Zielfunktionswert haben und mit den Endlösungen mehr übereinstimmen. Das Wissen über die LP-Lösung (im speziellen die ,,integralen" Belegungen) trägt dazu bei, daß die Kandidatenlösungen stärker in Richtung optimaler Lösung gerichtet sind.

Abschließend sind in den Abbildungen 7.6 bis 7.11 der Vergleich über alle drei Ansätze (CPLEX, Standard GA und GA basierend auf LP-Lösung) pro Problemklasse (D bis F) angegeben. In diesen ist anschaulich dargestellt, daß der Ansatz basierend auf dem Branch-and-Bound Verfahren (CPLEX) sich meistens zwischen dem Standard GA und dem LP-basierenden GA platzieren kann. Während der LP-GA eindeutig besser ist als der H3-GA, sowohl bezüglich der Qualität der Lösungen als auch betreffend der Anzahl der Generationen.

	Größe		IP \ LP		H3-GA					LP-GA					
Typ	m	n	Optimum	% IG	% gap_{avg}	% SD	% gap_{best}	gen_{sol}	t_{sol}	% gap_{avg}	% SD	% gap_{best}	gen_{sol}	t_{sol}	% p
D	5	100	6353	0.12	0.31	0.06	0.19	1248014	327.4	0.08	0.03	0.05	308338	82.9	< 0.01
		200	12736.2	–	0.39	0.16	0.24	1635381	814.7	0.15	0.02	0.11	424045	223.1	0.05
		400	25670	0.02	0.44	0.13	0.28	3495381	3609.7	0.03	0.01	0.02	241022	268.2	< 0.01
	10	100	6323.5	–	1.24	0.22	0.78	1083907	472.6	0.75	0.08	0.58	299368	132.2	< 0.01
		200	12418.4	–	1.17	0.18	0.95	2259536	1909.7	0.26	0.04	0.21	588684	511.9	< 0.01
		400	25274.8	–	1.31	0.16	1.08	4061815	7042.7	0.16	0.02	0.13	633166	1139.7	< 0.01
	20	100	6142.5	–	2.36	0.31	1.90	1126251	902.3	1.67	0.20	1.23	656233	515.2	< 0.01
		200	12217.7	–	2.00	0.33	1.38	2457659	3825.7	0.81	0.10	0.71	633267	1038.3	< 0.01
		400	24546.8	–	1.97	0.12	1.82	4041905	13049.6	0.44	0.05	0.38	876090	2988.9	< 0.01
	40	100	6092.0	–	3.72	0.21	3.40	1361463	2072.9	3.37	0.29	2.99	784784	1206.7	0.48
		200	12244.9	–	3.06	0.30	2.68	2357294	7116.2	1.63	0.17	1.30	902377	3034.4	< 0.01
		400	24371.8	–	2.81	0.29	2.43	3574368	23374.1	0.86	0.08	0.74	1325198	9049.5	< 0.01
	80	100	6110.5	–	7.01	0.32	6.59	846320	2878.6	7.01	0.32	6.44	948131	3047.5	49.15
		200	12132.3	–	3.76	0.36	3.18	2001149	13075.6	3.05	0.24	2.63	1377981	9664.6	0.01
		400	24177.0	–	3.02	0.18	2.68	3195382	55945.1	1.57	0.08	1.44	2370939	41258.3	< 0.01

Tabelle 7.7: *Vergleich H3-GA versus LP-GA, Testklasse D*

	Größe		IP \ LP		H3-GA					LP-GA					
Typ	m	n	Optimum	% IG	% gap_{avg}	% SD	% gap_{best}	gen_{sol}	t_{sol}	% gap_{avg}	% SD	% gap_{best}	gen_{sol}	t_{sol}	% p
E	5	100	7757	0.17	0.24	0.07	0.13	552212	147.1	0.07	0.02	0.04	232404	65.3	< 0.01
		200	15611	0.04	0.23	0.07	0.14	1281834	669.8	0.04	0.01	0.03	211499	120.6	< 0.01
		400	30794	0.02	0.28	0.05	0.20	2744401	3026.7	0.03	0.01	0.01	192115	231.9	< 0.01
	10	100	7387.8	–	0.91	0.20	0.67	1005960	447.3	0.60	0.05	0.48	359258	170.3	0.04
		200	15039.8	–	0.96	0.16	0.70	1807355	1557.2	0.24	0.03	0.20	437348	408.9	< 0.01
		400	29977.9	–	0.94	0.12	0.72	3390020	6136.3	0.11	0.01	0.09	591807	1126.7	< 0.01
	20	100	7348.2	–	1.74	0.23	1.37	1458153	1171.4	0.97	0.12	0.79	891059	746.0	< 0.01
		200	14765.2	–	1.70	0.19	1.32	1955937	3119.4	0.61	0.07	0.49	1041321	1742.7	< 0.01
		400	29500.3	–	1.60	0.20	1.39	4255476	14462.3	0.33	0.05	0.24	787977	2892.4	< 0.01
	40	100	7316.1	–	3.11	0.32	2.69	976572	1573.5	2.86	0.19	2.66	976760	1600.8	3.18
		200	14630.4	–	2.20	0.14	1.88	1889586	6061.5	1.32	0.10	1.19	1164883	3899.2	< 0.01
		400	29186.6	–	2.14	0.23	1.80	2997713	20536.8	0.63	0.05	0.56	1099475	8387.9	< 0.01
	80	100	7650	5.0	0.78	0.11	0.60	737647	2614.1	0.57	0.11	0.43	806626	2862.6	0.04
		200	14566.7	–	2.93	0.29	2.25	1584730	11078.3	2.43	0.13	2.21	1360348	9879	0.03
		400	29161.3	–	2.49	0.28	2.06	3063944	56250.8	1.20	0.10	1.04	2063900	34618.8	< 0.01

Tabelle 7.8: *Vergleich H3-GA versus LP-GA, Testklasse E*

Größe			IP \ LP		H3-GA					LP-GA				
Typ	m	n	Optimum	% IG	% gap_{avg}	% SD	% gap_{best}	gen_{sol}	t_{sol}	% gap_{avg}	% SD	% gap_{best}	gen_{sol}	t_{sol}	% p
F	5	100	2755	0.53	0.41	0.29	0.00	507714	123.0	0.18	0.00	0.18	90928	21.6	2.36
		200	5294	0.18	0.35	0.16	0.02	647177	293.2	0.06	0.02	0.02	307978	137.0	0.03
		400	10745	0.07	0.25	0.07	0.11	1235605	1130.0	0.05	0.02	0.02	199224	196.7	< 0.01
	10	100	2276.8	–	3.95	0.53	3.39	1019383	386.4	2.88	0.21	2.47	365538	135.9	< 0.01
		200	4644.6	–	3.12	0.35	2.70	1565513	1127.4	1.22	0.12	1.02	702420	505.4	< 0.01
		400	9372.7	–	2.78	0.57	1.96	2897607	4207.9	0.65	0.12	0.47	1154144	1663.1	< 0.01
	20	100	2145.1	–	8.38	1.25	6.57	1514783	1010.3	5.64	0.63	4.89	536732	362.1	< 0.01
		200	4310.1	–	6.55	1.10	4.99	2227943	2878.4	2.96	0.36	2.43	760128	1021.5	< 0.01
		400	8479.4	–	7.15	0.91	5.49	3984825	10390.0	1.51	0.15	1.20	1090608	3034.6	< 0.01
	40	100	2110.1	–	18.27	1.26	16.39	1191719	1572.9	16.94	1.16	14.88	1090204	1395.7	1.64
		200	4086.5	–	12.86	1.60	10.81	2461344	6228.2	7.02	0.62	6.08	788030	2213.0	< 0.01
		400	8274.3	–	10.36	0.49	9.53	3676777	19192.9	3.80	0.21	3.43	1202311	7212.2	< 0.01
	80	100	2064.4	–	26.77	1.16	24.64	1620919	3640.1	26.57	0.62	25.66	1092314	2932.0	32.82
		200	4123.4	–	17.33	1.34	15.29	1738801	9828.6	15.67	0.82	14.27	2624327	13613.2	0.32
		400	8167.1	–	12.55	0.59	11.61	2661273	32432.1	7.12	0.64	6.14	1457916	22542.5	< 0.01

