ZUSCHNITTPROBLEME UND DER GENETISCHE ALGORITHMUS

T. Hürlimann

Working Paper No. 204

September 1992

INSTITUTE FOR AUTOMATION AND OPERATIONS RESEARCH

University of Fribourg

CH-1700 Fribourg / Switzerland

Bitnet: HURLIMANN@CFRUNI51

phone: (41) 37 21 95 60 fax: 37 21 96 70

Zuschnittprobleme
und der genetische Algorithmus

Tony Hürlimann, Dr. rer. pol.

Key-words: packing, knapsack, lay-out, cutting-stock, genetic algorithm

Abstract: This paper give an overview of different cutting-stock problems. One of them, the 2-dimensional orthogonal packing problem, will be exposed more deeply. It is well known that all variants of the cutting-stock problem are (at least) NP-complete. It is, therefore, improbable that practical algorithms will be found to solve them optimally. Especially, different Greedy-methods will be introduced, together with their worst-case performance ratio, which solve the problem appoximatively. Furthermore, an brief introduction will be given into the subject of genetic algorithms and their application to the two-dimensional packing problem.

Stichworte: Packungs-, Knapsack-, Lay-Out- und Zuschnittprobleme, genetische Algorithmen.

Zusammenfassung: In diesem Paper soll einen Einblick in die Vielfältigkeit der Zuschnittprobleme und ihrer Anwendung gegeben werden. Vor allem soll eine vertiefte Einblick in das zwei-dimensionale, orthogonale Packungsproblem und seiner heuristischen Lösungsmethoden vermittelt werden. Da dieses Problem NP-hart ist, werden anhand dieses Problems verschiedene, spezielle Heuristiken wie Next-Fit, First-Fit, Best-Fit usw., welche auch auf das eindimensionale Bin-Packing-Problem anwendbar sind, vorgestellt. Ferner soll eine informelle Einführung in das Gebiet der genetischen Algorithmen gegeben und deren Anwendung auf das 2-dimensionale Streifenzuschnittproblem erörtert werden.

Das Paper ist entstanden als ein Arbeitspapier für ein Informatikprojekt in Informatik des zweiten Jahres (Informatik II), im Rahmen des Informatikstudiums mit Lizentiatsabschluss in vier Jahren. Ziel des Projektes ist es ein Gefühl für schwierige Probleme und deren Lösung auf dem Computer zu vermitteln, sowohl ein Einblick in

das faszinierende Gebiet der genetischen Algorithmen zu geben.

1. EINLEITUNG

"To find a better strategy try variations
on what has worked well in the past".
(Davis)
"However many ways there may be of being alive,
it is certain that there are vastly more ways of being dead."
(Dawkins)
"To err is human. To really foul up, use a computer".
Grafitti, cited by Goldberg)

Auf einem Streifen gegebener Breite B sollen kleinere Rechtecke der Breite b und der Höhe h so platziert werden, dass die maximale Höhe H des verbrauchten Streifens minimal wird. Das Drehen der Rechtecke um 90 Grad ist dabei nicht erlaubt.
Beispiel (aus Coffman 1980a):

Breite: 20, Rechtecke $\{(b_i,h_i)\}$ = { (7,9), (6,5), (8,4), (5,4), (4,2) }

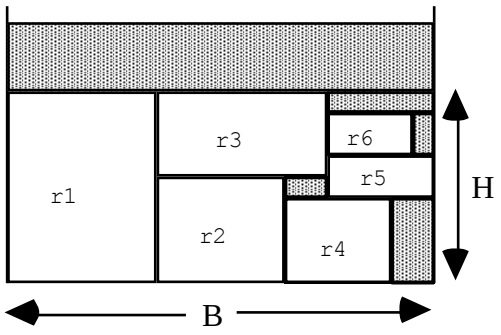


Figure 1-1

Figure 1-1 gibt die Platzierung der sechs Rechtecke im Streifen B bei minimaler Höhe an. Die Höhe H ist 9 bei der Breite B von 20. Da die Gesamtfläche der sechs Rechtecke 153 Einheiten ist, ist die ausgenutzte Fläche 85% (153/180). Der Rest wird als Verschnitt (Abfall) bezeichnet. Die minimal erreichbare Höhe H ergibt sich durch die ganzzahlige Division der Gesamtfläche durch die Breite (153/20 = 8). Diese Höhe kann in unserm Beispiel allerdings nicht erreicht werden. Die in Figure 1-1 angegebene Platzierung ist optimal in dem Sinne, dass die Höhe H=9 nicht mehr weiter verkleinert werden kann.

Dieses Problem gehört zu der umfangreichen Klasse der Zuschnittprobleme. Man spricht von einem Zuschnittproblem immer dann, wenn es um die effektive Anordnung ebener Figuren in einem vorgegebenem oder vorzugebendem Gebiet oder

Gebieten geht. Genauer: Unter einem Zuschnittproblem soll folgendes Problem verstanden werden: "Aus einer Menge von ebenem Figuren ist eine Teilmenge auszuwählen, die durchschnittsfremd in einem der Struktur nach oder fest vorgegebenen Gebiet anzuordnen ist, so dass eine optimale Fächenauslastung erreicht wird." (Terno S.9).
Eine Taxonomie von Zuschnittproblemen wird in Dyckoff (1990) vorgeschlagen.

Es existiert eine Vielzahl von Varianten der Zuschnittprobleme, welche in der Praxis eingesetzt werden (können). Bei Fertigungsvorgängen, wo aufgrund der natürlichen Gestalt oder gegebender Produktionsbedingungen das Material aus fixer, geometrischer Form vorliegt und in kleinere Einheiten zerlegt werden muss, entstehen Zerschneideprozesse. Beispiele davon sind: Zuschnitt von Blechen, Guillotine-Zuschnitt in der Glasindustrie, Zusammenstellung von Schnittbildern in der Textil- und Schuhindustrie, Zerschneiden von Stangenmaterial, Brettern und Platten oder Zuschnitt von Graphitblocken aus einem grösseren (3-dimensionalen) Block. Auch Platzierungs-, Anordnungs- , Beladeprobleme von Kontainern und Layoutprobleme von Leiterplatten oder Werkhallen, sowie plannungstechnische Probleme wie der zeitliche Einsatz von beschränkten Resourcen können dieser Klasse untergeordnet werden. Erwähnt werden sollte auch, dass Zuschnitt- und Packungs- oder Platzierungsprobleme zueinander 'dual' sich: Werden bei den Zuschnittproblemen kleinere Formen aus grösseren ausgeschnitten, so werden bei Packungsproblemen kleinere Formen auf grösseren platziert, die beiden Probleme sind identisch. (Beispiel: Formen aus dem Teig werden ausgeschnitten, Ausstechformen werden auf dem Teig platziert!).

Um einen Einblick in die Fülle der Probleme zu geben, sollen einige konkrete Probleme vorgestellt und deren Formulierung präzisiert werden. Das faszinierende an den Zuschnittproblemen ist der Gegensatz zwischen der unmittelbare Anschaulichkeit einerseits und der komplexen, internen Struktur des Problems andererseits. Es kommt dazu, dass ein Zuschnittproblem mathematisch oft elegant und einfach formuliert werden kann, diese Formulierung zur Lösung des Problems aber ineffizient oder unbrauchbar ist und verschiedene Hilfskonstruktionen und -algorithmen gesucht werden müssen, um das Problem überhaupt nur annähernd lösen zu können. Es scheint schwierig, die Problemformulierung von der anzuwendenden Lösungsmethode zu trennen. Dies trifft für eine Grosszahl von Problemen aus der Klasse in der diskreten Optimierung zu, welcher die meisten Zuschnittprobleme angehören.

In der 'Praxis' sind diese kombinatorischen Probleme allerdings selten in Reinform anzutreffen; oft kommen mathematisch schwiwerig zu formulierende Einschränkungen hinzu. Nichtsdestotrotz wird der Einsatz von Techniken der mathematischen Optimierung im weitesten Sinn immer wichtiger.

Bei verschiedenen Optimierungsproblemen wie bei den Zuschnittproblemen fällt auf, dass Probleme, welche scheinbar nichts miteinander zu tun haben, auf dieselbe mathematische Struktur zurückführbar sind. Es soll hier aber erwähnt werden, dass häufig auch das Gegenteil anzutreffen ist, dass nämlich Probleme, welche oberflächlich verwandt scheinen, ein völlig andere mathematische Struktur haben.

Allgemein kann ein Optimierungsproblem folgendermassen formuliert werden:

> Es sei S eine gegebene Menge und f eine auf diese Menge definierte (reellwertige) Funktion. Es ist ein Element $x^* \in S$ so zu bestimmen, dass $f(x^*) \leq f(x)$ für alle $x \in S$.

Oder in der Kurzschreibweise

$$\{\text{minimiere } f(x) : x \in S\}.$$

f heisst **Zielfunktion** und S ist der **zulässige Bereich** oder der **Lösungsraum**. Jedes Element $x \in S$ ist ein **zulässiges Element** (zulässige Lösung).

Anschliessend an die Formulierung des Problems werden verschiedene Lösungsmethoden für die einzelnen Zuschnittprobleme kurz erläutert. Es geht dabei weniger darum, die Techniken im einzelnen zu erklären, sondern darzulegen, welche Methoden auf welche Zuschnittprobleme anzuwenden sind. Da all diese Probleme (mindestens) NP-hart sind, sollen vor allem auch heuristische Verfahren erörtert werden. Es sollte aber nicht vergessen werden, dass die Feststellung, dass ein Problem NP-hart ist, nicht bedeutet, dass keine exakten Algorithmen gesucht werden sollen. Gerade für diese Probleme wurde lange Zeit die Erforschung exakter Algorithmen vernachlässigt, zu unrecht, wie aus dem Buch von Martello al. [1990] zu erfahren ist. Dort werden nämlich effiziente, exakte Algorithmen für die verschiedenen Knapsack-Probleme vorgestellt.
Verschiedene Lösungtechniken können auf die Zuschnittprobleme angewandt werden:

- ganzzahlige (lineare) Optimierung
- Branch-und-Bound (Backtracking, Enumeration und Baumsuche)

- dynamische Optimierung
- Greedy-Techniken
- statistische Heuristiken
- Simulated Annealing
- genetische Algorithmen
- neuronale Netze
- Tabu-search-Techniken
- Mensch-Maschine Dialog

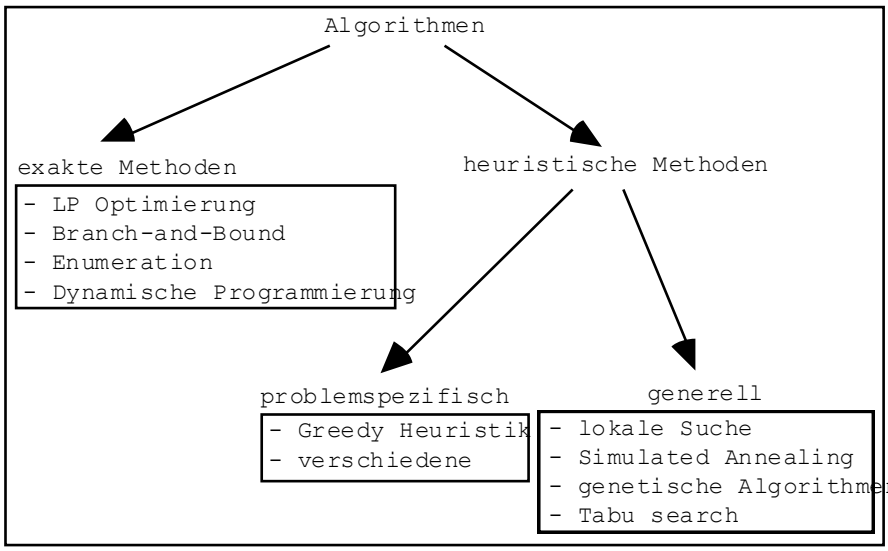


Abbildung 1-1

Die Methoden sind natürlich nicht überschneidungsfrei. Auf Grund der hohen Komplexität der Probleme kommen Kombinationen solcher Methoden in der Praxis häufig vor. Dabei muss zwischen exakten Verfahren und heuristische Näherungsverfahren unterschieden werden (Abbildung 1.1). Exakte Verfahren liefern die optimale Lösung eines Problems, wohingegen Näherungsverfahren meist nur einen Näherungswert für das Optimum liefern. Warum werden dann Näherungsverfahren überhaupt eingesetzt? *Weil die optimale Lösung durch exakte Verfahren in vernünftiger Zeit auch mit Hilfe der schnellsten Computer nicht gefunden werden kann*. Bei den Näherungsverfahren können die problemspezifischen Heuristiken von den allgemeinen, meist statistischen Verfahren unterschieden werden. Von beiden sollen in diesem Paper ein Vertreter vorgestellt werden.