Tabelle 7.9: *Vergleich H3-GA versus LP-GA, Testklasse F*

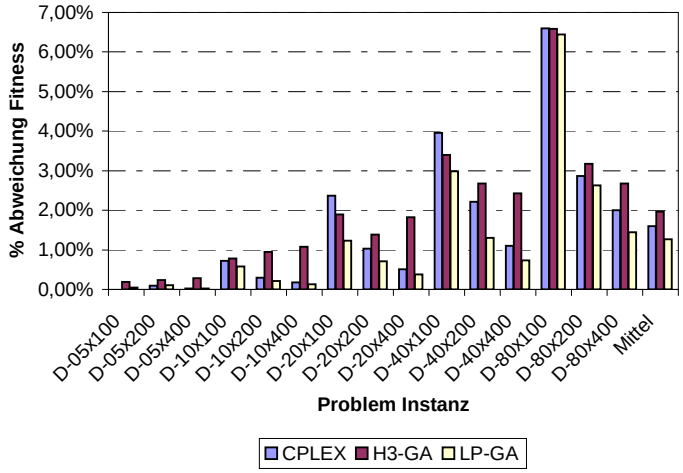

Abbildung 7.6: *Vergleich über Problemklasse D*

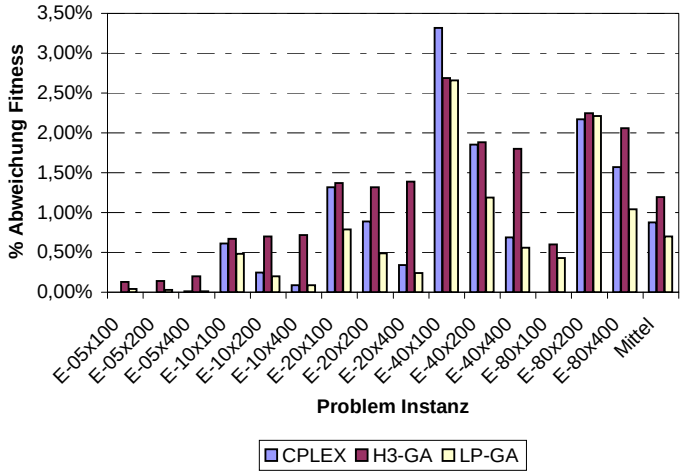

Abbildung 7.7: *Vergleich über Problemklasse E*

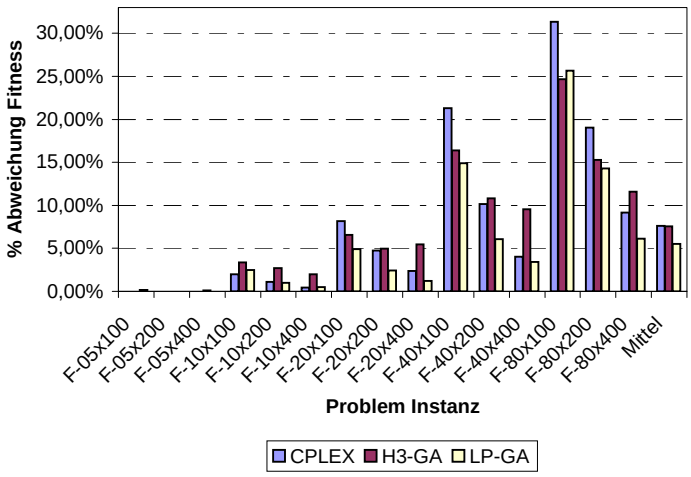

Abbildung 7.8: *Vergleich über Problemklasse F*

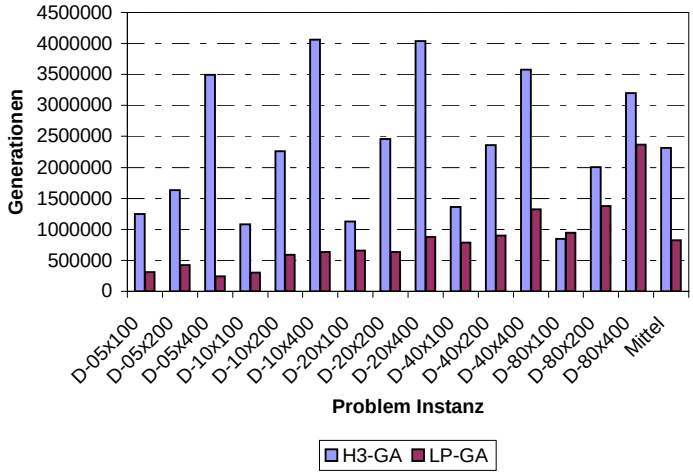

Abbildung 7.9: *Vergleich über Problemklasse D*

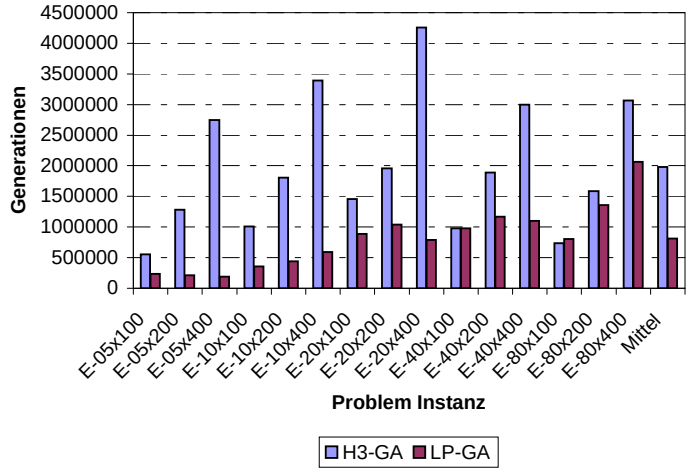

Abbildung 7.10: *Vergleich über Problemklasse E*

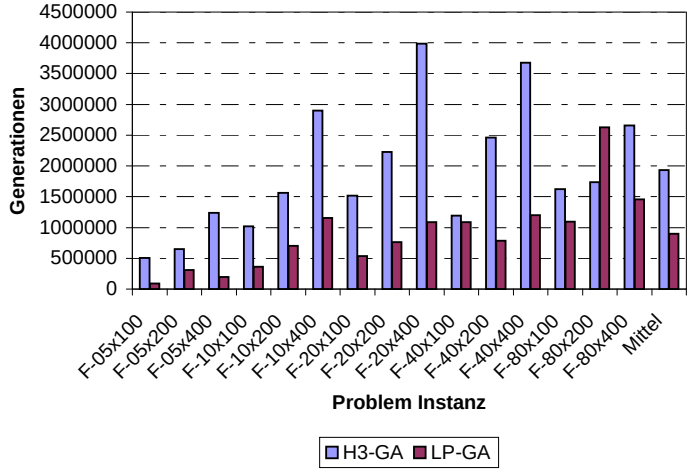

Abbildung 7.11: *Vergleich über Problemklasse F*

7.12 Erkenntnisse

Mit Hilfe des kommerziellen Programmpakets CPLEX (basiert auf Branch-and-Bound Techniken) konnte in 19 von 24 Fällen (ca. 80%) das ganzzahlige Optimum der Standard-Testinstanzen ($A - D$) ermittelt werden. Dadurch zeigte sich, daß der vorhandene Ansatz von Chu und Beasley bereits sehr gute Ergebnisse liefert. Auf der anderen Seite genügen die vorhandenen Testinstanzen, nach heutigen Ansprüchen, nicht mehr. Sie sind zu klein bzw. einfach, um eine Herausforderung an mordene Lösungsalgorithmen darzustellen.

Wir konnten zeigen, daß der GA-Ansatz von Chu und Beasley durch einige, wesentliche Neuerungen noch weiter verbessert werden konnte. Dazu zählen u.a die Generierung von gültigen Kandidatenlösungen in der Ausgangspopulation. Das Vorhandensein (möglichst) gültiger Kandidatenlösungen in der Ausgangspopulation trägt entscheidend dazu bei, daß der GA bessere Ergebnisse liefert. Dazu zählt auch, daß die initiale Generation vor dem eigentlichen Ablauf des GA optimiert wird. Weiters profitierte der GA durch die Bewertung mittels Lack-Evaluation, bei der ungültige Individuen immer schlechter gestellt werden als gültige Individuen und damit bereits früh aus der Population verschwinden.

Die optimale Populationsgröße beträgt für die vorliegenden Testdaten $N = 100$ Individuen. Die Populationsgröße darf nicht unter einen kritischen Wert fallen – bei Halbierung der Population waren die erzielten Ergebnisse erheblich schlechter, während eine Verdoppelung keine nennenswerten Verbesserungen brachte. Um möglichst gute Ergebnisse zu erzielen, ist es wichtig, daß der GA nicht zu rasch konvergiert. Je größer die Konvergenz in den ersten Generationen, umso schlechter die erzielbaren Ergebnisse und umgekehrt. Es zeigte sich, daß das Abbruchkrierium mit 500.000 Generationen in der Regel ausreichend gewählt war.

So wichen die Ergebnisse für die oben erwähnten Testklassen, im Mittel, um nicht mehr als 0,39% vom IP- bzw. LP-Optimum ab (mit einer durchschnittlichen, besten Abweichung von 0,24%). Im Vergleich dazu lag die mittlere Abweichung bei dem Ansatz von Chu und Beasley bei 0,57% und die durchschnittliche, beste Abweichung bei 0,34%. Die besten Ergebnisse lieferte der CPLEX-Ansatz mit 0,19% Abweichung.

Ein interessanter Nebenaspekt ist, daß mit geringerer, durchschnittlicher Maschinenbelegung ($\frac{n}{m}$) der Abstand LP- zu IP-Lösung (integrality gap) größer wird. Dies liegt anscheinend darin begründet, daß mit kleinerem Verhältnis die Anzahl der fraktionalen Belegungen in der LP-Lösung steigt. Bei großer durchschnittlicher Anzahl von Aufgaben pro Maschine ist die ganzzahlige Lösung somit dem LP-Optimum näher.

Aufbauend auf den bisherigen Erkenntnissen wurde der Ansatz LP-GA entwickelt. Mittels einer Heuristik die auf der LP-Lösung aufbaut, werden Kandidatenlösungen für die initiale Population generiert. Durch diese Heuristik konnte die Qualität der initialen Kandidatenlösungen erheblich gesteigert werden und damit die Endlösungen unseres GA-Ansatzes. Ein positiver Nebeneffekt davon war die ebenfalls verringerte Generationsanzahl bzw. Laufzeit.

Um den eigenen Ansatz und die exakten Verfahren, wie z.B. das Programm CPLEX, besser beurteilen zu können, wurden die Testklasse D erweitert und die Testklassen E und F ins Leben gerufen. Für diese ergab sich folgende Reihung der verschiede-

nen Ansätze: der LP-GA liegt klar vor dem CPLEX, gefolgt von dem H3-GA (für den LP-GA gilt, daß die Lösungen im Schnitt sogar besser waren als die besten, gefundenen Lösungen von CPLEX). Der Vorteil des Branch-and-Bound basierenden Ansatzes schwindet mit steigender Problemgröße und Komplexität (weiteren Schranken: Kapazitätsbeschränkungen).

In Abbildung 7.12 wird der H3-GA mit dem LP-basierenden GA verglichen. Hierbei ist ersichtlich, daß der LP-GA mit einer wesentlich besseren Kandidatenlösungen startet und auch zu einem besseren Endergebnis führt.

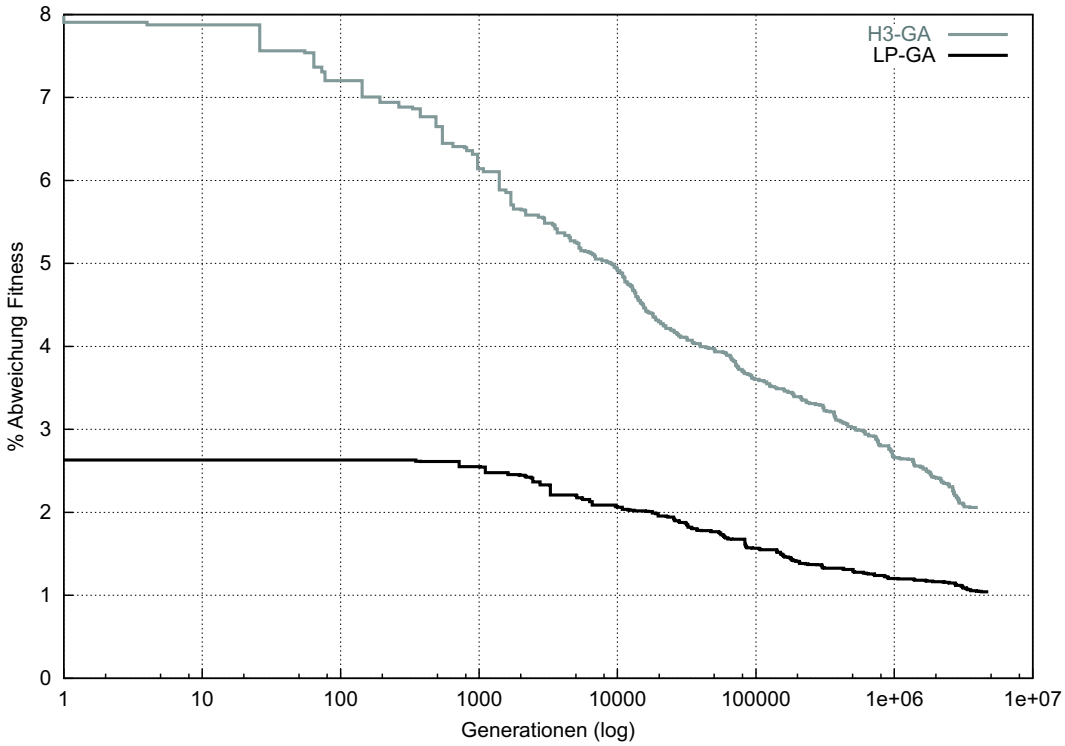

Abbildung 7.12: *Vergleich H3-GA versus LP-GA* ($E - 80 \cdot 400$)

Auch wenn unsere Ansätze nicht garantieren, das ganzzahlige Optimum zu finden, sind diese auch dort anwendbar, wo andere exakte Verfahren (a'la Ansatz von Martello und Toth (Abschnitt 3.2.1) bzw. Savelsbergh (Abschnitt 3.2.2)) a priori ausscheiden. Das größte Problem von Savelsbergh besteht aus insgesamt 1.000 Variablen ($20 \cdot 50$) im Vergleich zu 32.000 Variablen ($80 \cdot 400$) bei unseren Ansätzen. Damit liefern unsere Ansätze sehr gute Ergebnisse für Testinstanzen, die mit exakten Verfahren sonst nicht lösbar wären oder nur angenähert werden könnten, wie im Fall von CPLEX.

Kapitel 8

Implementierung

Im folgenden wird eine Zusammenfassung der in diesem Programm existierenden Klassen und Methoden sowie eine Gebrauchsanleitung für den Programmaufruf gegeben. Die vollständige Dokumentation ist beim Programm enthalten und wurde mit Hilfe des Programms *doxygen* (version 1.2.10) aus dem Sourcecode generiert.

8.1 Allgemeines

Die Implementierung der besprochenen Anwendung erfolgte auf einer Linuxplattform unter Kernel 2.4.19. Für den Genetischen Algorithmus wurde die C++ Bibliothek *EALib* 1.0 [22] verwendet. Diese Bibliothek unterstützt die Programmierung von Evolutionären Algorithmen durch vorgefertigte Standardkomponenten. Als Compiler wurde der GNU C++ (gcc version 2.95.3) verwendet.

8.2 Klassenbeschreibung

8.2.1 Bibliothek EAlib

Wie bereits erwähnt basiert das Programm *GAP:S*[1] auf der C++ Bibliothek EALib von Raidl [22]. Die EALib stellt ein Grundgerüst für Evolutionäre Algorithmen dar. Für die Entwicklung von GAP:S war es ausreichend eine einzige Klasse der EALib zu erweitern; die Chromosomen-Basisklasse *chromosome*. In dieser findet die Anwendung der genetischen Operatoren statt. Sie stellt eine abstrakte Basisklasse für Genome (Chromosomen-Klasse)[2] dar, aufgrund derer eine konkrete Realisierung mittels Ableitung auf den gewünschten Datentyp stattfindet.

[1] Das Akronym GAP:S steht für Generalized Assignment Problem Solver.

[2] Ein Genom [griech.] (Chromosom) ist der Träger der Erbanlagen. Unter einem Genom wird die Gesamtheit aller in einer Zelle vorhandenen Erbanlagen verstanden (der Gene und der genetischen Signalstrukturen). Ein Gen [griech.] stellt einen Erbfaktor bzw. eine in den Chromosomen lokalisierte Erbeinheit dar und ist der eigentliche Träger der Vererbung.

Die konkrete Implementierung dieser Basisklasse muß unter anderem folgende genetische Operatoren enthalten:

- **objective** ()
  Diese Funktion liefert den skalierten Zielfunktionswert (siehe Abschnitt 4.5).
- **initialize** ()
  Die Initialisierung des Chromosoms findet statt (siehe Abschnitt 4.4).
- **mutate** ()
  Das aktuelle Chromosom wird mutiert (siehe Abschnitt 4.8).
- **crossover** ()
  Zwei Chromosomen werden rekombiniert (siehe Abschnitt 4.7).

Die folgende Aufstellung soll einen kurzen Überblick über die Funktionalität der EALib geben:

- genetischer Algorithmus in Form eines Steady-State GA mit Elitismus inkl. dem Ausschluß von Duplikaten (siehe Abschnitt 4.9.3)
- Verwaltung von Individuen in Form einer Populations-Klasse
- Chromosomen-Basisklasse
- Statistik- und Protokollfunktionen über den Ablauf des GA
- komfortable Parameterverarbeitung

8.2.2 GAP:S-Klassen

In diesem Abschnitt wird der grundsätzliche Aufbau des genetischen Programms beschrieben. Dieses gliedert sich in drei Hauptkomponenten: das Chromosome, die Population und den eigentlichen genetischen Algorithmus. Das Chromosome stellt die Datenstruktur zur Repräsentation einer Kandidatenlösung dar und enthält die Erbinformation, im konkreten Fall die Daten einer möglichen Lösung für die Optimierungsaufgabe. Weiters stellt es die verschiedenen GA-Operationen, wie etwa Rekombination und Mutation, zur Verfügung. Zur Verwaltung der Chromosomen dient die Population. Der GA greift über die Population auf die einzelnen Kandidatenlösungen zu und führt die entsprechenden Bearbeitungsschritte aus. Eine Übersicht über die Klassen von GAP:S ist in Abbildung 8.1 (Abschnitt 8.3) ersichtlich.

GAPGenome

Diese Klasse stellt eine Ableitung der Chromosome-Basisklasse der Bibliothek EALib dar und stellt die Verknüpfung zum GA (Steady State GA) über die jeweiligen Methoden her. Der Zugriff auf das Chromosome wird durch die innere Klasse *GAPShell* durch spezielle gekapselte Zugriffsfunktionen vereinfacht. Zur Repräsentation einer

Lösung wurde ein Ganzzahl-Feld (*integer array*) herangezogen. Die Größe dieses Feldes entspricht der Anzahl der zu erfüllenden Aufgaben (jobs), wobei jedes Element einen Wert aus der Menge $\{0, \ldots, m-1\}$ (mögliche Zuweisungen) annehmen kann.

Die eigentliche Funktionalität der GA-Operatoren (Bewertungs- und Penaltyfunktion, Initialisierung der Population, Rekombination, Mutation und Reparaturfunktion) wurde in eigenen Hilfsklassen (BaseEvaluator, BaseInitializer, BaseCrossover, BaseMutator sowie BaseImprovement) ausgelagert. Die eigentlichen Realisierungen leiten sich von diesen abstrakten Basisklassen ab. Über Polymorphismus wird die jeweils gewünschte Instanz des entsprechenden GA-Operators aufgerufen.

GAPData

Diese Klasse übernimmt das Einlesen der Testdaten eines GAP und stellt Zugriffsmethoden auf diese Daten zur Verfügung. Das Variablen-Reduktions-Schema wird ebenfalls von dieser Klasse durchgeführt.

GAPS

Die Datei *gaps.C* stellt das eigentliche Hauptprogramm zum Optimieren von GAP-Aufgaben dar. Die eigentliche Abarbeitung des GA wird an die entsprechenden Module weiterdelegiert. Das Hauptprogramm übernimmt lediglich das Anlegen und Starten des GA sowie das Erzeugen der Chromosomen. Die Bearbeitung des GA wurde zur Gänze der Klasse *steadyStateEA* aus der Bibliothek EALib [22] überlassen.

8.3 Klassen-Hierarchie

Die folgenden Diagramme enthalten die Module mit allen ,,öffentlichen" Klassen, d.h. den Klassen, die nicht nur innerhalb eines Moduls benötigt werden. Insbesondere aber wird die Hierarchie der Schnittstellenimplementierungen und Klassenerweiterungen dargestellt.

Die Abbildung 8.1 gibt einen Überblick über alle Abhängigkeiten zwischen den beteiligten Klassen, insbesondere der Chromosome-Klasse *GAPGenome* und der genetischen Operatoren. In Abbildung 8.2 sind die Basisklassen ,,BaseEvaluator" und ,,BaseMutator" ersichtlich, inklusive ihrer konkreten Realisierungen (Spezialisierung). In Abbildung 8.3 sind die Basisklassen ,,BaseInitializer", ,,BaseCrossover" und ,,BaseImprovement" ersichtlich, inklusive ihrer konkreten Realisierungen.