Beispiele für exakte Verfahren für die ganzzahlige (lineare) Optimierungsprobleme sind der Simplex zusammen mit einer Branch-and-Bound Methode. Diese liefert in möglicherweise unakzeptierbar langer Zeit eine optimale Lösung des Problems, falls eine existiert. Für nicht-lineare Probleme gibt es keine universelle Lösungsverfahren

mehr. Es müssen heuristische Näherungsmethoden eingesetzt werden, ausser für ganz spezielle Problemfälle, welchen mit analytischen Mitteln beizukommen ist. Solche Methoden sind verschiedene Greedy-Techniken. Diese können ihrerseits mit statistischen Verfahren kombiniert werden. So können Greedy-Verfahren eine Population von verschiedenen, hoffentlich guten Näherungslösungen schnell finden, und zur weiteren Verbessung der Lösung werden dann genetische Algorithmen oder Simulated Annealing eingesetzt; zum Abschluss können die besten Lösungen unter Umständen durch die lokale Suche (hill-climbing-Techniken) weiter verbessert werden.

2. KNAPSACK- UND VERWANDTE PROBLEME

Bei der Lösung von Zuschnittproblemen fallen als Unterprobleme oft Knapsack-Probleme an. Daher sollen drei Varianten davon hier kurz eingeführt werden. Verwandte Probleme sind das Bin-Packing- und das Multiprocessor-Scheduling-Problem. Sie werden hier ebenfalls kurz vorgestellt, weil sie auch als Zuschnittprobleme betrachtet werden können.

Knapsack-Probleme sind ihrerseits NP-hart. Allerdings gibt es zu ihrer Lösung sehr effiziente Verfahren (Martello al. 1990).

Jedem Knapsack-Problem liegt folgendes mathematischen Problem zu Grunde:

> Gegeben eine Menge S, wähle aus dieser Menge eine Untermenge aus, welche verschiedenen Bedingungen erfüllt.

DAS 0-1 KNAPSACK-PROBLEM

Gegeben sei eine Menge j={1...n} und für jedes Element sei p_j der Wert und w_j das Gewicht des Elements. Zudem sei c ein vorgegebenes Gesamtgewicht. Es soll eine Untermenge von Elementen ausgewählt werden, welche den Gesamtwert maximieren, ohne dass das Gesamtgewicht der ausgewählten Elemente die Grösse c übersteigt.

Mathematisch kann das Problem formuliert werden als:

$$\begin{aligned} &\text{maximiere} \quad \sum_{j=1}^{n} p_j x_j \\ &\text{sodass} \\ &\sum_{j=1}^{n} w_j x_j \le c \\ &x_j \in \{0,1\} \end{aligned} \qquad \text{(0-1 Knapsack)}$$

Dieses scheinbar einfache Problem ist bereits NP-hart; allerdings nur im schwachen Sinne, da man für das Problem einen pseudo-polynomialen Algorithmus angegeben kann. Effiziente Lösungsmethoden für dieses Problem sind in Martello & Toth (1990) beschrieben.

DAS BOUNDED-KNAPSACK-PROBLEM

Wenn dieselben Elemente im 0-1 Knapsack-Problem maximal b_j-mal ($b_j>0$)

ausgewählt werden können, so erhalten wir das Bounded-Knapsack-Problem. Es kann formuliert werden als:

$$\begin{aligned} &\text{maximiere} \quad \sum_{j=1}^{n} p_j x_j \\ &\text{sodass} \\ &\quad \sum_{j=1}^{n} w_j x_j \le c \\ &\quad x_j \in \{0\text{K } b_j\} \end{aligned} \qquad \text{(Bounded-Knapsack-Problem)}$$

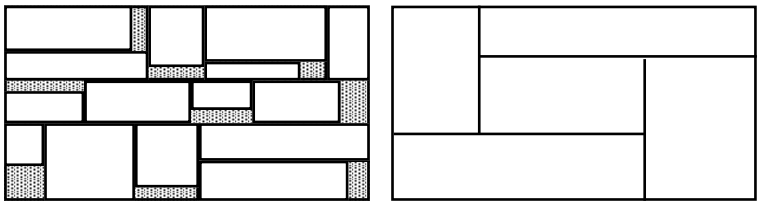
Dieses Problem wird mit Vorteil in ein 0-1 Knapsack-Problem übersetzt. Dies geschieht dadurch, dass jedes Element durch $\lfloor \log_2 b_j \rfloor$ Elemente mit den Werten und Gewichten $(p_j, w_j), (2p_j, 2w_j), (4p_j, 4w_j), \text{K}$ ersetzt wird (Martello & Toth 1990, S. 82).

In Martello & Toth ist auch ein FORTRAN-Kode mit die effizientesten Lösungsmethoden für verschiedene Knapsack-Probleme zu finden. Im Rahmen dieses Papers wurde ein PASCAL-Programm geschrieben, welches diesen FORTRAN-Code in PASCAL Source-Code übersetzt. Nach einer leichten, manuellen Nachbesserung entstand daraus der Code wie er im Anhang abgebildet ist (Prozeduren *Knapsack01()* und *KnapsackB()*).

DAS 2-SPACE-KNAPSACK-PROBLEM

Ein wichtige Rolle bei 2-dimensionalen Zuschnittproblemen spielt das 2-Space-Knapsack-Problem (Hinxman 1980). Dieses Problem ist wesentlich schwieriger als die einfachen Knapsack-Probleme. Gegeben sei eine Menge i={1...n} von Rechtecken mit der Ausdehnung (l_i,b_i) sowie ein grosses Rechteck (L,B). Gesucht ist die Untermenge der Rechtecke $S \subset i$, für welche der Ausdruck $LW - \sum_{k \in S} l_k w_k$ ein Minimum ist.

Eine leichte Verallgemeinerung dieses Problems ist in der Literatur unter dem Namen *2-dimensional Cutting-Problem* bekannt: Gegeben ein grosses Rechteck (L,B) und eine Menge kleiner Rechtecke R = { (l_1,b_1), (l_2,b_2),..., (l_n,b_n) }. Zu jedem Rechteck aus R gibt es eine Wert v_i und eine maximale Anzahl a_i. Wie müssen die Rechtecke von R aus dem grossen ausgeschnitten werden, damit der Gesamtwert $\sum_{i=1}^{n} v_i x_i$ (mit $x_i \le a_i$) maximal wird. Das 2-space-Knapsack-Problem entsteht dadurch, dass $v_i = l_i b_i$ gesetzt wird (der Wert ist proportional zur Fläche, der Verschnitt wird minimiert). Exakte Algorithmen gibt es nur wenige, da das Problem äusserst schwierig ist. Verschiedene Autoren haben sich dem Spezialfall, bei dem nur **Guillotinen-Schnitte** erlaubt sind, angenommen. Guillotinen-Schnitte sind Schnitte, welche ein Rechteck ganz in zwei kleinere Rechtecke zerlegen. Der Schnitt muss dabei durchgehend sein (Abb. 2-1a). Der Verschnitt in Abbildung 2-1b kann beispielsweise nicht durch einen Guillotinen-Schnitt realisiert werden.



a) Guillotinen-Schnitt b) kein Guillotinen-Schnit

Abbildung 2-1

Gilmore & Gomory (1965, 1966) geben ein Verfahren an, welches das auf Guillotinen-Schnitte beschränkte Problem mit Hilfe der dynamischen Programmierung löst. Beasley (1985a) weist auf einen Fehler des Gilmore & Gomory-Verfahrens hin. Bei dieser Methoden der exakten Problemlösung wird rekursiv die vorgegebene Fläche in kleinere Rechtecke verschnitten (Abbildung 2-2) (siehe auch Terno S.139ff).

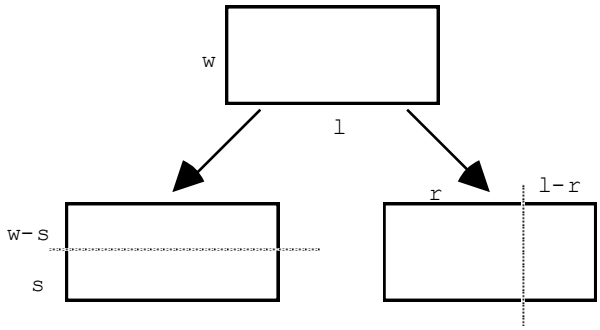


Abbildung 2-2

Herz (1972) schlägt eine rekursive Prozedur vor, Christofides & Whitlock (1977) geben ein Baumsuchverfahren an. Ein heuristisches Verfahren ist von Wang (1983) vorgetragen worden. Nur gerade ein Artikel befasst sich mit dem allgemeinen Problem, welches die Schnittart nicht auf Guillotinen-Schnitte einschränkt (Beasley 1985c). Beasley formuliert das Problem als ein 0-1-ganzzahliges Problem mit O(mLB) Variablen und O(LB) Restriktionen. Die Komplexität kann allerdings verringert werden, wenn nur normale Schnittmuster verwendet werden (siehe S.52).

DAS BIN-PACKING-PROBLEM

Wird das 0-1 Knapsack-Problem dahingehend erweitert, dass n Knapsäcke der Kapazität c gegeben sind, so spricht man vom Bin-Packing-Problem. Das Problem besteht nun darin, alle n Elemente in möglichst wenige Knapsäcke unterzubringen. Mathematisch kann das Problem formuliert werden als

$$\text{minimiere} \quad \sum_{j=1}^{n} y_j$$
$$\text{sodass}$$
$$\sum_{j=1}^{n} w_j x_{ij} \le c \quad \text{, für alle } i \in \{1\text{K } n\} \qquad \text{(Bin-Packing-Problem)}$$
$$\sum_{i=1}^{n} x_{ij} = 1 \quad \text{, für alle } j \in \{1\text{K } n\}$$
$$x_{ij}, y_j \in \{0,1\}$$

Die Variablen haben dabei folgende Bedeutung: Es ist y_i =1, wenn Knapsack *i* benützt wurde, sonst ist y_i = 0. Ferner ist x_{ij} = 1, wenn Element *j* im Knapsack *i* platziert wird, sonst ist x_{ij} = 0.

Greedy-Approximations-Verfahren spielen eine ausserordentliche Rolle bei der Lösung des Bin-Packing-Problems. Da diese Verfahren wiederum im zweidimensionalen Zuschnittproblem auftauchen und leicht zu implementieren sind, sollen einige klassische Verfahren kurz eingeführt werden:

Next-Fit (NF): Die Elemente werden der Reihe nach in denselben Knapsack gepackt, bis ein Element keinen Platz mehr hat, dann wird zum nächsten Knapsack übergegangen. Dies wird solange fortgesetzt, bis alle Elemente gepackt sind.

First-Fit (FF): Die Elemente werden der Reihe nach in denjenigen bereits teilweise gepackten Knapsack gepackt. Falls dies nicht möglich ist, so wird ein neuer Knapsack begonnen.

Best-Fit(BF): Wie FF, allerdings wird das Element in den Knapsack gepackt, welcher nach der Packung am wenigsten Leerraum übriglässt.

Die Verfahren *Next-Fit-Decreasing (NFD)*, *First-Fit-Decreasing (FFD)* und *Best-Fit-Decreasing (BFD)* entstehen aus den drei oben beschriebenen Verfahren dadurch, dass die Elemente absteigend nach w_j geordnet sind.

All diese Verfahren bilden Approximationsverfahren, d.h. die erhaltene Lösung ist nicht notwendigerweise die optimale. Daher stellt sich für diese approximative Algorithmen die Frage, wie weit entfernt die erhaltene Lösung vom Optimum ist. Sei OPT(I) die optimale Lösung einer Probleminstanz I und sei A(I) die Lösung der Probleminstanz I, die durch den Algorithmus A erhalten wurde, so definieren wir

$$r_A(I) = \frac{A(I)}{OPT(I)} .$$

Die **worst-case performance ratio** r_A ist dann folgendermassen definiert durch

$$r_A = \inf\{r \ge 1 : r_A(I) \le r \text{, for all instances } I\} .$$

Diese Zahl gibt über alle Probleminstanzen die kleinste Zahl grösser eins wieder, für welches das Verhältnis $r_A(I)$ am grössten ist, d.h. man wähle die Lösung der Probleminstanz mit dem grössten Lösungswert und vergleiche diese mit der optimalen Lösung. Ist z.B. r_A = 2, so bedeutet dies, dass der Algorithmus eine maximal 100%-ig schlechtere Lösung liefert als das Optimum.

Die **asymptotic performance ratio** r_A^∞ ist definiert als

$$r_A^\infty = \inf\{r \geq 1 : \exists N > 0, r_A(I) \leq r \text{ , for all } I \text{ with } OPT(i) \geq N\}.$$

Diese Zahl wiedergibt eine etwas abgeschwächte Form der worst-case performance ratio: die Ungleichung $r_A(I) \leq r$ wird nicht mehr für alle Instanzen jedoch für die 'meisten' gefordert. Die Definition erlaubt unter anderem, konstante Terme in der Analyse zu unterdrücken (siehe dazu Coffman al. 1984).

Nach der Einführung der Effizienzgrade für approximative Algorithmen können diese nun für die obigen Algorithmen angegeben werden. Es gelten folgende Beziehungen:

$$r_{NF}^\infty = 2 \text{ , } r_{FF}^\infty = r_{BF}^\infty = \tfrac{17}{10} \text{ , } r_{NFD}^\infty = 1.691\ldots \text{ , } r_{FFD}^\infty = r_{BFD}^\infty = \tfrac{11}{9}$$

Je näher r_A^∞ bei eins ist, desto besser der Algorithmus im schlimmsten Falle.

DAS MULTIPROCESSOR-SCHEDULING-PROBLEM

Dieses Problem wird gewöhnlich den Scheduling-Problemen zugerechnet, ist aber mit dem Bin-Packing-Problem eng verwandt. Gegeben sei eine Menge von Aufgaben (tasks), die je auf einem beliebigen Prozessor auszuführen sind. Jede Aufgabe beansprucht eine bestimmte Zeitspanne. Wie sollen die Aufgaben auf die Prozessoren verteilt werden, sodass alle Aufgaben in minimaler Zeit erledigt werden können.

3. VERSCHIEDENE ZUSCHNITTPROBLEME

Im folgenden sollen verschieden Zuschnittprobleme formuliert werden. Die Übersicht der kurz besprochenen Probleme ist in Abbildung 3-1 zu sehen. Wir werden uns auf einige Varianten des 1- und des 2-dimensionalen Zuschnittproblems geschränken. Beim 1-dimensionalen Problem geht es darum 1-dimensionalen Objekte in kleinere 1-dimensionale Objekte zu verschneiden (z.B. Stangenmaterial). Es sollen drei Varianten vorgestellt werden. Etwas ausführlicher soll das 2-dimensionale Problem, von denen es eine unübersichtliche Anzahl Varianten gibt, vorgestellt werden. Beim 2-dimensionale Problem sollen aus grösseren Flächen kleinere Flächen ausgeschnitten werden. Diese Probleme werden zunächst nach der Form der grossen Objekte aufgeteilt. Die Form kann ein Rechteck, ein Streifen (die Höhe ist unendlich), oder beliebig sein.
Der dritten Fall soll nicht weiterverfolgt werden (Ausschneiden aus Fellen).
Der erste Fall führt auf das Problem, aus Rechtecken kleinere Formen auszuschneiden. Dabei können drei Fälle unterschieden werden: 1. Eine fixe Grösse von grossen Rechtecken ist gegeben: wieviele grosse Rechtecke braucht es, um eine gegebene Menge von kleinen Formen auszuschneiden? 2. Verschiedene Grössen des

grossen Rechteck sind gegeben: von welchen braucht es wieviele grosse Rechtecke, um die kleinen Formen auszuschneiden? 3. Die minimale Grösse des grossen Rechtecks ist gesucht, sodass alle kleineren Formen Platz haben.

Beim zweiten Fall ist das grosse Objekt als Streifen unendlicher Höhe gegeben, und es sollen alle kleinen Formen so platziert werden, dass ein möglichst kurzer Streifen verbraucht wird.

Die Variante, dass die kleineren Flächen nicht Rechtecke sind, soll hier auch nicht weiterverfolgt werden (Kleider-, Schuhindustrie).

Weiter wird unterschieden, ob die kleineren Rechtecke orthogonal (parallel zum grossen Rechteck) oder nicht orthogonal platziert werden können. Bei der orthogonalen Variante wird zudem zwischen den Fällen unterschieden, ob eine Drehung der kleinen Rechtecken um 90° erlaubt ist oder nicht. Bei den meisten praxis-relevanten Anwendungen ist eine solche Drehung nicht erlaubt. Unabhängig von dieser Drehung gibt es bei den orthogonalen Zuschnitten je nach Schnittarten verschidene Varianten. Guillotinen-Schnitte (solche, die ein Rechteck vollständig in zwei oder mehrere kleinere Rechtecke zerlegen, siehe weiter unten), sind z.B. in der Glasindustrie üblich. Bei den Guillotinen-Schnitten kann unterschieden werden, wie oft die Aufteilung rekursiv fortgesetzt werden kann: bei 2-stage-Schnitten kann das grosse Rechteck in kleinere und diese wiederum in kleinere zerlegt werden. Jedoch die durch den zweiten Schnitt entstanden Rechtecke dürfen nicht mehr weiter aufgeteilt werden. 3-stage-Schnitte sind in der Glasindustrie üblich.

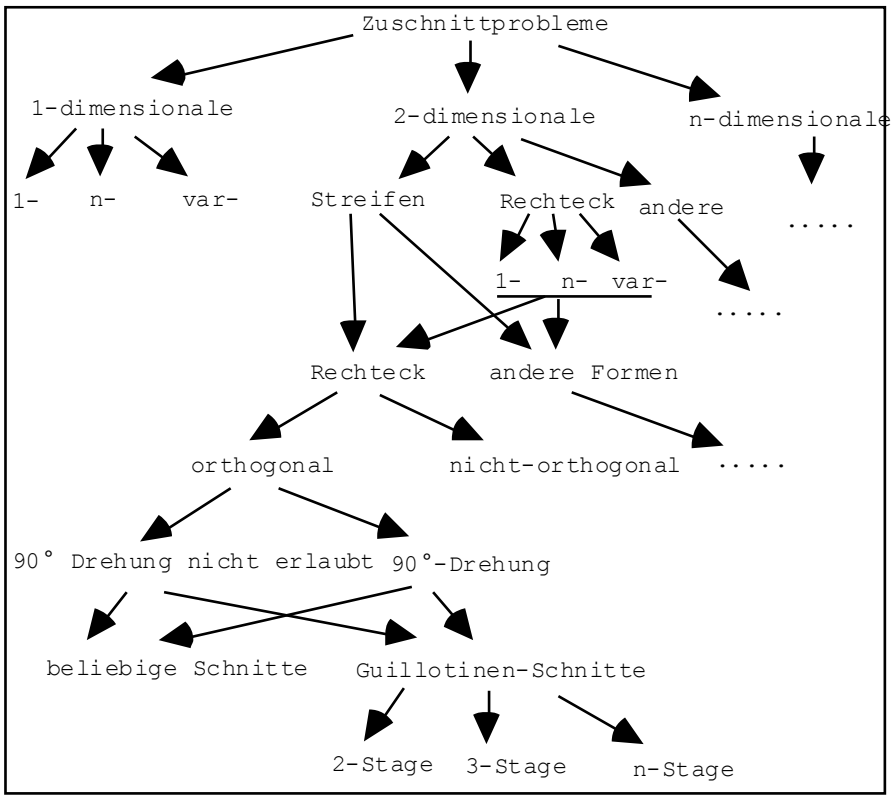


Abbildung 3-1

Zu (fast) jedem Problem soll ein konkretes Beispiel sowie eine formale, mathematische Formulierung, falls möglich, gegeben werden. Verschiedene Lösungsmethoden werden erörtert. Dabei werden oft Methoden, die sich ähneln und auch für andere Varianten einsetzbar sind, zusammenbehandelt.
Die in Abbildung 3-1 wiedergegebene Taxonomie ist natürlich etwas willkürlich, wiederspiegelt aber die in der Literatur behandelten Probleme am besten.

3.1 DAS 1-DIMENSIONALE ZUSCHNITTPROBLEM (CUTTING-STOCK)

Es soll zunächst ein konkretes Beispiel für das 1-1-dimensionale Zuschnittproblem gegeben werden. Hernach wird eine allgemeine mathematische Formulierung vorgeschlagen.

Beispiel

In der Holzverarbeitung tritt das Problem auf, Stangen gegebener Länge in kleinere Längen zuzuschneiden. Zum Beispiel sollen aus Stangen der Länge L=6500mm

50 Stangen der Länge 2000mm
100 Stangen der Länge 1600mm und
100 Stangen der Länge 1100mm

zugeschnitten werden. Wieviel Stangen der Länge 6500mm werden mindestens und in welchen Kombinationen verbraucht und wie gross ist der Verschnitt (Abfall)?
Offensichtlich gibt es mehrere Möglichkeiten, das Ausgangsmaterial zuzuschneiden. Aus eine Stange von 6500mm können zum Beispiel 3 Stangen der Länge 2000mm mit eine Verschnitt (*c*) von 500mm produziert werden. Insgesamt erhält man 11 Verschnittvarianten (Tabelle 3-1).

Variante	1	2	3	4	5	6	7	8	9	10	11
2000mm	3	2	2	1	1	1	0	0	0	0	0
1600mm	0	1	0	2	1	0	4	3	2	1	0
1100mm	0	0	2	1	2	4	0	1	3	4	5
Verschnitt	500	900	300	200	700	100	100	600	0	500	1000

Tabelle 3-1

Das 1-1-dimensionale Problem

Das zuzuschneidende Material (das grosse Objekt) ist in einer fixen Grösse L, wie im einleitenden Beispiel, gegeben. Die Verschnittvarianten können durch einfache Enumeration erhalten werden (siehe Prozedur *GenVariants* im Anhang). Sind alle Varianten bekannt, stellt sich die Frage, von welchen Varianten wieviele Stangen zu zerschneiden sind. Um die 100 1100mm Stangen zu gewinnen, könnten wir 20 mal

Variante 11 anwenden. Die 1600mm Stangen könnten durch Variante 7 (25 Stangen) und die Stangen zu 2000mm durch Variante 1 (17 Stangen) produziert werden. Wir müssen dabei 20+25+17=62 Stangen zerschneiden, und der gesamte Verschnitt kann berechnet werden durch 20x1000+25*100+17*500+2000=33000. Verschiedene Variantenkombinationen sind möglich. Um alle Kombinationen systematisch zu erfassen, wird das Problem in eine mathematische Form gefasst. x_j sei die unbekannte Anzahl der Variante *j*. Die Unbekannten müssen offenbar folgende Ungleichungen erfüllen:

$$3x_1 + 2x_2 + 2x_3 + x_4 + x_5 + x_6 \geq 50$$
$$x_2 + 2x_4 + x_5 + 4x_7 + 3x_8 + 2x_9 + x_{11} \geq 100$$
$$2x_3 + x_4 + 2x_5 + 4x_6 + x_8 + 3x_9 + 4x_{10} + 5x_{11} \geq 100$$

Um die Anzahl der zu zerschneidenden Stangen zu minimieren, muss der Ausdruck $z = \sum_{j=1}^{11} x_j$ möglichst klein werden. Eine andere Möglichkeit wäre, nicht die Anzahl der Stangen sondern den gesamten Verschnitt zu minimieren ($z = \sum_{j=1}^{11} c_j x_j$).

Allgemein: i={1..m} sei die zu produzierende Menge mit den Längen l_i und b_i Stückzahlen, N={1..n} die Menge der Varianten, *L* die Ausgangslänge, a_{ij} die Anzahl der Längen l_i in der Variante j und c_j der Verschnitt der Variante. Dann kann das Problem formuliert werden als

minimiere $z = \sum_{j=1}^{n} x_j$

mit

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i \quad \text{, for all } i = \{1 \text{K } m\}$$
$$x \in N$$

Damit haben wir ein ganzzahliges, lineares Optimierungsmodell, welches mit der Branch-und-Bound Methode zusammen mit dem Simplex-Verfahren gelöst werden kann. Zwei Schwierigkeiten entstehen dabei: die Ganzzahligkeitsbedingung beschränkt die Lösung auf kleine Probleme. Das Problem wird meist umgangen, indem diese Bedingung fallengelassen wird. Bruchzahlen werden dann einfach auf- oder abgerundet, je nachdem ob eine Über- oder Untererfüllung erwünscht ist. Die zweite Schwierigkeit besteht darin, dass die Anzahl der verschiedenen Varianten sehr gross sein kann. Für eine typische Aufgabe mit einer Ausgangslänge von 200mm, welche in 40 verschiedene, kleinere Längen l_i mit $20 \leq l_i \leq 40$ hat Gilmore & Gomory (1963) eine Anzahl von 10^7 Varianten geschätzt. Da die Problemgrösse des Optimierungsmodells mit der Anzahl Varianten wächst (Anzahl Variablen=Anzahl Varianten), kann das Problem theoretisch zwar formuliert, aber praktisch nicht gelöst werden. Ein Ausweg ist, einige Varianten, welche einen kleinen Verschnit aufweisen, auszuwählen und im Verlaufe der Lösung neue Varianten, welche die Lösung

verbessern, zu generieren. Gilmore & Gomory (1963) haben eine systematische Methode dafür entwickelt. Die Variantengenerierung führt dabei auf ein **Bounded-Knapsack Problem**.

Algorithmus von Gilmore & Gomory (1961, 1963):

1. Generiere einige Varianten (durch zufällige Auswahl, oder durch eine systematische Methode. Eine solche Methode ist: Wähle m Varianten, wobei jede Variante genau eine Länge enthält).
2. Löse das LP-Problem mit Hilfe des Simplexalgorithmus
3. s_i seien die Schattenpreise in der LP-Lösung. a_i sei eine neue, unbekannte Variante. Löse das bounded-Knapsack-Problem:

   maximiere $\sum_{i=1}^{m} s_i a_i$

   sodass

   $\sum_{i=1}^{m} l_i a_i \le L$

   $a_i \in \{0K\ L\}$
4. Falls eine solche Variante a nicht existiert, so stoppe, sonst füge die Variante a dem LP als neue Kolonne hinzu und gehe zu Schritt 2.

Das n-1-dimensionale Problem

Es seien s unterschiedliche Längen L_j des Ausgangsmaterials gegeben, die in m kleinere Längen l_i in den Stückzahlen b_i zu zerschneiden sind. Dieses Problem kann genaugleich, wie das 1-1-dimensionale Problem formuliert und gelöst werden, ausser dass die Variantengenerierung modifiziert werden muss: Statt Varianten zu einer gegeben Ausgangsgrösse, sind nun Varianten zu verschiedenen Ausgangsgrössen gegeben. Sei beim obigen Beispiel noch die Ausgangslänge 5500mm gegeben, dann kommen zu den in Tabelle 3-1 zu der Länge 6500mm definierten Länge noch weitere 8 Varianten hinzu (Tabelle 3-2).