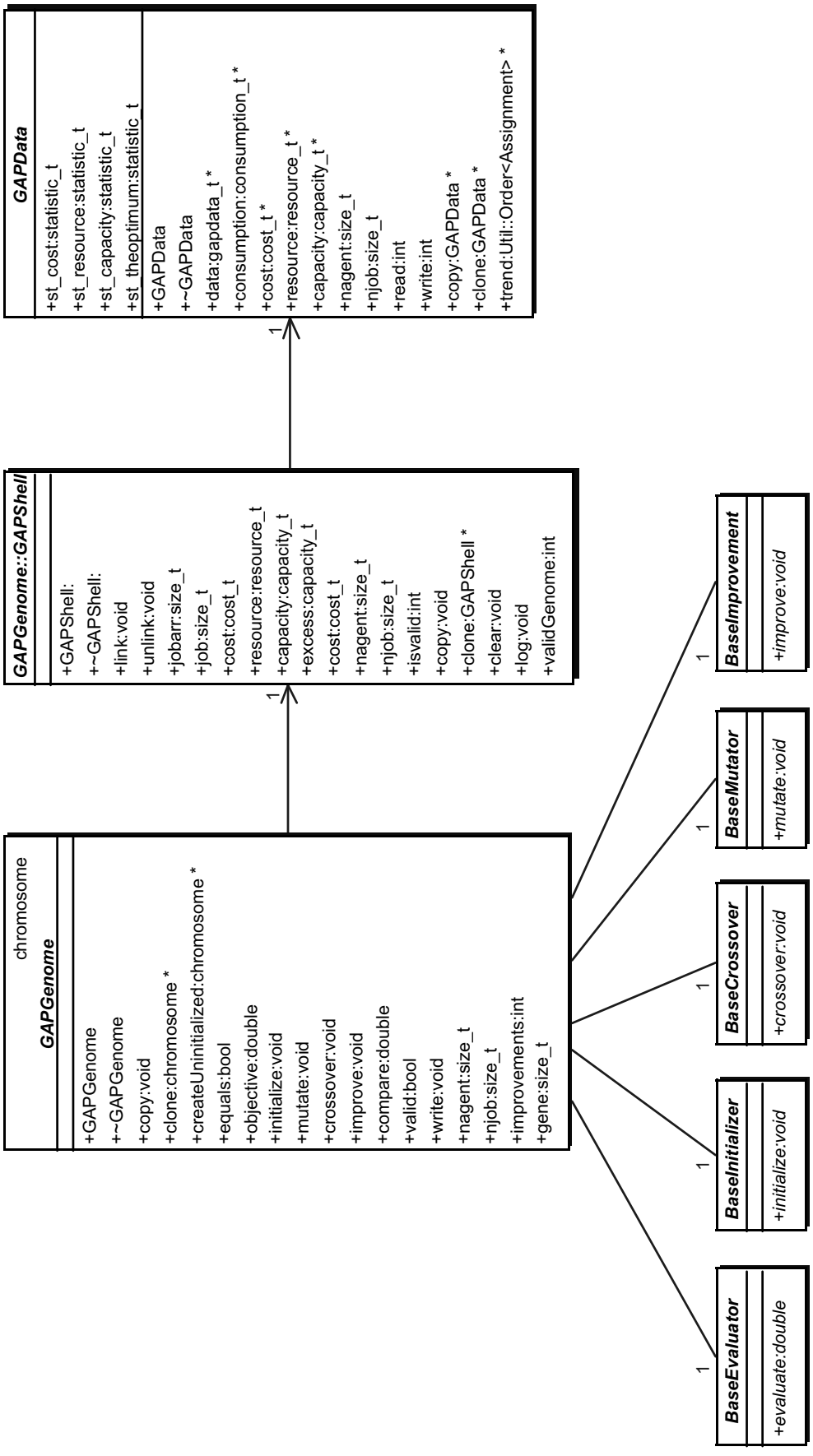

Abbildung 8.1: *Klassen Diagramm – Überblick*

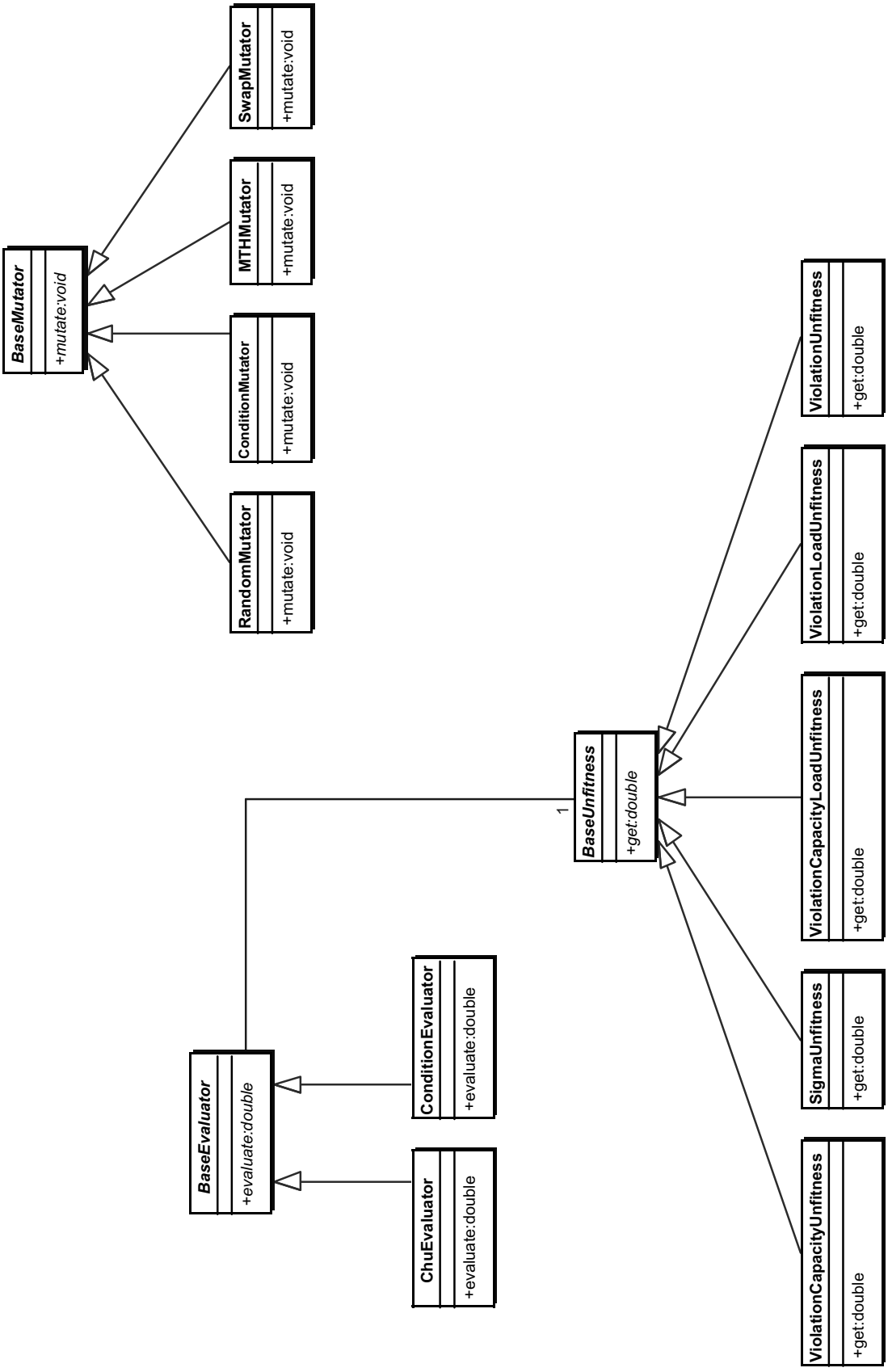

Abbildung 8.2: *Klassen Diagramm – Detail*

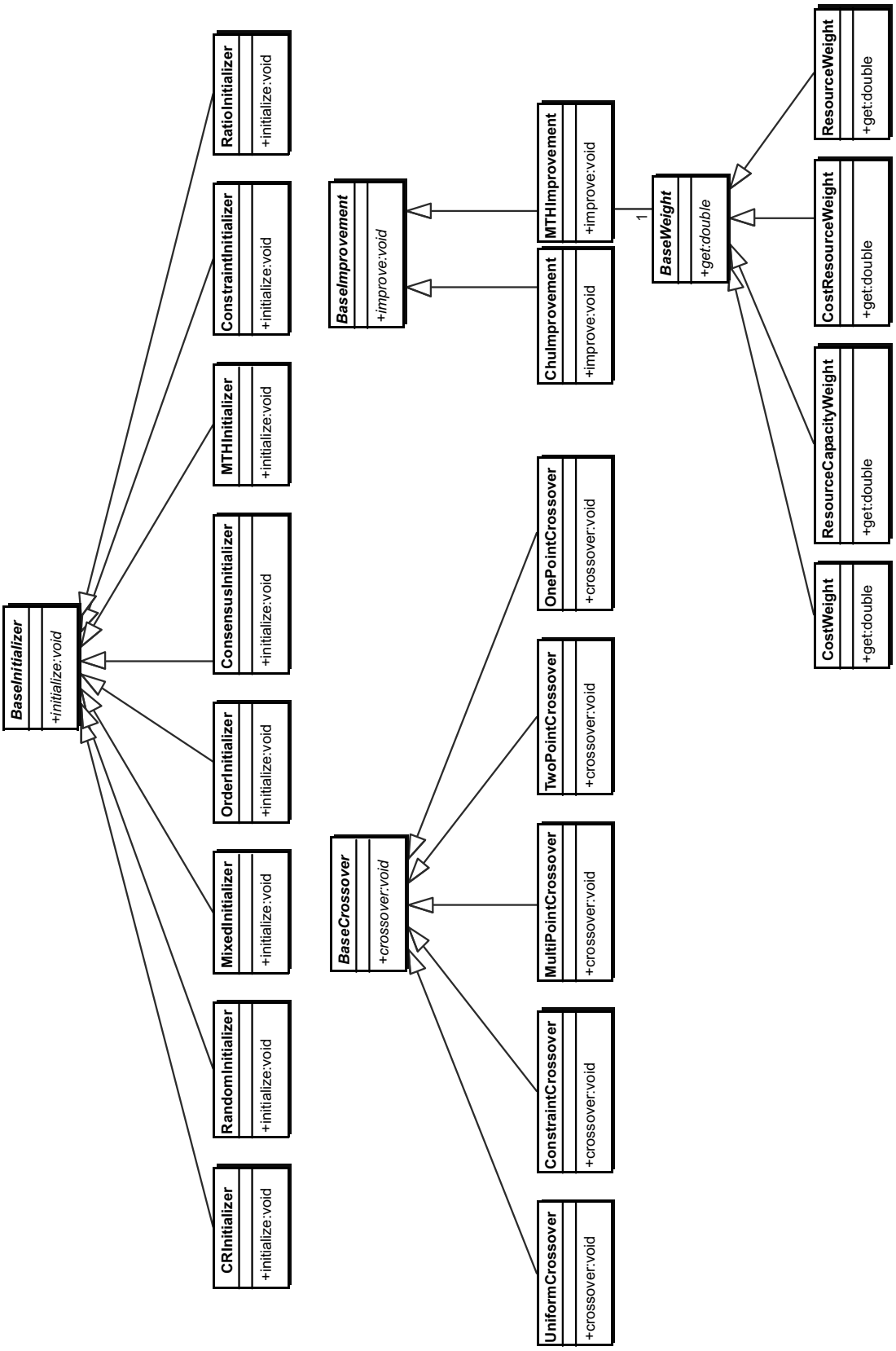

Abbildung 8.3: *Klassen Diagramm – Detail*

8.4 Benutzerdokumentation

In diesem Abschnitt wird der Programmaufruf mit allen möglichen Parametern erklärt und ein Aufrufbeispiel gegeben.