Variante	12	13	14	15	16	17	18	19
2000mm	2	1	1	1	0	0	0	0
1600mm	0	2	1	0	3	2	1	0
1100mm	1	0	1	3	0	2	3	5
Verschnitt	400	300	800	200	700	100	600	0

Tabelle 3-2

Der Gilmore & Gomory kann ebenfalls auf dieses Problem angewandt werden: im Schritt 3 müssen mehrere Knapsack-Probleme (für jede Ausgangslänge maximal eines) gelöst werden.

Ein Alternative, das Problem vollständig als LP-Modell zu formulieren ist folgende (Terno S. 16): x_{jk} sei die Anzahl der Längen L_j, die nach der Variante $k = \{1..t_j\}$, und y_{ijk} sei die Anzahl der Teillängen l_i in der k-ten Variante von L_j. Dann kann das

Problem durch folgendes nicht-lineares, diskretes Optimierungsmodell formuliert werden.

$$\text{minimize} \quad z = \sum_{j=1}^{s} L_j \sum_{k=1}^{t_j} x_{jk}$$

where

$$\sum_{j=1}^{s} \sum_{k=1}^{t_j} y_{ijk} x_{jk} \geq b_i \quad , \quad i = \{1\text{K}\ m\}$$

$$\sum_{i=1}^{m} l_i y_{ijk} \leq L_j \quad , \quad k = \{1\text{K}\ t_j\},\ j = \{1\text{K}\ s\}$$

Es ist im allgemeinen jedoch nicht möglich, dieses Problem mit Hilfe von analytischen Mittel zu lösen.

Das var-1-dimensionale Problem: Zerschneiden von Papierrolle

Wie im obigen Problem sollen aus Papierrollen der Breite L kleinere Rollen produziert werden. Die Breite L kann dabei innerhalb einer Bandbreite in diskreten Grössen gewählt werden. Die für den Verschnitt zu bestellende Breite L soll so gewählt werden, dass der Gesamtverschnitt minimal wird. Dieses Problem wird so gelöst, dass zu jedem L das obige Optimierungsmodell gelöst wird. Es wird dann die Breite L ausgewählt, welche den minimale Verschnitt liefert.

2.2 Das 2-dimensionale Zuschnittproblem aus Rechtecken

Bei dieser Problemklasse sollen nur Probleme betrachtet werden, bei denen die Formen der grossen und kleinen Objekte auf Rechtecke beschränkt ist.

Zuschnitt von Rechtecken aus einem Rechteck: Beispiel Zuschnitt von Blechen

Aus einer gegebenen Grösse 1000mm x 1000mm sollen kleinere rechteckige Bleche ausgeschnitten werden der folgenden Grösse und Anzahl:

400mm x 400mm	15 Stück
300mm x 600mm	10 Stück
700mm x 300mm	5 Stück

Dieses Problem ist identisch mit dem klassischen Cutting-Stock-Problem (1-1-dimensionales Problem, ausser dass die Verschnittvarianten verschieden generiert werden müssen. Das Generieren der Varianten ist dabei schwieriger und führt auf ein 2-space-Knapsack. Wenn die Walzrichtung der kleineren Bleche vorgegeben ist, so gibt es genau 6 Varianten (1-6) (Tabelle 3-3) (siehe auch Abbildung 3-3).

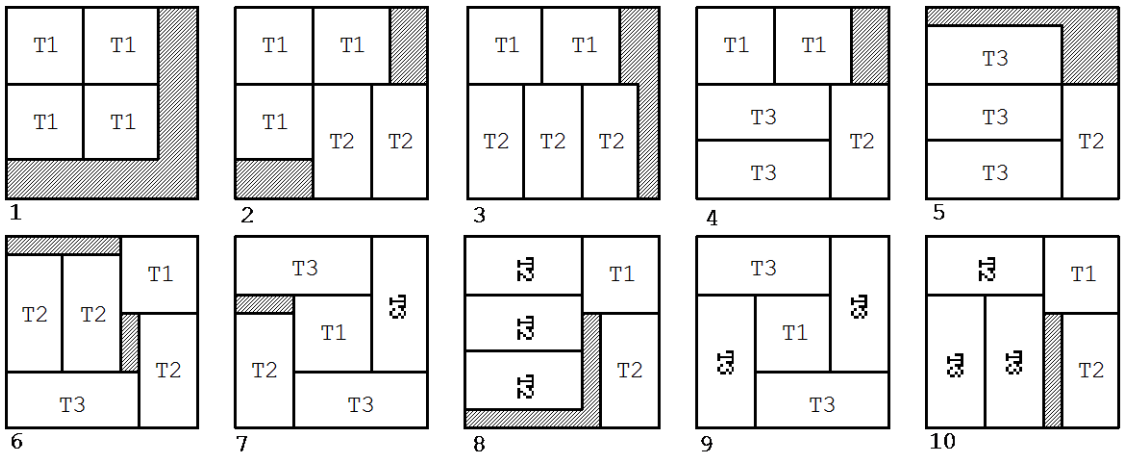


Abbildung 3-3 (Terno S.17-18)

Variante	1	2	3	4	5	6
T1: 400x400mm	4	3	2	2	0	1
T2: 300x600mm	0	2	3	1	1	3
T3: 700x300mm	0	0	0	2	3	1
Verschnitt	36	16	14	8	19	9

Tabelle 3-3

Ist das Drehen der Rechtecke um 90° erlaubt, so erhalten wir 9 Varianten (Tabelle 3-4). (Man beachte, dass Variante 7 die Variante 5 dominiert, wenn die Rechtecke gedreht werden). Sind nur Guillotinen-Schnitte zugelassen, so dürfen die Varianten 6, 7 und 9 nicht generiert werden.

Variante	1	2	3	4	(5)	6	7	8	9	10
T1: 400x400mm	4	3	2	2	0	1	1	1	1	1
T2: 300x600mm	0	2	3	1	1	3	1	4	0	2
T3: 700x300mm	0	0	0	2	3	1	3	0	4	2
Verschnitt	36	16	14	8	19	9	3	12	0	6

Tabelle 3-4

Zuschnitt von Rechtecken aus mehreren, verschieden grossen Rechtecken

Sind verschiedene Rechtecke unterschiedlicher Grösse als Verschnittmaterial gegeben, so kann das Problem analog wie beim n-1-dimensionalen Problem behandelt werden: Statt nur Verschnittvarianten eines Rechtecks zu betrachten, müssen jetzt alle Varianten der verschiedenen Ausgangsrechtecke betrachtet werden. Die Verschnittvarianten müssen jetzt mit mehreren 2-space-Knapsack-Problemen generiert werden.

Zuschnitt von Rechtecken aus einem Rechteck variabler Grösse

Alle kleinen Rechtecke sollen so platziert werden, dass sie in einem Rechteck minimaler Grösse angeordnet sind. Spezielle Probleme dieser Art haben Eingang in die Unterhaltungsmathematik gefunden. Als Beispiel sei das Packen von Einheitsquadraten in einem Quadrat gegeben: Gegeben n Einheitsquadrate, suche das Quadrat minimaler Fläche (k^2) , welches die n Einheitsquadrate packt. Wenn n<6 ist, konnten die Lösungen gefunden werden, wobei einzig n=5 nicht trivial ist (Abbildung 3-4). Für n>5 sind zwar Lösungen bekannt, aber bisher konnte nicht bewiesen werden, ob diese optimal sind. (Siehe z.B. für n=19 in Gardner 1979).

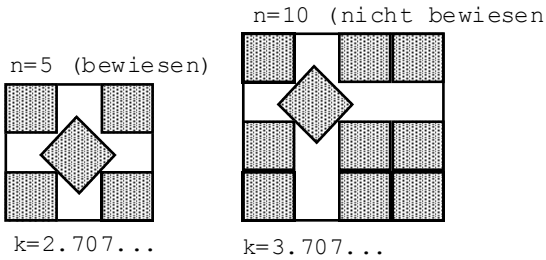


Abbildung 3-4

Nicht-orthogonale Verschnittprobleme sollen hier nicht weiter behandelt werden. Es sind offenbar auch keine systematische Verfahren bekannt, mit denen diese Probleme angegangen werden könnte. Wie in Abbildung 3-4 zu sehen ist, ist es für dieses offenbar einfache Beispiel mit 10 Einheitsquadraten völlig offen, ob es eine bessere Lösung gibt oder nicht.

2.2 Das 2-dimensionale Zuschnittproblem aus Streifen

Das grosse Objekt ist ein unendlich langer Streifen fixer Breite. Daraus sind die kleinen Objekte so auszuschneiden, dass eine minimaler Streifenlänge verwendet wird.

Ein abstraktes, geometrisches Beispiel wurde in der Einleitung gegeben. Ein konkretes Problem entsteht dadurch, dass man die Breite als ein fix vorgegebener Speicher (RAM) und die Höhe als Zeitdimension interpretiert. Die zu platzierende Rechtecke sind Tasks, die auf einem Computer ausgeführt werden müssen. Jeder Task besitzt eine fix vorgegebene Zeitdimension (Höhe) und Speicherplatzdimension (Breite). Die Tasks sollen so auf dem Computer ausgeführt werden, dass alle Task in kürzester Zeit abgearbeitet sind; die Ausführungsreihenfolge untereinander spielt dabei keine Rolle. Dieses Problem entspricht dem geometrischen Problem der Platzierung von Rechtecken in einem Streifen vorgegebener Breite. Das Beispiel könnte man dahingehend verallgemeinern: Gegeben sind eine Anzahl Projekten j = {1..n}, die je von zwei Resourcen (z.B. Zeit und irgendeine Materialmenge) eine fixe Menge t_j und r_j beanspruchen. Die Projekte sind voneinander unabhängig und können in beliebiger Reihenfolge ausgeführt werden. Da eine der Resourcen nicht in

ungeschränkter Menge zu Verfügung steht, können die Projekte nicht alle gleichzeitig ausgeführt werden. Die zeitliche Abfolge bei minimaler Zeit muss bestimmt werden. Ein Beispiel mit 24 Projekten ist in Tabelle 3-5 gegeben (Terno S. 23)

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
t_j	5	8	4	10	5	10	4	9	10	7	6	8	7	4	4	5	8	9	7	4	10	8	8	10
r_j	8	4	10	6	7	4	9	7	8	4	10	8	7	7	8	6	8	4	9	4	5	5	6	9

Begrenzung der Resource R_j auf 25, T_j keine Begrenzung.

Tabelle 3-5

Ein Tupel (r_j,t_j) kann als Breite und Höhe eines Rechtecks interpretiert werden. Die Rechtecke sollen so auf einem Streifen von einer Breite mit 25 Einheiten platziert werden, dass die Höhe des verbrauchten Streifens minimal wird. (Drei gute Lösungen dieses Problems sind in Terno S 24 abgebildet, gibt es bessere?).

Weitere Problemvarianten

Aus technologischen, materialspezifischen oder andern Gründen sind nicht alle Verschnittvarianten in allen Verschnittprozessen möglich. Wenn die Walzrichtung für die Metallzuschneidung z.B. keine Rolle spielt, so dürfen die Rechtecke um 90° gedreht werden, und es kann im allgemeinen eine Lösung mit kleinerem Verschnitt gefunden werden. Dasselbe trifft meist für die Glaszerschneidung zu, obwohl es scheint, dass der Aufwand stark ansteigt. In der Glasindustrie werden daher meist nur Lösungen vorgeschlagen, welche die Rechtecke nicht drehen.

Für das eingangs erwähnte Beispiel der Resourcenallokation dürfen hingegen die Rechtecke (Tasks) nicht gedreht werden, da die Zeit- und Speicherresource nicht austauschbar sind.

Eine weitere Einschränkung ist oft vom Material her gegeben: Metall kann zwar ausgestanzt werden, Glas hingegen nicht. Daher kann Glas nur in sogenannten Guillotinen-Schnitten zugeschnitten werden, Metall hingegen kann beliebig 'um die Ecke' geschnitten werden. Verschiedene Algorithmen liefern verschiedene Schnittmuster: Die KLM-, die Level- und Shelf-Heuristiken liefern Guillotinen-Schnitte, BL-Heuristiken hingegen nicht (siehe unten)

Weitere Einschränkungen ergeben sich dadurch, dass die auszuschneidenden Rechtecke permanent anfallen und deren Menge nicht zum voraus gegeben ist. Dies bedeutet insbesondere, dass ein Rechteck r_i ohne Kenntnisse der Rechtecke r_k: $k>i$ angeordent werden muss. Algorithmen, welche dieser Forderung genügen müssen, bezeichnet man als **on-line-Algorithmen**. Die andere Algorithmen, welche eine vorgegebene Liste von Rechtecken voraussetzt, werden als **off-line-Algorithmen**

bezeichnet. Level- und Shelf-Algorithmen gehören eher zu den on-line-Algorithmen, während CLS, KLM den off-line-Algorithmen zuzurechnen sind. BL-Heuristiken können für beide verwendet werden.