8.4.1 Programmaufruf

GAP:S 1.2.0 (Oct 12th 2001). Aufruf:
./gaps [options—-h—-v]

name:	default:	range:	description:
@	(stdin)		read parameters from specified file
crovm	(0)	[0,4]	which crossover operator to use
dupelim	(1)	[0,1]	use duplicate elimination?
evalm	(0)	[0,2]	evaluator mode
gapdir	(./)		directory of GAP data-file
gapname	(gap.dat)		filename of GAP data
impvm	(0)	[0,3]	improvement mode
initm	(0)	[0,7]	initialization mode
initpr	(0)	[0,1]	print initial solution?
lbuffer	(10)	[1,10000000]	number of log entries that are buffered
lchonly	(1)	[0,1]	log only, when best obj.val. changes
lfreq	(1)		frequency for writing log entries
logext	(.log)		extension for log file
maxi	(1)	[0,1]	should be maximized?
mth_div	(0)		division of weight factors for mth
mth_imp	(1)	[0,1]	improvement of mth?
mth_sz	(0)	[0,10000]	group size for mth initialization
muttm	(0)	[0,3]	mutation mode
nagents	(5)		number of agents
nformat	(%f)		format for writing double values
njobs	(5)		number of jobs
odir	()		directory for all output files
oname	(stdout)		base-name for all output files
outext	(.out)		extension of stdout file
outm	(0)		output mode
pcross	(1)	[0,1]	crossover probability
pmut	(-1)	[-100,100]	mutation probability
popsize	(100)	[2,10000000]	size of the population
repl	(1)	[0,1]	replacement scheme
seed	(0)		seed value of random number generator
tcond	(0)	[0,1]	termination criterion
tgen	(100000)	[0,100000000]	generations until termination
tselk	(2)	[1,10000]	group size for tournament selection
unfitm	(0)	[0,4]	unfitness mode
xpoint	(1)	[0,10000]	number of crossover points

8.4.2 Aufruf-Beispiel

Folgender Aufruf hat in der Regel die besten Ergebnisse erzielt:
./gaps crovm 1 evalm 0 impvm 1 initm 7 mth_div 50%50% muttm 2 pcross 1 \ pmut 2 tcond 1 tgen 500000 tselk 2 unfitm 0

crovm 1
One-Point Crossover.

evalm 0
Lack Evaluator. Dieser bewertet ungültige Individuen immer schlechter als gültige.

impvm 1
Improvement-Operator von Chu und Beasley. Im ersten Schritt wird versucht die Gültigkeit herzustellen, im zweiten die Performance (Kosten) zu verbessern.

initm 7
Falls eine gültige Lösung der beiden Initialisierungs-Methoden *constraint* und *ratio* existiert, wird diese genommen, ansonsten die bessere der beiden Lösungen. Die Constraint-Initialisierung berücksichtigt die Kapazitätsbeschränkungen der einzelnen Maschinen, die Ratio-Initialisierung wählt die Zuweisung mit dem besten Resourcen-Kosten Verhältnis.

mth_div 50%50%
Bei der Heuristik von Martello und Toth werden folgende Gewichtsfaktoren *cost* und *cost-resource* mit 50-prozentiger Wahrscheinlichkeit angewandt.

muttm 2
Mutation nach der Heuristik von Martello und Toth.

pcross 1
Der Parameter *pcross* gibt die Rekombinationswahrscheinlichkeit an; in diesem Fall: 100%.

pmut 2
Damit wird die Mutationswahrscheinlichkeit angegeben. Gibt die Anzahl der Mutationen pro Chromosome an.

tcond 1
Als Abbruchbedingung wurde *convergence* gewählt, d.h. es wurde in den letzten *tgen* Generationen kein besseres Individuum als das derzeit, beste Individuum gefunden.

tgen 500000
Über den Parameter *tgen* wird die Anzahl der zu berechnenden Generationen eingestellt. Dieser Wert ist das Abbruchkriterium des GA.

tselk 2
Mit dem Parameter *tselk* wird der Selektionsdruck bei der Tournament Selektion eingestellt.

unfitm 0
Für die Unfitness (Penalty) wird die Verletzung der Nebenbedingungen (Kapazitätsbeschränkungen) herangezogen.

popsize 100
Mit dem Parameter *popsize* wird die Populationsgröße für den Genetischen Algorithmus eingestellt.

Bei jedem Testlauf wird ein Protokoll inkl. der besten Lösung in Form einer Protokolldatei angelegt. Zunächst werden alle eingestellten Parameter ausgegeben. Weiters sind die wichtigsten Daten für die statistische Auswertung enthalten. Immer wenn ein neues, bestes Individuum in der Population auftritt, wird ein Eintrag generiert. Dieser enthält die aktuelle Generationsnummer, das Optimum, das Schlechteste, den Mittelwert, die Standardabweichung der Population, die Korrelation zum initialen Optimum sowie dem letzten Optimum und die Anzahl der gültigen Individuen innerhalb der Population (#_gen best worst mean dev corr_init corr_last #_valid). Nach der Terminierung des GAs werden weitere Daten wie die beste Lösung inkl. Generationsnummer, Rechen-Zeit und Fitnesswert, die Anzahl der Rekombinationen, Mutationen sowie die Anzahl der eliminierten Duplikate und Verbesserungen ausgegeben.

8.4.3 Parameter Dokumentation

Die Parameter-Dokumentation liegt ausschließlich in der Sprache Englisch vor, da sie direkt aus der von doxygen generierten Sourcecode-Dokumentation von GAP:S übernommen wurde.

- int_param ***crovm***

  Crossover mode to be used. Determines the mode to be used for the crossover operator.

  Values:
  - ***(0)*** **uniform crossover** random parts of parents are used to generate children
  - ***(1)*** **one point crossover** one part is taken from the first parent the rest is taken from the other
  - ***(2)*** **two point crossover** extension of one point crossover - based on two crossover-points
  - ***(3)*** **multipoint crossover** extension of one point crossover - supporting multiple crossover-points
  - ***(4)*** **constraint crossover** crossover takes capacity constraints into account

- bool_param ***dupelim***

  Duplicate elimination. If set, new solutions are checked if they are duplicates of existing solutions, in which case they are not included in the population. (EAlib)

o int_param ***evalm***

Evaluator mode to be used. Determines the mode to be used for evaluation.

Values:

- ***(0)*** **lack** invalid chromosomes are always replaced first according to infeasibility
- ***(1)*** **fitness** replacement takes place according to worst fitness; doesn't regard the infeasibility of the chromosome (introduced by Chu and Beasley)
- ***(2)*** **condition** takes the condition of the chromosomes into account

o int_param ***impvm***

Improvement mode to be used. Determines the mode to be used for improvement.

Values:

- ***(0)*** **off** no improvement used at all
- ***(1)*** **CB-improvement** based on the improvement of the feasibility followed by the improvement of the overall cost (introduced by Chu and Beasley)
- ***(2)*** **feasibility** improves feasibility (condition) of chromosome (1st part of CB-improvement)
- ***(3)*** **performance** improves cost of chromosome (2nd part of CB-improvement)

o int_param ***initm***

Initialization mode to be used. Determines the mode to be used for the initialization operator.

Values:

- ***(0)*** **random** assignment of jobs to agents is done completely random
- ***(1)*** **MTH-initialization** initialization is based on heuristic introduced by Martello and Toth
- ***(2)*** **constraint** makes assignment based on capacity constraints
- ***(3)*** **consensus** based on cost and resource efficient assignment
- ***(4)*** **ratio** best resource/cost ratio wins
- ***(5)*** **order** assignment based on order of cost-resource consumption
- ***(6)*** **mixed** takes best solution of constraint-, ratio- and order-initialization
- ***(7)*** **ctrt** takes best solution of constraint- and ratio-initialization

o bool_param ***initpr***

Output of initial solution. If set, outputs best and worst individual of first generation.

o bool_param ***maxi***

Should be maximized? True if maximization, false for minimization. (EAlib)

o string_param ***mth_div***

Division of weight factors for the heuristic of Martello and Toth. Different weight factors are supported by the MTH operator.

Values:

cost this takes the cost or profit into account (depends on the purpose)

cost/resource cost or profit relative to the resource consumption

resource resource usage involved

resource/capacity resource consumption relative to agents capacity

The format is ,,cost%cost/resource%resource%resource/capacity%".

o bool_param ***mth_imp***

Improvement mode for the heuristic of Martello and Toth. If set, improvement of MTH (2nd phase of heuristic) is performed.

o int_param ***mth_sz***

Group size used for MTH initialization. A group size equal to the number of jobs indicates one large group (the whole chromosome). To receive different solutions one has to apply a group-size which is smaller than the chromosome length. The group-size separates the chromosome into smaller parts, the MTH initialization is then performed for every part independently.

o int_param ***muttm***

Mutation mode to be used. Determines the mode to be used for the mutation operator.

Values:

(0) **random** the assignment of jobs to agents is done completely random

(1) **swap** the jobs between two agents are randomly swapped

(2) **MTH-mutation** the mutation is based on heuristic introduced by Martello and Toth

(3) **condition** the mutation is based on condition of agents

o int_param ***nagents***

Number of agents in GAP. Determined by the GAP-data file.

o int_param ***njobs***

Number of jobs in GAP. Determined by the GAP-data file.

o double_param ***pcross***

The crossover probability. Probability for generating a new chromosome by crossover. (EAlib)

o double_param ***pmut***

The mutation probability. Probability/rate of mutation within a new chromosome. If the value is negative, it is interpreted as an average value per chromosome instead of a fixed rate, and for each gene it is randomly decided if mutation takes place or not. (EAlib)

- **o** int_param ***popsize***

  The population size. The number of chromosomes the population contains. (EAlib)

- **o** int_param ***repl***

  Replacement scheme. (EAlib)

  Values:

  (0) **random** A new chromosome replaces a randomly chosen existing chromosome with the exception of the best solution.

  (1) **worst** A new chromosome replaces the worst existing solution. Duplicate eliminiation takes place according to parameter dupelim.

- **o** int_param ***tgen***

  The number of generations until termination. Interpreted according to tcond. (EAlib)

- **o** int_param ***tselk***

  Group size for tournament selection. Used in tournamentSelection(). (EAlib)

- **o** int_param ***tcond***

  The termination condition. Decides the strategy used as termination criterion. (EAlib)

  Values:

  (0) **generations** Terminate after tgen generations.

  (1) **convergence** Terminate after convergence, which is defined as: the objective value of the best solution in the population did not change within the last tgen generations.

- **o** int_param ***unfitm***

  Unfitness mode to be used. Determines the unfitness mode to be used on calculation of the fitness function.

  Values:

  (0) **violation** violation of capacity constraint

  (1) **violation/capacity** ratio of capacity constraint violation to capacity

  (2) **violation/load** ratio of capacity violation to assigned jobs

  (3) **violation/capacity/load** ratio of capacity violation to assigned jobs and capacity

  (4) **condition** unfitness is based on condition

- **o** int_param ***xpoint***

  Number of crossover points. Number of crossover points used for multipoint crossover. Multipoint crossover is only used for values equal to or higher than three otherwise the corresponding crossover operators are used directly.

Kapitel 9

Zusammenfassung

Es existieren zahlreiche Ansätze mit exakten sowie heuristischen Algorithmen für das GAP. Exakte Verfahren liefern zwar optimale Lösungen, sind jedoch zum Lösen großer Instanzen ungeeignet. Heuristische Methoden versuchen Näherungslösungen für große Instanzen innerhalb bestimmter Zeitlimits zu liefern, welche manchmal jedoch schlecht sind. Die Vergleiche mit den in den bisherigen Arbeiten veröffentlichten Algorithmen fallen zu gunsten des Genetischen Algorithmus in Kombination mit Heuristiken aus. Dieser hybride Ansatz soll die Lücke zwischen den exakten und primitiven heuristischen Verfahren schließen. Die Qualität des gewählten, hybriden Ansatzes ist besser, jedoch ist die Laufzeit mancher anderen heuristischen Verfahren geringer.

Am meisten profitierte der GA von den drei folgenden Heuristiken (wie bereits durch den Namen H3-GA angedeutet): der LP-Heuristik bzw. Constraint-Ratio-Heuristik bei der Initialisierung, des Heuristic Improvement Operators von Chu und Beasley bei der Reparatur sowie der Heuristik von Martello und Toth im Rahmen der Mutation.

Den größten Einfluß in Hinblick auf die Endlösungen hat dabei die LP-Initialisierung; diese führte zu sehr guten Ergebnissen. Generell kann gesagt werden, daß die (heuristische) Initialisierung einen großen Stellenwert hat. Diese soll sicherstellen, daß sich von Anfang an möglichst nur gültige Individuen in der Population befinden. Es ist von entscheidender Bedeutung, daß möglichst früh nur mehr gültige (und gute) Lösungen vorhanden sind. Im Optimalfall von Beginn an in Form der initialen Population, ansonsten im Rahmen der Ersetzung durch rasche Eliminierung der ungültigen Individuen. Das wirkt sich nicht nur auf die Laufzeit (Anzahl der Generationen) positiv aus, sondern gerade auch auf die Qualität der Endlösungen. Durch die Anwendung von Heuristiken bei den genetischen Operatoren soll der Suchraum und damit die Suchdauer verringert werden, da sich der GA nur auf ,,vielversprechende" Regionen konzentriert.

Genauso von Vorteil ist die Optimierung der initialen Population – gültige und gute Individuen tragen eher zu gültigen und guten Individuen bei. Wie bereits Chu und Beasley gezeigt haben, ist die Einbeziehung problemspezifischer Heuristiken von großem Nutzen. So trägt der heuristische Reparaturoperator entscheidend zu den guten Ergebnissen bei; die heuristische Mutation nach Martello und Toth spielt eine nebensächliche Rolle.

Bei den vorliegenden Instanzen waren aufgrund der gewählten Initialisierung meistens von Anfang an gültige Individuen in der Ausgangspopulation vorhanden, d.h. die Bewertungsfunktion entspricht der Zielfunktion. Alle drei Evaluierungsmethoden verhalten sich dann praktisch gleich. Ansonsten war die explizite Berücksichtigung der Fitness als auch der Untauglichkeit von Kandidatenlösungen gegenüber der reinen Fitness-Evaluierung im Vorteil. SAW-ing muß in Kombination mit der Condition Evaluation angewendet werden, um ungültige Individuen sukzessive aus der Population zu ,,drängen". In diesem Fall waren die erhaltenen Ergebnisse recht vielversprechend - auf gleichem Niveau wie bei der Lack Evaluation.

Für die Rekombination empfiehlt sich eine Strategie, die wenig destruktiv ist (a'la 1pt-Crossover). Bei dieser bleiben die Lösungsstrukturen eher erhalten.

Die gewählte Preprocessing-Strategie, in Form des Variablen-Reduktions-Schemas (VRS), brachte kaum Vorteile. Sowohl in Hinblick auf die Endergebnisse als auch betreffend der Laufzeit des GA's, d.h. gesamtdurchschnittlich war keine Zeitersparnis festzustellen; die Ergebnisse waren dafür im Mittel nur um 0,02 Prozentpunkte schlechter.

Mit Hilfe des Programmpakets CPLEX wurden die IP-Optima bzw. die LP-Optima (fungieren als untere Schranke für die IP-Lösungen) bestimmt. Andererseits standen die erhaltenen Ergebnisse von CPLEX in direkter Konkurrenz zu unseren eigenen Ansätzen. Für kleine bzw. einfache Instanzen lieferte CPLEX in kurzer Zeit sehr gute Ergebnisse. Allerdings lieferte es für große bzw. komplexe Probleme eindeutig schlechtere Ergebnisse (aufgrund der natürlichen Grenzen des Branch-and-Bound Ansatzes) als unser LP-basierender Ansatz.

Der GA kann mit einem Branch-and-Bound Verfahren kombiniert werden, um das exakte Optimum zu bestimmen. Die beste gefundene Lösung des GA dient dabei als Ausgangslösung (Supremum) für das Branch-and-Bound Verfahren. Die obere Schranke dient dazu, den Verzweigungsbaum frühzeitig abzubrechen, sobald die beste erzielbare Lösung eines Teilbaums schlechter als diese Schranke ist.

Kapitel 10

Anhang

10.1 Lineare Programmierung LP

10.1.1 Constraint Programming

Das zu lösende Problem wird definiert als Menge von Unbekannten und deren Wertebereiche, sowie einer Menge von Nebenbedingungen (Restriktionen). Constraint Programming sucht nach einer Belegung der Unbekannten innerhalb der Wertebereiche, welche die gegebenen Bedingungen erfüllt und – bei Optimierungsproblemen – eine zusätzlich gegebene Funktion minimiert bzw. maximiert. Kennzeichnend für das Constraint Programming ist die direkte Weiterleitung von Implikationen von Einschränkungen des Wertebereichs einer Unbekannten auf die Wertebereiche der anderen Unbekannten.

10.1.2 Kombinatorische Optimierungsaufgabe

Kombinatorische Optimierung (*combinatorial optimization*) befaßt sich mit mathematischen Optimierungsaufgaben, die eine endliche Menge von Lösungen haben. Die allgemeine Form für ein solches Problem lautet:

Gegeben sei eine endliche Menge I von $\Im$ (die Menge aller gültigen Lösungen) und eine Funktion $f : \Im \to \mathbb{R}$, finde ein Element $I^* \in \Im$ mit

$$f(I^*) = \max\{f(I) \mid I \in \Im\}.$$

Lineare Programmierung LP (*linear programming*) ist eines der Grundmodelle der mathematischen Optimierung. Hierbei gilt es lineare Probleme zu optimieren; die Lösung wird als Vektor dargestellt, wobei die Bedingungen an den Lösungsvektor sowie die Funktion, die es zu minimieren bzw. zu maximieren gilt, linear sind.
Für Lineare Probleme LP sind die Nebenbedingungen und die Zielfunktion linear. Das Optimum liegt immer in einer Ecke (oder entlang einer Kante) des zulässigen Bereichs. Mit Berücksichtigung einer weiteren Nebenbedingung wird immer nur ein Stück vom zulässigen Lösungsbereichs mit einem geraden Schnitt abgetrennt. Dadurch können nur konvexe Lösungsmengen entstehen. Daraus folgt, daß die optimale Lösung immer am Rand bzw. in einer Ecke liegen muß und niemals im Inneren des zulässigen Lösungsbereichs.

10.1.3 Lineare Optimierungsaufgabe LP

Gegeben sei eine Matrix $A \in \mathbb{R}^{(m,n)}$ und Vektoren $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$,

$$x := \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \quad c := \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \quad b := \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} \quad A := \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$$

finde einen Vektor $x^* \in \mathbb{R}^n$, mit

$$c^T x^* = \max\{c^T x \mid Ax \leq b\}.$$

Die Funktion $c^T x : \mathbb{R}^n \to \mathbb{R}$ wird Zielfunktion z (*objective function*) genannt. Die Ungleichungen im System $Ax \leq b$ werden Restriktionen (*constraints*, Nebenbedingungen) genannt. Eine Lineare Optimierungsaufgabe (*linear optimization problem*) wird auch Lineare Programmieraufgabe (*linear programming problem*) oder kurz Lineares Programm (*linear program*, LP) genannt.

Ein lineares Minimierungsproblem kann durch Multiplikation der Zielfunktion z mit dem Wert -1 in ein lineares Maximierungsproblem umgewandelt werden; deshalb ist es hinreichend Maximierungsaufgaben zu betrachten.

In vielen Fällen wird gefordert, daß einige oder alle Variablen ganzzahlig sein müssen. Durch Hinzufügen von Ganzzahligkeitsbedingungen zu einer Menge der Variablen, erhält man ein ,,*linear mixed integer optimization problem*" (MILP).

10.1.4 Linear mixed integer optimization problem

Gegeben sei eine Matrix $A \in \mathbb{R}^{(m,n)}$, Vektoren $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$ und eine Teilmenge $I \subseteq \{1, \ldots, n\}$, finde einen Vektor $x^* \in \mathbb{R}^n$ mit x_i^* ganzzahlig für alle $i \in I$ und

$$c^T x^* = \max\{c^T x \mid Ax \leq b,\ x_i \text{ ganzzahlig } \forall\ i \in I\}.$$

Wenn alle Variablen ganzzahlig sein müssen, d.h. $I = \{1, \ldots, n\}$, so wird aus dem ,,linear mixed integer optimization problem" ein ,,*linear integer optimization problem*" (ILP). Wenn die Variablen aus der Menge I die Werte 0 oder 1 annehmen müssen, so wird diese Optimierungsaufgabe ,,*linear (mixed) zero-one optimization problem*" genannt.

10.1.5 Relaxation

Es sei gegeben, die Menge $F = \{x \mid Ax \leq b,\ x_i \text{ ganzzahlig } \forall i \in I\}$ der gültigen Lösungen eines gemischt-ganzzahligen Programms $\max\{c^T x \mid x \in F\}$.

Eine Maximierungsaufgabe $\max\{r(x) \mid x \in R\}$ ist eine Relaxation[1] einer gemischt-ganzzahligen Optimierungsaufgabe, wenn $F \subseteq R$ und $c^T x \leq r(x)\ \forall x \in F$.

[1] In der Mathematik wird mit Relaxation ein Verfahren zur näherungsweisen Lösung von Gleichungen bezeichnet [lat. *relaxatio* ,,Entspannung"].