Exakte Lösungsverfahren

Allgemein kann das Problem folgendermassen formuliert werden: Gegeben seien j = {1...n} Rechtecke der Grösse (b_j,h_j). Es sei (x_j,y_j) die unbekannte, zu bestimmende, linke-untere Koordinate des Rechtecks innerhalb der zu platzierende Ebene. Die linke untere Ecke der Ebene wird auf (0,0) normiert (Abbildung 3-4). Die Gesamtbreie wird mit *B* bezeichnet.

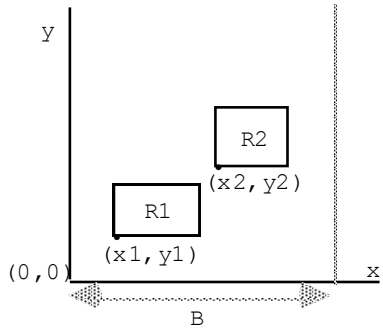


Abbildung 3-4

Die Bedingung, dass sich zwei Rechtecke (b_k,h_k) und (b_l,h_l) nicht überschneiden, kann dann formuliert werden als

$$(x_k + b_k \le x_l) \vee (x_l + b_l \le x_k) \vee (y_k + h_k \le y_l) \vee (y_l + h_l \le y_k) \quad \text{for all } k,l \in i$$

Die Forderung, dass alle Rechtecke innerhalb der Breite sein müssen, kann ausgedrückt werden durch:

$$0 \le x_k \le B - b_k\,,\ 0 \le y_k \quad \text{for all} \, k \in i$$

Die Zielfunktion, dass die Gesamthöhe möglichst klein sein soll, wird wiedergegeben durch

$$\text{minimiere} \quad \max_{k=1}^{n}(y_k + h_k)$$

Da diese Formulierung direkt in ein MIP-Problem umgewandelt werden kann, könnte das Problem mit den Standardmethoden (Simplex, Branch-and-Bound) gelöst werden. Leider führt die Umwandung zu einem grossen Modell. Schon bei 10 Rechtecken müssen 100 0-1-Variablen eingeführt werden. Die Zahl der Variablen steigt im Quadrat mit der Anzahl Rechtecke.

Problemspezifische Heuristiken

Am aussichtsreichsten, das Problem durch ein Näherungsverfahren zu lösen, sind *problemspezifische, heuristische Verfahren*, von denen jetzt eine Reihe in Anlehnung

an das Bin-Packing-Problem vorgestellt werden sollen. Ein wichtige Klasse von Algorithmen könnte unter dem Begriff **bottom-up, left-justified** (BL-Algorithmen) zusammengefasst werden. Dabei geht man von einer bestimmten vorgegebenen Anordnung (Permutation) der Rechtecke aus. Das jeweils nächste Rechteck in der Anordnung wird so weit unten wie möglich und auf dieser Höhe so weit links wie möglich platziert (Abbildung 3-5).

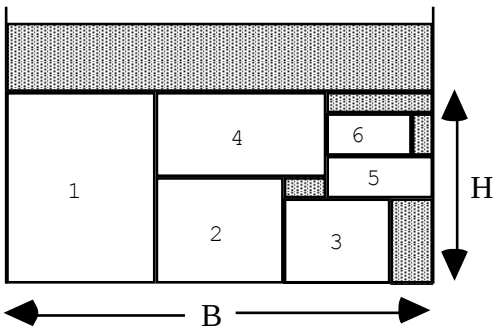


Abbildung 3-5

Jede Permutation liefert demnach eine bestimmte Höhe. Bei einer ungünstigen Permutation kann die Höhe beliebig weit von der optimalen Höhe abweichen. Da es n! Permutationen gibt, gibt es auch dementsprechend Anzahl Lösungen. Es ist naheliegend, die Güte des Algorithmus mit Hilfe der worst-case performance ratio r_A - wie oben definiert - zu untersuchen, wenn die Rechtecke nach auf- oder absteigender Breite (increasing, decreasing width (BLIW, BLDW)) oder nach auf- oder absteigender Höhe (increasing, decreasing height (BLIH, BLDH)) sortiert sind. Es gelten folgende Sätze (alle aus Baker al. 1980):

1. Für jedes M>0 gibt es eine Liste von Rechtecken, sodass r_{BLIW}>M ist.
1'. Für jedes M>0 gibt es eine Liste von Rechtecken, sodass r_{BLDH}>M ist.
2. Für jedes M>0 gibt es eine Liste von Rechtecken, sodass r_{BLDW}>2-M.
3. Für jede Liste von Rechtecken gilt r_{BLDW} <= 3.
4. Besteht die Liste der Rechtecke aus Quadrat so gilt r_{BLDW} <= 2.

Eine weitere Möglichkeit besteht darin, die Rechtecke nach ihrer Fläche (Area) zu ordnen: dann hätten wir den BLA-Algorithmus.

Eine Frage stellt sich natürlich sofort: Liefert eine der n! Permutation immer die optimale Lösung? Leider ist dies nicht der Fall; mit andern Worten, es gibt eine Liste von Rechtecken, für die *jeder* bottom-up, left-justified Algorithmus mindestens eine Höhe 12/11 über der optimalen Höhe liefert. Diese etwas erstaunliche Tatsache kann an einem Beispiel gezeigt werden (Baker 1980, S. 852). Gegeben seien neun Quadrate mit den Seitenlängen { 6, 6, 5, 5, 4, 4, 3, 1, 1 }. Diese sollen in einem Streifen der Breite 15 platziert werden. Die optimale Lösung ist in Abb. 3-6a wiedergegeben. Die

BL-Heuristik findet diese Platzierung durch die Permutation {5, 4, 6, 1, 3, 6, 1, 5, 4 }. Wird nun die Streifenbreite auf 15+ε und das Quadrat mit der Seitenlänge 3 auf 3+ε vergrössert, so ist die Platzierung in Abbildung 3-6b offenbar optimal. Die Lösung der Abbildung 3-6b verletzt aber die BL-Packung, und kann daher von keiner Permutation generiert werden.

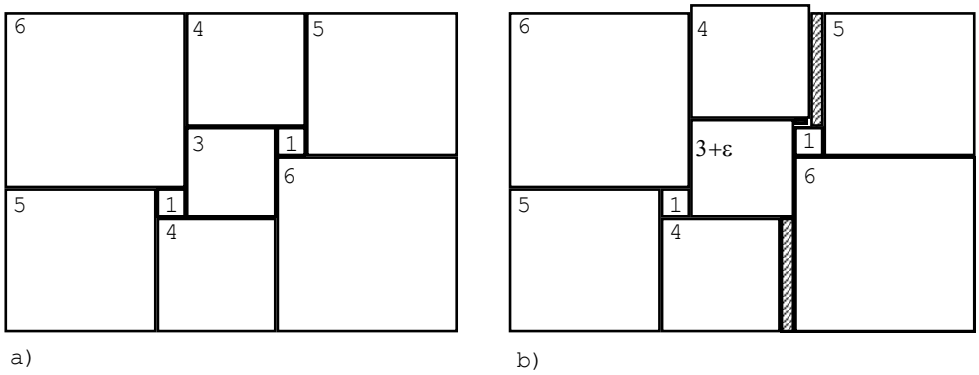


Abbildung 3-6

Die BR (bottom-up right-justified)-Heuristik wäre nichts anderes als eine symmetrische Variante der BL-Heuristik. Wenn hingegen beide Varianten zu einer **BLR-Heuristik** erweitert wird (von mir), so können dadurch mehr Packmuster generiert werden. Die BLR-Heuristik besteht darin, dass - wie bei der BL-Heuristik - die Rechtecke in einer Ordnung vorgegeben werden. Zudem ist ein boolscher Wert für jedes Rechteck gegeben. Das jeweils nächste Rechteck in der Ordnung wird so weit unten wie möglich und auf dieser Höhe so weit links wie möglich, wenn der boolsche Wert T ist, oder soweit rechts wie möglich, wenn der boolsche Wert F ist, platziert.

> Behauptung (von mir): die BLR-Heuristik liefert alle Platzierungsmuster, unter denen auch die optimale Lösung sich befindet.

Immerhin liefert die BLR-Heuristik die optimale Lösung für das Problem in Abbildung 3-6b, wenn die Permutation {5, 4, 6, 1, 3, 6, 1, 5, 4 } mit dem boolschen Vektor {T, T, F, T, T, T, T, T, T } vorliegt.

Eine weitere Klasse von Algorithmen sind die **Level-Heuristiken**. Drei Varianten Next-Fit-Level (NFL), First-Fit-Level (FFL) und Best-Fit-Level (BFL) können direkt von den Bin-Packing-Heuristiken abgeleitet werden. Beim NFL-Algorithmus werden die Rechtecke von links nach rechts angeordnet, bis das nächste Rechteck auf der Ebene keinen Platz mehr hat. Dann wird eine horizontale Linie auf der Höhe der oberen Kante des platzierten Rechtecks, welches alle überragt, gezogen. Die nächsten

Rechtecke werden beginnend auf dieses Horizontalen platziert. FFL und BFL sind sinngemäss anzupassen. Der NFL-Algorithmus ist geeignet für die on-line Verarbeitung und ist ein linearer Algorithmus. Dies ist nicht der Fall für beide FFL und BFL-Algorithmus. Werden die verbleibenden Breiten der Levels in einem 2-3-Baum organisiert, so ist die Komplexität beider Algorithmen O(nlog(n)). Werden die Rechtecke zusätzlich nach absteigender Höhe sortiert (DH) und dann auf Levels angeordnet, so entstehen weitere Varianten. (Abbildung 3-7).

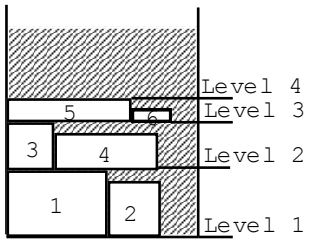


Abbildung 3-7

Level 1 ist die Grundlinie. Jedes Level ist durch die horizontale Linie definiert, welche durch die Oberkante des höchsten (ersten) Rechtecks des aktuellen Levels gelegt ist. In Anlehnung an die Bin-Packing-Heuristiken können dann Next-Fit-Decreasing-Height (NFDH), First-Fit-Decreasing-Height (FFDH) und Best-Fit-Decreasing-Height (BFDH) entsprechend definiert werden. Es gelten die folgenden Formeln wie für den eindimensionalen Fall:

$r_{NFDH} = 3$, $r^{\infty}_{NFDH} = 2$, $r_{FFDH} = 2.7$ und $r^{\infty}_{FFDH} = 1.7$ (Coffman 1984, and Coffman 1980a).

Die worst-case- performance-Analyse zeigt somit ein erstaunliches Resultat: Die BL-Methoden schneiden im schlimmsten Falls schlechter ab, als die Level-Methoden, obwohl diese letzteren nur Guillotinen-Schnitte zulassen.
Man könnte sich verschiedene Methoden überlegen, wie Level-Heuristiken verbessert werden könnten. Statt jedes Level links zu beginnen, könnte man alternierend die ungeradzahligen Levels links und die geradzahligen Levels rechts beginnen und dann die Levels zusammenschieben (keine Guillotinen-Schnitte mehr!). Eine andere Möglichkeit wäre, den nicht-belegten Platz zwischen den Levels zu nutzen und mit den kleinsten Rechtecken aufzufüllen. Auch eine Rotation um 90° könnte in Betracht gezogen werden. Es hängt stark vom Problem ab, ob solche Modifikationen Verbesserungen mit sich bringen. Wichtiger hingegen ist, dass sie die worst-case-Performance bei all diesen Varianten von Level-Heuristiken nicht verbessern (Coffman 1980a, p.825).
Shelf-Heuristiken sind on-line Varianten der Level-Heuristiken und besonders geeignet, wenn die Verteilung der Rechtecksgrössen bekannt ist. (siehe zu den vier

Varianten FFS, NFS, BFS, BAS Coffman p. 139ff in: Dyckhoff 1990, und Nelissen 1991 S.20).

Der **CLS-Algorithmus** (Coffman p.137:in: Dyckhoff 1991) sortiert die Rechtecke zunächst nach absteigender Breite (ist also ein off-line-Algorithmus). Alle Rechtecke breiter als B/2 werden mit der BLDW-Heuristik angeordnet. Die restlichen Rechtecke werden nun ab der erreichten Höhe am rechten Rand des Streifens in aufsteigender Breite gestapelt und als Keil senkrecht so weit hinuntergeschoben wie möglich (Abbildung 3-8).

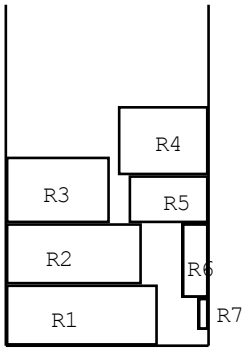


Abbildung 3-8

Der **UD-Algorithmus** (Baker al. 1981, Nelissen S.21) erreicht die beste worst-case Performance: 5/4. Die Rechtecke werden dabei in fünf Gruppen L_i, i={1..5} aufgeteilt. In den vier Gruppen i={1..4} befinden sich die Rechtecke, deren Breite im Intervall (1/(i+1, 1/i] liegt, (B wird auf eins normiert). Die Rechtecke werden nach absteigender Breite sortiert. In der Gruppe L_5 befinden sich alle Rechtecke deren Breite kleiner oder gleich als 1/5 ist. Diese werden nach absteigender Höhe sortiert. Die ersten vier Gruppen werden teils nach einer BL-Strategie teils nach der TR (top-left) Strategie - wie beim CLS-Algorithmus angeordnet. Die letzte Gruppe wird in die Zwischenräume nach einer NFDH-Strategie eingestreut. Ein exakter Beschreib des Algorithmus findet sich in Baker 1981.

Der **KLM-Algorithmus** wird in Karp al. 1984 beschrieben.

Zwei Spezialfälle

Das oben definierte Bin-Packing- und das Multiprocessor-Scheduling-Problem können als Spezielfälle des eben formulierten, zwei-dimensionalen Streifenzuschnittproblems angesehen werden. Wenn gefordert wird, dass die Höhen aller Rechtecke gleich sind, so erhält man das Bin-Packing-Problem, wenn umgekehrt gefordert wird, dass alle Breiten gleich sind, so erhält man das Multiprocessor-Scheduling-Problem.

Layout von Elementen auf Leiterplatten

Anordnungs oder Platzierungsprobleme, die eng verwandt sind mit den Zuschnittproblemen, entstehen auch auf andern Gebieten. Als Beispiel soll das Layout-Problem von Chips auf einer Leiterplatte gewählt werden. Auf einer rechteckigen Leiterplatte sollen *m* Elemente so angeordent werden, dass die gesamte Verbindungsstrecke zwischen den Elementen minimal wird. Sei *C* die (symetrische) Zusammenhangsmatrix, deren Elemente c_{ij} die Anzahl der elektrischen Verbindungen zwischen Element *i* und *j* angebe. Weiter sei (x_i,y_i) die Koordinate des Mittelpunkts eines Elements *i* auf der Platte (Wir nehmen vereinfachend die Elemente als kreisförmig an). Der **euklidische Abstand** zwischen zwei Elementen *i* und *j* ist definiert als $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ und der **orthogonale Abstand** als $|x_i - x_j| + |y_i - y_j|$. Da die Verbindungen orthogonal verlaufen sollen soll der Ausdruck $z = \sum_{i=1}^{m} \sum_{j=1}^{i-1} (|x_i - x_j| + |y_i - y_j|) c_{ij}$ minimiert werden, sodass die Koordinaten (x_i,y_i) innerhalb der Leiterplatte liegen. Dieses Problem kann durch die Simulationsmethode der kontrahierenden Umgebungen angenähert werden.

Ein ähnliches Problem ist das Problem der Anordnung von Elementen auf vorgegebenen Plätzen: Auf einer rechteckigen Leiterplatte der Breite m and der Länge n sollen N=n*m Elemente so angeordent werden, dass die geforderten Verbindungen minimal werden. Die N Plätze können in geeigneter Weise durchnumeriert werden. Die Permutation gibt dann alle möglichen Anordnungen. Jede Permutation ist daher Kandidat der optimalen Lösung.

4. GENETISCHE ALGORITHMEN

Für verschiedene Zuschnittprobleme wurden spezielle Greedy-Heuristiken vorgestellt. Eine weite Klasse von Heuristiken, die nicht nur auf Zuschnittprobleme anwendbar ist, sind verschiedene 'statistischen' Verfahren, wie *binde Suche*, *lokale Suche*, *Simulated Annealing*, *genetische Algorithmen* und andere. Es sind allgemeine Black-Box-Heuristiken, die auf sehr verschiedene Problemklassen angewandt werden können. Das eingangs definierte, allgemeine Optimierungsproblem $\{\text{minimiere } f(x): x \in S\}$ ist ausgezeichnet geeignet, um diese Algorithmen zu testen.

Um eine Kostprobe zu geben wählen wir zufällig ein Verfahren aus, welches eine Mischung zwischen lokaler Suche und Simulated Annealing ist: die Simulationsmethode der kontrahierenden Umgebungen (Terno S 133). Gegeben sei das allgemeine Optimierungsproblem $\{\text{minimiere } z = f(x): x \in S\}$. Vorgegeben wird eine Populationszahl *N* und ein Umgebungsmass *r* einer Lösung. Die Simulationsmethode zur Suche der besten Lösung *x* im Lösungsraum *S*, sodass *z* am kleinsten wird, kann dann folgendermassen geschrieben werden:

1. Erzeuge eine Startpopulation von zufälligen Lösungen
2. Die beste Lösung sei z°, setze z´=z°
3. Setze W´=0, ferner: falls z°<z´ setze z´=z°
4. Aus der Umgebung r von z° soll eine neue Population produziert werden
5. z° sei die beste Lösung in der neuen Population.
6. Mittelwert *m* und Standartabweichung *s* sollen berechnet werden durch

$$m = \frac{\sum_{i=1}^{N} z_i}{N} \text{ und } s = \sqrt{\frac{\sum_{i=1}^{N} z_i^2 - m^2 N}{N-1}} \text{ und } W = GAUSS(\tfrac{z^\circ - m}{s})$$

   wobei GAUSS() die gaussische Normalverteilung angibt.
7. Falls W < W´, verkleinere den Umgebungsparameter r (z.B. halbieren), ansonsten setze W´=W
8. Gehe zu Schritt 4, falls die Abbruchbedingung nicht erfüllt ist.

Der Rest dieses Papers soll den genetischen Algorithmen gewidmet sein. Es soll ein intuitiver Zugang zu diesem Verfahren gegeben werden, welches sich für zahlreiche Optimierungsprobleme ausgezeichnet bewährt hat, für andere Probleme, die eher kombinatorischer Art sind, anscheindend versagt. Es soll versucht werden, genetische Algorithmen auf das schwierige 2-dimensionale Zuschnittproblem anzuwenden. Da dieses Problem eher kombinatorischer Art ist, besteht der eigentliche Challenge darin, zu zeigen, dass genetische Algorithmen erfolgreich auch auf diese Probleme angewendet werden können.

BIOLOGISCHER HINTERGRUND

Genetische Algorithmen versuchen die Strategien und Mechanismen der biologischen Evolution zur Lösung von komplexen Problemen, z.B. Optimierungsproblemen, heranzuziehen. Daher sollen hier kurz die wichtigsten Begriffe und Mechanismen der natürlichen Evolution wiedergegeben werden.
In der biologischen Evolution sind die Spezien damit beschäftigt, eine Anpassung an die sich ständig verändernden Umweltbedingungen zu finden. Das 'Wissen' darum ist im Chromosomenstrang der Individuen abgespeichert. Dieser kann nur durch die Reproduktion verändert werden. Die Operationen dazu sind die zufällige Mutation, die Inversion und die Überkreuzung (crossover) (bei der sexuellen Reproduktion). Zufällige Mutationen bringt Variation in den Code und erlaubt, neue Anpassungsräume abzusuchen. Inversion verschiebt Gene innerhalb von Chromosomen und erlaubt dadurch 'verwandte' Gene zu bündeln (cluster). Überkreuzung tauscht Genmaterial zwischen Elternchromosomen aus.

Zwei fundamentale Mechanismen erklären die natürliche, biologische Evolution: *die blinde, zufällige Suche* und ein ganz leichter, kaum spürbarer *Selektionsdruck*. Es ist wichtig zu sehen, dass *beide* Mechanismen zusammenwirken müssen, um die biologische Evolution erklären zu können. Die blinde Suche ist offenbar das einzige, effiziente Mittel, um den ganzen, zulässigen Anpassungsraum homogen abzusuchen. Dadurch können erst 'robuste' Individuen entstehen. Die blinde Suche ist für die biologische Evolution jedoch nicht hinreichend. Um das zu sehen, braucht man nur die Wahrscheinlichkeit zu berechnen, dass ein komplexes Protein oder gar ein DNS-Strang durch rein zufälliges Mischen von Atomen entsteht. Diese Wahrscheinlichkeit ist zwar nicht Null aber verschwindend klein. Selbst wenn die Atome seit der Entstehung des Universums in schneller Folge durchmischt worden wäre, so wäre diese Wahrscheinlichkeit immer noch verschwindend klein, so dass man von einem ausserordentlichen Zufall reden müsste, was als Wunder (im Sinne Dawkins) zu bezeichnen wäre. Um die Evolution zu erklären, können wir jedoch nicht auf dieses Wunder als Erklärungskomponente zurückgreifen; und wir müssen dies auch nicht, da ein zweiter Mechanismus, die Selektion, im Spiel ist. Die Selektion ist ein ausgezeichnetes Mittel, nicht 'lebensfähige' Individuen tendenziell zu eliminieren. Wenn Selektion jedoch allein am Werk wäre, so würde der Genpool schnell einmal auf das 'best angepasste' Individuum zusammenschmelzen und es wäre unerklärlich, wieso es so viele verschiedene Spezien gibt. Für eine ausführliche, profunde und packend geschriebene Diskussion der Evolutionstheorie aus biologischer Sicht kann das Buch von Dawkins (1987) empfohlen werden.

Genetische Algorithmen legen genau diese beiden Mechanismen zu Grunde, um in einem mathematische Lösungsraum eine optimale Lösung zu suchen. Was in der biologischen Evolution der 'lebensfähige' Anpassungsraum ist, ist hier der mathematische Lösungsraum. Der Chromosomenstrang entspricht einer Lösungsrepresentation und die 'Überlebensfunktion', welche die Überlebensfähigkeit eines Individuums misst, wird durch die Zielfunktion f(x) repäsentiert.
Die eigentliche Stärke von genetischen Algorithmen besteht darin, dass diese in einem grossen Lösungsraum relativ schnell gute Lösungen liefern können. Weniger geeignet ist dieser Ansatz hingegen, wenn aus Lösungen, die sich nahe beim Optimum befinden, das Optimum gefunden werden soll.

EIN EINFÜHRENDES BEISPIEL

Anhand eines einfachen Beispiels (Goldberg 1989) soll zunächst die Funktionsweise von genetischen Algorithmen (GA) kurz beschrieben werden. Gegeben sei die

Funktion $f(x)=x^2$ und ein Wertebereich von $x = [0,31]$ wobei x ganzzahlig sein soll. Es soll ein x so gefunden werden, welches die Funktion $f(x)$ maximiert. Wir haben also das spezielle Optimierungsproblem $\{\text{maximiere } f(x) = x^2 : x \in \{0\text{K } 31\}\}$. Um das Problem mit Hilfe von GA anzugehen, muss zuerst der Lösungsraum von x kodiert werden. Da der Lösungsraum 32 Elemente enthält, kann er in 5 Bits abgebildet werden, d.h. jeder 0-1-Kombination von 5 Bits kann einem Wert von x zugeordnet werden (Tabelle 4-1). Die Abbildung muss aber nicht bijektiv sein, eine injektive Abbildung genügt.

x	kodiert	f(x)
0	00000	0
1	00001	1
2	00010	4
..	..	
31	11111	961

Tabelle 4-1

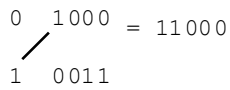
Die Abbildung kann beliebig sein, mit Vorteil wähle man eine einfache Kodierung, sodass die Transformation zwischen der internen Bitdarstellung und dem Wert von x einfach wird. In Tabelle 4-1 ist die binäre Darstellung von ganzen Zahlen im Computer gewählt worden, dies erleichert die Umwandlung. Aus jedem kodierten Wert kann der zugehörige Funktionswert f(x) berechnet werden.
Nun generieren wir zufällig eine 'Anfangspopulation' von Lösungen. Ein Beispiel einer Population von vier Individuen ist in Tabelle 4-2 wiedergegeben:

Individuum	Kodierung	x	f(x)	%-Fitness
1	01101	13	169	14.4
2	11000	24	576	49.2
3	01000	8	64	5.5
4	10011	19	361	30.9
Total			1170	100%

Tabelle 4-2

Aus diesem Genpool (Kodierung) von vier Individuen (Lösungen) sollen weitere Individuen produziert werden. Die Güte der Anpassung (die Fitnessfunktion f(x)) bestimmt die Wahrscheinlichkeit, mit welcher Individuen an der Reproduktion teilhaben und welche Individuen sterben, da die Grösse der Gesamtpopulation gleichbleiben soll. Es sind verschiedene Ausprägungen möglich, wie die Fitnessfunktion in die Generierung einer neuen Population eingegen kann. Um den Selektionsdruck nicht zu forcieren, und damit den Genpool nicht zu schnell zu homogenisieren, kann die Fitnessfunktion auch nur für die Sterbenswahrscheinlichkeit verwendet werden, während zur Reproduktion zufällig

ausgewählte Individuen gepaart werden können. Im Beispiel soll angenommen werden, dass die Individuen 3 und 4 in die Reproduktion eingehen und dass Individuum 2 (leider das Beste!) sterbe. Bei der Reproduktion werden die Kodierungen der beiden wie bei der biologischen sexuellen Reproduktion verschmelzt, z.B.