10.1.6 Ausgewählte Optimierungsaufgaben

10.1.6.1 0-1 Rucksack Problem

Das einfache 0/1-Rucksack-Problem (*Single 0/1 Knapsack Problem*) wird wie folgt charakterisiert: Gegeben sei eine Menge von Gegenständen $S = \{1, ..., n\}$, wobei jeder Gegenstand i eine bestimmtes Gewicht w_i und Profit p_i hat. Weiters sei genau ein Rucksack mit Kapazität c gegeben.
Ziel ist es, den Rucksack mit größtmöglichem Profit zu füllen, d.h. gesucht ist die Untermenge $S' \subset S$, die den Wert $\sum_{i \in S'} p_i$ maximiert.

$$\begin{array}{ll} \text{Maximiere} & \sum_{i=1}^{n} p_i \cdot x_i \\ \text{sodaß} & \sum_{i=1}^{n} w_i \cdot x_i \leq c \\ & x_i \in \{0, 1\} \end{array}$$

Man spricht hier von einem 0/1-Rucksackproblem, da die Gegenstände nicht geteilt werden dürfen (der Gegenstand nimmt entweder zur Gänze teil oder gar nicht).

Das allgemeine 0/1-Rucksack-Problem (*Multiple 0/1 Knapsack Problem*) ist die Verallgemeinerung des einfachen Rucksackproblems. Der Unterschied besteht darin, daß mehrere unabhängige Rucksäcke mit verschiedenen Kapazitäten vorhanden sind. Weiters hat jeder Gegenstand ein unterschiedliches Gewicht für jeden Rucksack.

10.2 Komplexität

Die Komplexität wird als Funktion der von einem Algorithmus zu verarbeitenden Datenmenge (N) und eventuell weiterer spezifischer Konstanten dargestellt. Dann betrachtet man das Verhalten des Aufwandes bei immer größer werdenden N, mit anderen Worten: man vergleicht den asymptotischen Verlauf der Funktionen. Von Interesse ist zumeist der schlechteste Fall (*worst-case*), der als obere Schranke fungiert als auch der Durchschnitt (*average-case*) – bei unseren Betrachtungen wird nur das Worst-Case Szenarium untersucht. Neben dem Zeitverhalten eines Algorithmus kann auch der Speicherverbrauch von Interesse sein.

Der Aufwand eines Algorithmus wird mit Hilfe der *O-Notation* beschrieben. Die O-Notation trifft Aussagen über die Ordnung von Algorithmen, es wird die asymptotische Aufwandsentwicklung bei (theoretisch unendlich) hohen Datenmengen betrachtet. Eine wesentliche Eigenschaft der O-Notation besteht nun darin, daß man die Abschätzung durch Weglassen der niederwertigeren Teile stark vereinfachen kann. In einer O-Notation zählt immer nur der aufwendigste Teil der Abschätzung. Man schreibt, das Problem hat eine Komplexität der Ordnung $f(n) = O(g(n))$, wenn eine Konstante $c > 0$ existiert, so daß für genügend große n gilt: $f(n) \leq c \cdot g(n)$.

- Polynomiale Algorithmen sind durch $O(n^K)$ beschränkt, K = Konstante. Auch $O(n^K \cdot \log_2(n))$ wird als polynomial bezeichnet.
- Exponentielle Komplexität liegt für Schranken wie $O(k^n)$, $O(n^{\mathrm{ld}(n)})$, $O(n!)$ vor.

Die Tabelle soll die Unterschiede der Komplexität verdeutlichen:

Komplexität des Algorithmus	Zeitmultiplikator		
n	$1 \cdot 10^1$	$1 \cdot 10^2$	$1 \cdot 10^3$
$n \cdot \mathrm{ld}(n)$	$3{,}3 \cdot 10^1$	$6{,}64 \cdot 10^2$	$9{,}966 \cdot 10^3$
n^3	$1 \cdot 10^3$	$1 \cdot 10^6$	$1 \cdot 10^9$
$10^6 \cdot n^8$	$1 \cdot 10^{14}$	$1 \cdot 10^{22}$	$1 \cdot 10^{30}$
$n^{\mathrm{ld}(n)}$	$2{,}099 \cdot 10^3$	$1{,}93 \cdot 10^{13}$	$7{,}89 \cdot 10^{29}$
2^n	$1{,}024 \cdot 10^3$	$1{,}27 \cdot 10^{30}$	$1{,}05 \cdot 10^{301}$
$n!$	$3{,}6288 \cdot 10^6$	$1 \cdot 10^{158}$	$4 \cdot 10^{2567}$

10.2.1 NP-Vollständigkeit

Die Theorie der NP-Vollständigkeit beschäftigt sich mit den Problemen, für die kein effizienter (polynomieller) Algorithmus existiert. Dafür werden zwei Klassen definiert:

- **P:** Menge aller Probleme, die mit Hilfe deterministischer Algorithmen in polynomieller Zeit (Polynomialzeit-Algorithmus[2]) gelöst werden können.
- **NP:** Menge aller Probleme, die mit Hilfe nichtdeterministischer Algorithmen in polynomieller Zeit gelöst werden können (*non-deterministic polynomial*).

Die Fähigkeit des Nichtdeterminismus ist die unerfüllbare Annahme, daß bei der Wahl zwischen verschiedenen Varianten der Algorithmus die richtige ,,erät".

Bis heute ist nicht bewiesen worden, daß $P \neq NP$ ist.

Ein Problem, das *NP* angehört, muß als Entscheidungsproblem formuliert werden können, das mit ,,Ja" oder ,,Nein" beantwortet werden kann. Eine Instanz x des Problems muß in polynomialer Zeit verifizierbar sein, d.h. man kann mit polynomialem Aufwand feststellen, ob das Ergebnis des Algorithmus das Problem löst.

NP-vollständige Probleme (*NP-complete, NPC*) sind definiert als solche Probleme, die (a) aus *NP* sind und (b) in die sich alle anderen Probleme aus *NP* in polynomieller Zeit transformieren lassen. Für einen Beweis der NP-Vollständigkeit muß das Problem Π als Ja/Nein-Entscheidungsproblem formuliert werden. Eine polynomielle Prozedur muß entscheiden können, ob spezielle Werte der Variablen das Problem lösen (dann gehört es NP an). Anschließend muß gezeigt werden, daß sich ein bekanntes NP-vollständiges Problem polynomiell in das Problem Π transformieren läßt. Dann lassen sich gemäß Definition nämlich alle NP-vollständigen Probleme dorthin transformieren.

Daneben bezeichnet man als ,,NP-schwierig" (*NP-hard*) die Probleme, auf die sich zwar alle Probleme aus NP polynomiell reduzieren lassen, deren Zugehörigkeit zu NP aber nicht gezeigt werden kann (es sind keine Polynomialzeit-Algorithmen bekannt). Eine solche Optimierungsaufgabe ist angewiesen auf die Lösung eines NP-vollständigen Problems.

[2] Algorithmus, dessen Laufzeit durch ein Polynom in der Eingabegröße beschränkt werden kann.

10.3 Algorithmen Design

10.3.1 Divide-and-Conquer

Ein *Divide-and-Conquer* (Teilen und Herrschen) Algorithmus teilt das Problem in mehrere gleichartige, aber kleinere Teilaufgaben auf (divide). Diese Teilaufgaben werden irrerseits ebenfalls aufgeteilt (iterative/rekursive Zerlegung des Problems), solange bis diese klein genug sind, d.h. eine effiziente Lösung der Teilaufgabe möglich ist. Im letzten Schritt werden die so gelösten Teilaufgaben zu einer Gesamtlösung zusammengeführt (conquer).

10.3.2 Branch-and-Bound

Die Anzahl der Alternativen bei kombinatorischen Optimierungsproblemen ist zumeist riesig und wächst exponentiell mit zunehmendem Umfang der Aufgabe. Das vollständige Enumerieren (Aufzählen) kommt dann nicht mehr in Frage.
In vielen Bereichen ist *Branch-and-Bound* (auch *implicit enumeration*) daher zur Standardtechnik geworden, etwa in der Logistik bei der Transport-, Touren- und Standortplanung. Andere wichtige Anwendungsfelder sind Produktions-, Projekt- und Investitionsplanung.

Branch-and-Bound basiert auf dem *Divide-and-Conquer* Ansatz, bei dem das Originalproblem in kleinere Teilprobleme aufgeteilt wird; für diese werden untere und obere Schranken berechnet. Die obere Schranke (Maximierungsaufgabe) wird durch die Technik der Relaxation berechnet (siehe Kapitel 10.1.5). Eine Lösung des, durch Relaxation, aufgelockerten Problems stellt somit eine obere Schranke für den optimalen Zielfunktionswert des Ausgangsproblems dar. Die untere Schranke erhält man durch Auffinden einer gültigen Lösung der Gesamtaufgabe.

Die beiden Hauptbestandteile von Branch-and-Bound sind (siehe [8], [9]):

- **Branching** (Verzweigen):
  Die zu lösende Aufgabe wird in zwei oder mehr Teilaufgaben zerlegt. Dasselbe geschieht wiederum mit jeder entstehenden Teilaufgabe, so daß sich insgesamt eine Baumstruktur von (Teil-)Aufgaben ergibt.

- **Bounding** (Beschränken):
  Wenn man sicher sein kann, daß die optimale Lösung einer Teilaufgabe nicht besser als eine schon bekannte Lösung der Gesamtaufgabe ist, braucht diese Teilaufgabe nicht weiter betrachtet zu werden. Man spricht davon, daß sie ausgelotet ist. Um dies festzustellen, schätzt man den höchsten erreichbaren Gewinn für die betrachtete Teilaufgabe ab. Ein solcher Schätzwert heißt obere Schranke.

10.3.3 Branch-and-Cut

Branch-and-Cut stellt eine Spezialisierung dieser Branch-and-Bound Technik dar. Hierbei werden bereits während dem Aufbau des Entscheidungsbaums gültige

Schranken eingeführt. Dabei wird auf die Technik der *cutting planes* (Schnittflächen) zurückgegriffen, bei der fortlaufend Teile eines *Polytops* im mehrdimensionalen Raum abgetrennt werden, sofern sie für die Lösung offensichtlich nicht mehr benötigt werden. Durch diese Maßnahme wird die Effizienz von Branch-and-Bound wesentlich gesteigert.

10.3.4 Column Generation

Um Lineare Programme zu verkleinern, werden beim *Column-Generation* Ansatz anfangs nur sehr wenige Variablen berücksichtigt und dann nach und nach vielversprechende Variablen ins Lineare Programm hinzugefügt, bis man eine sehr gute oder optimale Lösung gefunden hat. Dabei ist typischerweise die Anzahl der eingefügten Variablen deutlich geringer als die Gesamtzahl der ursprünglichen Variablen. Technisch entspricht die Hinzunahme von Variablen der Generierung und dem Hinzufügen von Matrixspalten, daher der Name Column Generation.