Das erste Bit wird vom Individuum 4 und die restlichen vom Individuum 3 genommen, der Rest geht verloren. Es fehlt nicht an Ironie des Schicksals, dass das gestorbene Individuum 2 dadurch wieder aufersteht! Das gesamte Vorgang reproduziert dieselbe Population. Nichts ist gewonnen, nichts ist verloren durch diesen 'Generationenwechsel'. Natürlich ist dies reiner Zufall. Man hätte auch die ersten beiden Bits vom Individuum 3 und die restlichen von 4 nehmen können und das neue Individuum wäre 01011 gewesen, was die mittlere Überlebensfähigkeit der gesamten Population ($\sum_{i \in Pop} \frac{f(x_i)}{N}$) sogar verringert hätte. Damit muss man rechnen!

In der natürlichen Selektion ist es nicht anders.

Da die Fitnessfunktion über den - unter Umständen. nur leichten - Selektionsdruck in die Reproduktion eingeht, ist es jedoch sehr wahrscheinlich, dass die mittlere Überlebensfähigkeit der Population *langfristig* wächst, d.h. dass die Population, im Sinne der Fitnessfunktion, aus immer besseren Individuen besteht. Dieser Druck muss beim Generationenwechsel vorhanden sein, darf aber nicht zu stark forciert werden, da sonst der Genpool schnell auf möglicherweise gute Lösungen (Individuen) zusammenschmelzt, die sich durch die Reproduktion nicht mehr verbessern können (hinkender Vergleich: Inzest!), obwohl in weit entlegenen Regionen des Lösungsraumes möglicherweise noch bessere Lösungen (Individuen) existieren.

Die Anfangspopulation wurde zufällig bestimmt. Es besteht kein Grund, den gesamten Verlauf der Generationen ausschliesslich von diesem Startgenpool bestimmen zu lassen. Durch Mutation (zufälliges Ändern eines Bits in einem Individuum bei der Reproduktion) kann die Diversifizierung aufrecht erhalten oder sogar vergrössert werden. Dies ist von ausschlagender Bedeutung, um den Lösungsraum homogen nach besseren Induviduen, die aus Sicht der Population in weit entfernten Regionen des Lösungsraumes angesiedelt sind, abzusuchen. Allerdings darf auch hier nichts forciert werden. Zu grosse Mutationsraten lassen den Algorithmus zu stark im Lösungsraum blind immer neue Individuen schaffen, welche die Überlebensfähigkeit nicht verbessern. Die Grössenordnung der Mutationsrate sollte im Bereich von einigen Promillen sein.

Wie in der natürlichen Evolution wird ersichtlich, dass die Selektion und die blinde Suche in den genetischen Algorithmen zusammenwirken müssen:

1. "learn while searching: gather global information about the space and use it to concentrate the search effort in the most promising regions."
2. "sustain exploration: the search strategy must be able to move to other regions of the space and continue searching." (Ackley: in Davis p.170)

KOMPONENTEN DER GA

Die Komponenten, welche einen genetischen Algorithmen ausmachen, können nun folgendermassen zusammengefasst werden (Davis 1987 S.2):

1. eine chromosomale Representation von Lösungen zum vorliegenden Problem
2. die Möglichkeit, eine Anfangspopulation von Lösungen zu generieren
3. ein Auswertungsfunktion, welche die 'Anpassungsfähigkeit' liefert
4. genetische Operatoren, welche die Chomosomen verändert
5. versch. Parameter, wie Populationsgrösse, Mutationsahrscheinlichkeiten, usw.

Der Algorithmus kann in der allgemeinsten Art folgendermassen beschrieben werden:

```
procedure GA;
begin
  initialize population P(0);
  evaluate P(0);
  t=1;
  repeat {generations}
    select P(t) from P(t-1)
    recombine P(t);
    evaluate P(t);
  until (termination condition)
end.
```

Ein vollständiger Code in PASCAL findet sich im Anhang (Program *SgA*), der im wesentlichen von Goldberg (1989) übernommen wurde.

Die Repräsentation (Kodierung)

Eine 'gute' Kodierung ist von ausschaggebender Bedeutung für den Lösungsverlauf. Im obigen Beispiel war die Kodierung einfach. In den meisten komplexen Problemen ist es jedoch unter Umständen schwierig, ein gute Kodierung zu finden. Man bezeichnet die Kodierung auch als **Genotyp**, da sie das genetische Material repräsentiert, in Gegensatz zum **Phenotyp**, welche die Manifestation des Genmaterials, d.h. das 'sichtbare' Individuum, nach der Ontogenese bezeichnet. Auch hier gibt es Parallelen zur Biologie: Ein Individuum, wenn es erst einmal gezeugt

wurde, vollzieht eine ontogenetische Entwicklung. Die Manifestation des Genmaterials ist die äussere Erscheinung des Individuums, die sich in einer Anzahl von *Erbattributen* niederschlägt (blaue Augen, starke Hände, usw.). Dieser Attribute bestimmen das weitere Schicksal des Individuums mit. Insbesonders begünstigen Attribute, welche der Umwelt besser angepasst sind, das Überleben des Individuums; andere, welche den Besonderheiten der Umwelt wenig Rechnung tragen, beeinträchtigen das Überleben tendenziell. Dies muss sich natürlich bei der Reproduktion niederschlagen. Dadurch vererben Individuen ihre *Attribute*.
Dieser Mechanismus kann und sollte bei der Implementierung von genetischen Algorithmen für ein Problem berücksichtigt werden. Im einleitenden Beispiel ist der Genotyp eine Menge von fünf Bits. Der Phenotyp äussert sich in einem einzigen Erbattribut, dem Funktionswert f(x). Die Auswertung der Funktion f *entspicht* - wenn hier die Metapher weitergesponnen wird - der ontogenetischen Entwicklung des Individuums in der biologischen Entwicklung.

An einem komplexeren Beispiel, dem Traveling-Salesperson-Problem (TSP) soll diese Beziehung nochmals dargestellt werden. Das TSP-Problem besteht darin, dass ein Vertreter n Städte bei einer Rundfahrt je genau einmal besuchen soll, wobei die gefahrene Strecke minimal sein soll. Der Lösungsraum besteht aus allen Permutationen der Städte, also aus *n!* Möglichkeiten. Bei nur 100 Städten sind das immerhin bereits $9.3 \cdot 10^{157}$ Möglichkeiten! Bei 8 Städten, die von 1 bis 8 durchnumeriert sind, wäre eine mögliche Rundreise z.B. 1-2-3-4-5-6-7-8. Insgesamt sind 8!=40320 solche Rundreisen möglich und eine davon ist die kürzeste. Ein wichtiges Erbattribut ist sicher die Weglänge einer Permutation. Leider benügt dies nicht. 'Gute' Teilstrecken sollten auch als Attribute vererbt werden können.

Eine Kodierung für dieses Problem soll nun befunden werden. Zuerst soll eine Kodierung gegeben werden, welche zwar naheliegend, aber zur Lösung des Problems unbrauchbar ist, gerade weil diese Kodierung die Übertragung der guten Teilstrecke als Attribut nicht erlaubt. Die zweite Kodierung soll diesen Aspekten besser Rechnung tragen. Dieser Punkt ist fundamental, wenn genetische Algorithmen auf verschiedene Probleme erfolgversprechend angewendet werden wollen. Zu sehr wurde in der Literatur für die verschiedensten Probleme auf die blinde Anwendung der Bit-String-Kodierung und deren Operatoren des Erfinder von genetischen Algorithmen (Holland 1975) zurückgegriffen. Am TSP-Problem kann leicht gezeigt werden, dass ein solcher Unterfangen scheitern muss (siehe Kodierung 1 unten)

Kodierung 1 des TSP-Problems:

Der gesamte Lösungsraum könnte in $\log_2(n!)$ Bits [$\approx n*\log_2(n)$] abgebildet werden, das sind bei n=100 minestens 525 Bits oder 66 Bytes. Das Problem besteht nun darin, aus diesen 525 Bits, die man sich als binäre Darstellung einer Zahl im Intervall [0...$9.3\cdot10^{157}$] vorzustellen hat, eine konkrete Permutation zu generieren. Für kleine n (<12) kann dafür die Prozedur *RankPInv()* (siehe Anhang) verwendet werden. (Die inverse Operation ist in der Prozedur *RankP()* realisiert.) Für unser Beispiel bei 8 Städten verwenden wir eine Kodierung aus 16 Bits ($8!=40320>2^{15}$).
Zwei Permutationen seien gegeben:

{ 1, 3, 7, 4, 5, 8, 2, 6 } und { 2, 3, 8, 5, 1, 6, 7, 4 }.

Durch Umwandlung mit Hilfe der Prozedur *RankP()* werden daraus die Zahlen 1234 und 6411. Diese entsprechen in der binären Darstellung den Zahlen 0000'0100'1101'0010 und 0001'1001'0000'1011. Dies sei die chromosomale Darstellung der beiden Permutationen. Diese beiden Individuen gehen jetzt in die Reproduktion ein: es sollen die erste 7 Bit des ersten und die restlichen 9 Bits des zweiten Bitstroms genommen werden. Daraus entsteht das binäre Zahl 0000'0101'0000'1011 oder in dezimaler Darstellung die Zahl 1291. Diese Zahl entspricht der Permutation { 1, 3, 7, 6, 8, 2, 5, 4 } (durch Umwandlung mit Hilfe von RankPInv()). Die Schwierigkeit dieser Kodierung besteht also darin, dass diese chromosomale Darstellung einer Permutation sein (Teilstrecken-)Attribut durch die Reproduktion nicht fortpflanzen kann. Das neue Individuum erbt zwar die Teilstrecke 1-3-7 vom ersten Individuum, das zweite Individuum kann jedoch keine Teilstrecke beitragen - die Reproduktion durch zwei Individuen bringt hier daher nichts und ist völlig unnötig!

Kodierung 2 für das TSP-Problem:
Eine bessere Art, die Kodierung vorzunehmen, besteht darin, die interne, binäre Darstellung der Zahlen in der Permutation direkt für die Kodierung zu verwenden. Jede Zahl soll dabei durch vier Bits dargestellt werden (dies genügt für unser Beispiel mit n=8). Die beiden Permutationen

{ 1, 3, 7, 4, 5, 8, 2, 6 } und { 2, 3, 8, 5, 1, 6, 7, 4 }.

würden dann durch folgende Bitströme dargestellt:

0001'0011'0111'0100'0101'1000'0010'0110 und
0010'0011'1000'0101'0001'0110'0111'0100

Wird an der zufälligen Stelle 21 gekreuzt, so entstehen die Individuen

0001'0011'0111'0100'0101'1110'0111'0100
0010'0011'1000'0101'0001'0000'0010'0110

welche den illegalen Permutationen

{ 1, 3, 7, 4, 5, 14, 7, 4 } und {2, 3, 8, 5, 1, 0, 2, 6}

entsprechen. Diese Kodierung stellt uns vor ein Problem: der Überkreuzungsoperator dieser Kodierung produziert illegale Individuen. Verschiedene Möglichkeiten bestehen, um diese Situation zu beheben: Die Individuen können 'repariert' werden oder es muss ein neuer Überkreuzungsoperator eingeführt werden. Mit verschiedenen Überkreuzungsoperatoren, welche unten kurz eingeführt werden, wurde experimentiert. Der Vorteil dieser Kodierung besteht allerdings darin, dass beide Individuen Eigenschaften ihres Erbmaterials übertragen können, da Teilstrecken von beiden übernommen werden.

In der Kodierung liegt die wesentliche Schwierigkeit bei der Implementierung von genetischen Algorithmen. Verschiedene Möglichkeiten der Kodierung wurden erforscht: geordnete Listen (ordered lists) für das Bin-Packing-Problem, eingebettete Listen (embedded lists) für das Scheduling-Problem, variable Elementenlisten für das Layoutproblem von Leiterplatten. Alle diese Darstellungen verlangen natürlich spezielle Reproduktionsoperatoren.

Die Startpopulation

Die Startpopulation kann verschieden generiert werden. Sie kann aus zufälligen Lösungen gebildet werden, oder es können verschiedene Greedy-Verfahren Lösungen liefern. Diese können wiederum 'verrauscht' werden, um 'verwandte' Individuen (Lösungen in der Umgebung) zu generieren. Man sollte sich jedoch darüber im Klaren sein, dass eine Startpopulation, welche aus einem Verfahren 'gute' Individuen generiert, nicht unbedingt nur von Vorteil ist. Wenn der Genpool zu homogen gewählt wird, kann es leicht vorkommen, dass der Algorithmus im Lösungsraum in einem lokalen Optimum 'festfährt', ohne den weiteren Raum abzusuchen. Die Suche nach dem globalen Optimum muss daher scheitern.