10.4 Grundlagen

10.4.1 Set Partition

Eine *Set Partition* (Mengenpartitionierung) einer Menge A ist eine Sammlung nichtleerer Teilmengen von A sodaß jedes Element aus A in genau einer Teilmenge vorkommt. Jede Teilmenge wird *Block* einer Set Partition genannt.
Es sei gegeben die Menge $A = \{1, 2, \ldots, 15\}$; weiters sei gegeben die Menge α, deren Elemente eine Teilmenge der Menge A sind.

$$\alpha = \{\{1\}, \{2\}, \{9\}, \{3, 5\}, \{4, 7\}, \{6, 8, 10, 15\}, \{11, 12, 13, 14\}\}$$

Jede Sammlung von Teilmengen der Menge A, die folgende drei Punkte erfüllen, heißt dann Set Partition der Menge A:

1. Jedes Element der Menge α ist nicht leer.
2. Die Vereinigung aller Elemente der Menge α ergibt die Menge A.
3. Zwei Mengen $X \in \alpha$ und $Y \in \alpha$ haben folgende Relation, entweder gilt $X = Y$ oder $X \cap Y = 0$ (disjunkt).

10.4.2 Heuristik

Ein Prozeß, welcher ein gegebenes Problem lösen kann, aber keine Lösung garantiert, wird als Heuristik bezeichnet. Die Heuristik dient dazu, Wesentliches und Unwesentliches zu unterscheiden, relevante Suchpfade einzuschlagen und weniger relevante Pfade auszulassen. Die Heuristik wird dazu verwendet, den verbleibenden Weg zur eigentlichen Zielposition abzuschätzen. Ein heuristischer Algorithmus hat allerdings nichts mit Zufall oder Nichtdeterminismus zu tun. Er terminiert genau dann wenn

dieser ein Resultat erreicht. In einigen Fällen gibt es keine Garantie wie lange ein Algorithmus benötigt um ein Ergebnis zu liefern, und in manchen Fällen ist die Qualität der Lösung nicht garantiert.

10.4.3 Polytop

Ein Polytop ist die konvexe Hülle endlich vieler Punkte im $\mathbb{R}^n$. Typisch für ein Polytop ist, dass es eine endliche Anzahl von Seitenflächen hat; so entsteht es z.B. aus einem Stück Holz durch mehrfaches Absägen, bis das Holzstück am Schluss nur noch durch die Sägeflächen begrenzt wird. Außer den Seitenflächen hat ein Polytop noch Ecken und Kanten. Beispiele dafür sind ein Würfel oder eine Pyramide.

10.4.4 Integrality Gap

Das *integrality gap* (**IG**) ("ganzzahliger Abstand") ist der maximale Abstand (obere Schranke) der optimalen, ganzzahligen Lösung zu der optimalen, linearen Lösung und ist definiert als maximale Relation (*worst-case ratio*) von $\frac{|OPT(LPR)|}{|OPT(ILP)|}$, wobei das LPR die Relaxation des ILP ist und das ILP das ganzzahlige, lineare Problem.

D.h. die ganzzahlige Lösung liegt im besten Fall nicht weiter weg von der optimalen, linearen Lösung als durch das integrality gap gegeben (Supremum). Wenn die Lösung des LPR den exakt, gleichen Wert hat wie die Lösung des ILP, so ist das IG genau Null.

Anders ausgedrückt, mißt das IG die Qualität der Relaxation, d.h. den Fehler, der durch die Relaxation der Ganzzahligkeitsbedingung einhergeht.

10.5 ILOG CPLEX 8.0

Die Erläuterungen über CPLEX 8.0 enstammen aus der Originaldokumentation und liegen deshalb nur in englischer Sprache vor. Weitere Informationen finden sich unter [7].

ILOG CPLEX is a tool for solving linear optimization problems, commonly referred to as Linear Programming (LP) problems, of the form:

$$
\begin{array}{lllllllllll}
\text{Maximize (or Minimize)} & c_1 \cdot x_1 & + & c_2 \cdot x_2 & + & \ldots & + & c_n \cdot x_n & & \\
\text{subject to} & a_{11} \cdot x_1 & + & a_{12} \cdot x_2 & + & \ldots & + & a_{1n} \cdot x_n & \sim & b_1 \\
 & a_{21} \cdot x_1 & + & a_{22} \cdot x_2 & + & \ldots & + & a_{2n} \cdot x_n & \sim & b_2 \\
 & \ldots & & & & & & & & \\
 & a_{m1} \cdot x_1 & + & a_{m2} \cdot x_2 & + & \ldots & + & a_{mn} \cdot x_n & \sim & b_m \\
\text{with these bounds} & l_1 \leq x_1 \leq u_1 & & & & & & & & \\
 & \ldots & & & & & & & & \\
 & l_n \leq x_n \leq u_n & & & & & & & &
\end{array}
$$

where $\sim$ can be $\leq$, $\geq$, or $=$, and the upper bounds u_i and lower bounds l_i may be positive infinity, negative infinity, or any real number.

The elements of data you provide as input for this LP are:

Objective function coefficients	$c_1, c_2, \ldots, c_n$
Constraint coefficients	$a_{11}, a_{21}, \ldots, a_{n1}$
	$\ldots$
	$a_{m1}, a_{m2}, \ldots, a_{mn}$
Right-hand sides	$b_1, b_2, \ldots, b_m$
Upper and lower bounds	$u_1, u_2, \ldots, u_n$ and $l_1, l_2, \ldots, l_n$

The optimal solution that CPLEX computes and returns is:

Variables	$x_1, x_2, \ldots, x_n$

CPLEX also can solve several extensions to LP:

- Network Flow problems, a special case of LP that CPLEX can solve much faster by exploiting the problem structure.
- Quadratic Programming (QP) problems, where the LP objective function is expanded to include quadratic terms.
- Mixed Integer Programming (MIP) problems, where any or all of the LP or QP variables are further restricted to take integer values in the optimal solution (and where MIP itself is extended to include constructs like Special Ordered Sets (SOS) and semi-continuous variables).

Literaturverzeichnis

[1] M. Amini, M. Racer. *A rigorous computational comparison of alternative solution methods for the generalized assignment problem.* Management Science, 40:868–890, 1994.

[2] J. E. Beasley. *OR-Library.*
http://mscmga.ms.ic.ac.uk/info.html
Imperial College of Science, Technology and Medicine, London

[3] D. Cattrysse. *Set partitioning approaches to combinatorial optimization problems.* PhD thesis, Katholieke Universiteit Leuven, Centrum Industrieel Beleid, Belgium, 1990.

[4] L. Chalmet und L. Gelders. *Lagrangean Relaxations for a Generalized Assignment - Type Problem.* North-Holland, Amsterdam, 1977.

[5] P. C. H. Chu und J. E. Beasley. *A genetic algorithm for the generalised assignment problem.* Aus Computers & Operations Research 24, Seiten 17–23, 1997.

[6] V. Chvátal. *Linear Programming.* W. H. Freeman, New York, 1983.

[7] ILOG. *ILOG CPLEX 8.0.* http://www.ilog.com/

[8] W. Domschke und A. Drexl. *Einführung in Operations Research.* Springer, 3. Auflage, Berlin, 1995.

[9] W. Domschke, A. Drexl, B. Schildt, A. Scholl und S. Voß. *Übungsbuch Operations Research.* Springer, 2. Auflage, Berlin, 1997.

[10] A.E. Eiben und J.I. van Hemert. *SAW-ing EAs: adapting the fitness function for solving constrained problems.*

[11] FAQ: comp.ai.genetic. *A Guide to Frequently Asked Questions.*
http://www.cs.cmu.edu/Groups/AI/html/faqs/ai/genetic/part2/faq-doc-2.html

[12] M. R. Garey und D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W.H. Freeman, San Francisco, 1979.

[13] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, MA, 1989.

[14] J. H. Holland. *Adaption in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, 1975

[15] K. Jörnsten und M. Nasberg. *A new Lagrangian relaxation approach to the knapsack problem.* European J. Oper. Res. 27, Seiten 313–323.

[16] S. Martello und P. Toth. *An algorithm for the generalized assignment problem.* Aus J.P. Brans, editor, Operational Research '81, Seiten 589–603. North-Holland, Amsterdam, 1981.

[17] S. Martello und P. Toth. *Algorithm for the Solution of the 0-1 Single Knapsack Problem.* Aus Computing 21 (1978), Seiten 81–86.

[18] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer, New York, 1992.

[19] I. H. Osman. *Heuristics for the generalised assignment problem:simulated annealing and tabu search approaches.* OR Spektrum, 17:211–225, 1995.

[20] W. H. Press, S. A. Teukolsky, W. T. Vetterling und B. P. Flannery. *Numerical Recipes in C : The Art of Scientific Computing* Cambridge University Press, 1992.

[21] G. Raidl. *Evolutionäre Algorithmen.* Skriptum zur Vorlesung Evolutionäre Algorithmen, Institut für Computergrafik und Algorithmen, TU-Wien, 1996.

[22] G. Raidl. *Eine C++ Bibliothek für Evolutionäre Algorithmen.* Institut für Computergrafik und Algorithmen, TU-Wien, 2000.

[23] S. Reith, H. Vollmer. *Komplexitätstheorie: P=NP beweisbar schwierig.* Aus c't - Magazin für Computertechnik, Heft 7/2001, Seiten 240–251, Hannover, 2001.

[24] G.T. Ross, R. M. Soland. *A branch and bound algorithm for the generalized assignment problem.* Mathematical Programming, 8:91–103, 1975.

[25] M. W. P. Savelsbergh. *A Branch-And-Cut Algorithm for the Generalized Assignment Problem.* Aus Operations Research 45: 6, Seiten 831–841. North-Holland, Amsterdam, 1997.

[26] A. Scholl, G. Krispin, R. Klein und W. Domschke. *Besser beschränkt: Clever optimieren mit Branch und Bound.* Aus c't - Magazin für Computertechnik, Heft 10/1997, Seiten 336–345, Hannover, 1997.

[27] The WWW Virtual Library. *Random Numbers and Monte Carlo Methods.* http://random.mat.sbg.ac.at/links/index.html

[28] G. Wöhe. *Einführung in die Allgemeine Betriebswirtschaftslehre.* Aus Vahlens Übungsbücher der Wirtschafts- und Sozialwissenschaften (1984), Seite 444, München.

[29] M. Yagiura, T. Ibaraki, F. Glover. *An Ejection Chain Approach for the Generalized Assignment Problem.* INFORMS Journal on Computing (to appear).

Index