Die Auswertungsfunktion (Fitnessfunktion)

Die Fitnessfunktion entscheidet über die Güte einer Lösung. Um brauchbar zu sein, muss sie jedoch meist normalisiert (transformiert) werden. Der Grund besteht darin, dass gute und schlechte Lösungen oft nur wenige 'Punkte' voneinander abweichen und der Algorithmus durch die rohe Auswertung der Fitnessfunktion nicht genügend zwischen guten und schlechten Werten unterscheiden kann. Der genetische Algorithmus reagiert sehr sensitiv auf Normalisierungen der Fitnessfunktion. Wenn bei der Normalisierung zu starkes Gewicht auf die Unterscheidung der Individuen gelegt wird, kann es leicht vorkommen, dass schlechte Individuen zu schnell eliminiert werden und der Genpool rapide zusammenschrumpft, die Reproduktion wird dann immer unwichtiger. Werden hingegen bei der Normalisierung die

Unterschiede nicht genügend betont, so kann der Algorithmus zu lange erratisch im Lösungsraum immer neue gleichwertige Lösungen finden, ohne dass die Selektion zum Tragen kommt, d.h. ohne dass der Algorithmus zu guten Individuen konvergiert. *Diese Balance zwischen zufälligem Suchen und Selektion scheint überhaupt ein tragendes Element zu sein für eine erfolgversprechene Implementierung von genetischen Algorithmen.*

> "Searching a complex space of problem solutions often involves a tradeoff between two apparently conflicting objectives: exploring the best solutions currently available and robustly exploring the space." (Booker L. S.61, in :Davis 1987).

Die Abbilding 4.3 wiedergibt die Auswertung aller Permutationen eines TSP mit n=6. Vertikal sind die Werte der Optimierungsfunktion (=Wegstrecke der Rundreise) und horizontal alle 6!=720 Permutationen abgetragen. In der oberen Kurve sind die Permutationen lexikographisch geordnet. In der untere Kurve wurden die Permutationen nach der Auswertungsfunktion sortiert. Die Werte schwanken zwischen 31 und 101. Da es 720 Lösungen gibt, sind im Schnitt 10 Lösungswert identisch. In diesem Beispiel ist der Lösungsraum klein, für eine grossen Lösungsraum verschäft sich das Problem, dass viele Individuen voneinander durch die Zielfunktion nicht mehr zu unterscheiden sind. Die vier besten Lösungen sind die Rundreisen: 1-6-3-5-4-2, 1-6-4-2-3-5, 1-6-4-3-2-5, und 1-6-5-4-3-2 mit den Zielwerten 31, 36, 37, 37.

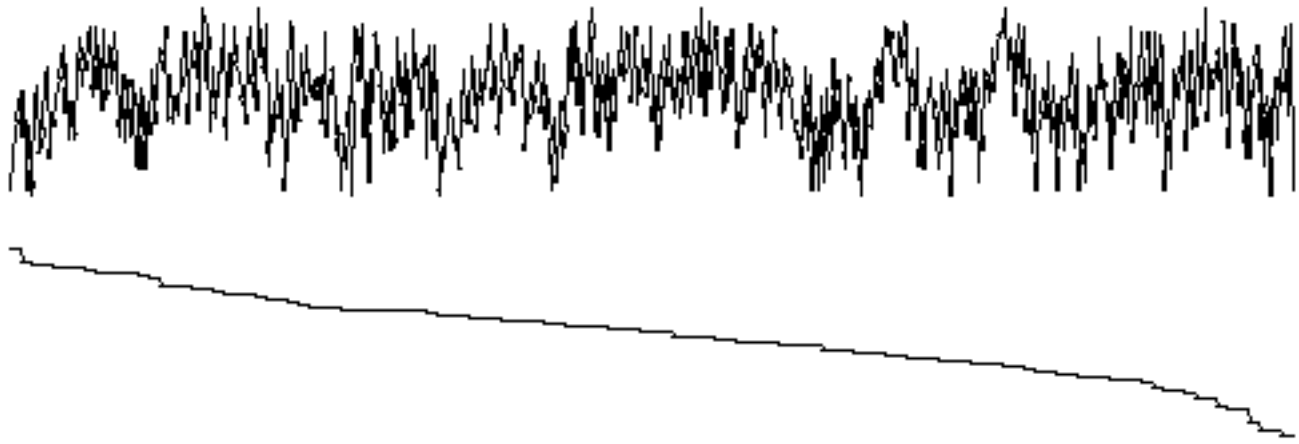


Abbildung 4-3

Dass die Fitnessfunktion das einzige Kriterium der Selektion ist, kann als ein Vorteil und als ein Nachteil betrachtet werden. Ein Vorteil ist es deshalb, weil der genetische

Algorithmus unabhängig von der Problemstellung implementiert werden kann und daher ein allgemeiner Black-Box-Problemlösungsalgorithmus ist. Die Lösungen sind robust in dem Sinne, dass der Lösungsverlauf unabhängig von der Problemstellung ist. Ein Nachteil ist dies sicher, wenn in einer Problemstellung die Güte der Fitnessfunktion nicht das einzige Attribut ist, welches ein Individuum vererben können sollte.

Eine wichtige Frage besteht darin, wie zusätzliche Modelrestriktionen gehandhabt werden. Restriktionen, die nicht verletzt werden dürfen, könnten so implementiert werden, dass Individuen mit einem grossen Strafwert (penalty) gelegt werden. Der Strafwert wird dann vom Fitnesswert abgezogen. Wenn der Lösungsraum so beschaffen ist, dass die Verletzung einer Restriktion wahrscheinlich ist, so führt ein solches Verfahren natürlich dazu, dass der Algorithmus die meiste Zeit damit verbringt, nicht lebensfähige (=illegale, welche die Restriktionen verletzt) Individuen zu eliminieren. Zudem kann es passieren, dass, sobald ein legales Individuum gefunden ist, alle andern aus der Population verdrängt werden und der Algorithmus in der Umbegung dieser Lösung steckenbleibt. Zu kleine Strafwert hingegen begünstigen illegale Individuen, die unter Umständen sogar besser abschneiden als legale Individuen, welche einen schlechten Fitnesswert haben.
Eine andere Möglichkeit, Restriktionen einzubeziehen, besteht darin, einen problemspezifischen Dekoder so in die Evaluation einzubauen, dass illegale Individuen schon gar nicht entstehen können. Diese Lösung ist allerdings meist sehr aufwendig und die Zeitkomplexität des Algorithmus kann stark anwachsen, ganz abgesehen davon, dass es oft schwierig ist, verschiedene Restriktionen in den Dekoder einzubauen.

Die genetischen Operatoren

Die genetischen Operatoren hängen sehr von der Kodierung ab. Werden die Individuen als Bit-Strings implementiert, so können die drei klassischen Operatoren angewandt werden: Single-Point-Crossover, Mutation eines Bits mit einer bestimmten Mutationsrate (z.B. 8‰) und Inversion (eher selten). Andere Operatoren sind möglich, wie two (or more)-point-crossover, uniform-crossover, usw.
Werden andere Kodierungen verwendet, so müssen die Operatoren angepasst werden. Der Crossover-Operator auf die zweite Kodierung von Permutationen (siehe oben), könnte z.B. folgendermassen definiert werden (Grefenstette, in: Davis 1987, S. 50):

1. Wähle aus der erste Permutation zufällig eine Teilstrecke T aus.
2. Entferne alle Elemente dieser Teilstrecke aus der zweiten Permutation und konkateniere die verbleibenden Elemente zur Teilstrecke S

3. Konkateniere beide Teilstrecken T und S.

Dieser Operator hat den Vorteil, dass gewisse Teilstrecken der Elternpermutation auf die Kinder übergeht. Ideal ist er allerdings nicht, denn viele Verbindungen in der Elternpermutationen werden zufällig aufgelöst und anders rekombiniert. Dies kommt einer hohe, versteckte Mutationsrate gleich. Experimente zeigen (Grefenstette), dass der genetische Algorithmus für das TSP-Problem unter Verwendung dieses Crossover-Operators eher erratisch im Lösungsraum herumirrt, als dass er gute Lösungen generiert.
Ein weiterer Operator ist der PMX-Operator von Goldberg (in: Greffenstette 1985, S. 154). Die Prozedur *PMX_Crossover()* im Anhang implementiert diesen Operator.

Ein 'idealer' Operator sollte Kinder so produzieren, dass ausschliesslich Verbindungen, welche in einem Elternteil vorkommen, verwendet werden. Dieser Forderung nahe kommt der Edge-Recombination-Operator (Whitley D al., in: Davis 1991, p. 352ff).

Die Wahl einer geeigneten Kodierung zusammen mit den passenden Operatoren ist von ausschlaggebender Bedeutung für die Güte eines genetischen Algorithmus. Die richtige Kombination zu finden, erfordert Erfahrung und Fingerspitzengefühl.

Die Parameter

Alles kann scheitern an einer falschen Parameterwahl. Welche Populationsgrösse soll gewählt werden? Wie gross soll die Mutationsrate sein? Sollen die Parameter konstant gehalten oder über die Generationen variert werden? Verschiedene Versuche müssen durchgeführt werden, bevor die Parameter festgelegt werden können. Die Parametersuche kann selbst durch einen genetischen Algorithmus implementiert werden. Versuche in dieser Hinsicht sind ermutigend: verschiedene genetische Algorithmen fanden gute Parametereinstellungen.

GENETISCHE ALGORITHMEN ANGEWANDT AUF DAS STREIFENPROBLEM

Nach dieser kurzen, informellen Einführung in die Denkweise der genetischen Algorithmen, sollen nun stichwortartig, verschiedene Anregungen gegeben werden, um das 2-dimensionale Streifenzuschnittproblem mit Hilfe dieses Ansatzes angehen zu können.

1. Kodierung:

Erste Möglichkeit: Eine Platzierung soll vorgegeben sein als eine Permutation der Rechtecke und einem boolschen Vektor, worauf die BLR-Heuristik angewandt wird. Wenn die Rechtecke um 90° gedreht werden können, so muss ein zweiter boolsche Vektor gegeben werden, welcher die Richtung des entsprechenden Rechtecks angibt. Die Kodierung kann dann durch eine Permutation und zwei Bit-Strings repräsentiert werden. (Eine ähnliche Methode wurde offenbar schon von Smith (in:Grefenstette 1985, S. 202ff) für das 2-dimensionale Packungsproblem versucht).
Zweite Möglichkeit: Die Kodierung wird als Baum implementiert, wie in Kröger S. 82ff beschrieben.
Dritte Möglichkeit: Die Platzierung ist gegeben als eine Liste von folgenden vier Primitiven, wobei *p* und *q* zwei Rechtecke repräsentieren: *p ist_oberhalb_von q*, *p ist_unterhalb_von q*, *p ist_rechts_von q*, und *p ist_links_von q*. Damit haben wir n^2-n Elemente in der Liste. Die Liste könnte verkürzt werden, wenn die vier primitiven Operationen auf die Interpretation *grenzt_unmittelbar_an* eingeschränkt würden. (Fourman, in Grefenstette 1985, S. 147)

2. Operatoren:
Für die erste Kodierung kommen als Crossover-Operator der PMX und der Edge-Rekombination-Operator in Frage. Für die andern Kodierungen müssen neue Operatoren gefunden werden. Der Mutationsoperator und die Mutationsrate sind ebenfalls zu bestimmen. Als Variante könnte der klassische 2-opt-Operator für TSP Probleme eingeführt werden.

3. Population:
Die Populationsgrösse soll zunächst auf 100 begrenzt sein. Es muss getestet werden, ob die gesamte Population beim Generationenwechsel auszutauschen ist oder nur ein Teil. Die Fragen, welche Individuen in die Reproduktion eingehen und welche sterben, sind völlig offen. Da es relativ schwierig sein wird, einen mutationsfreien Reproduktionsoperator zu finden, spricht intuitiv einiges dafür, dass eine elitaristische Strategie (das oder die besten Individuen sterben nie, die schlechten sterben) anzuwenden ist. Die Selektion kann auf der Basis des Fitnesswertes erfolgen oder die Individuen können zufällig ausgewählt werden und die Fitness hat nur einen Einfluss auf die Sterbewahrscheilichkeit.

4. Fitnessfunktion:
Der heikelste Punkt wird der Fitnesswert sein. Ein Streifenpackung ist umso besser, wenn die dazu verwendete Höhe des Streifens kleiner ist. Eine Fitnessfunktion, welche nur auf der erreichten Höhe basiert, wird aber sehr flach ausfallen, d.h. viele

verschiedene Packungen ergeben dieselbe Höhe. Die Individuen können nicht mehr unterschieden werden. Die Fitnessfunktion muss erweitert werden. Verschiedene Kriterien von guten Packungen bieten sich an: Packungshöhe der Hälfte aller Rechtecke, Verschnitt nachdem die Hälfte aller Rechtecke platziert sind, Bewertung der Schnittkantenlänge (Kröger S. 84). Die entgültige Fitnessfunktion hat verschiedene Kriterien in geeigneter Weise zu kombinieren.

ANHANG: PROZEDUREN

In diesem Anhang sollen einige Prozeduren, geschrieben in THINK-PASCAL, für die Implementierung verschiedener Aspekte aufgelistet werden.

KNAPSACK01 UND KNAPSACKB

Die folgenden Prozeduren implementieren das 0-1- und das Bounded-Knapsack-Problem. Der Code in FORTRAN kann in Martello & Toth gefunden werden. Die Procedure MT1 implementiert das 0-1-Knapsack-Problem. Die mitgegebenen sind die Werte *p*, die Gewichte *w*, das Gesamtgewicht *C*, die Anzahl Elemente *n* und die Dimension des Arrays *JDIM*. Die Prozedur gibt einen Vektor *x* bestehend aus 0 und 1 zurück, sowie den Zielwert *z*. Die Prozedur MTB2 löst das bounded-Knapsack-Problem. Die mitgegebenen Parameter sind die Vektoren *P*, *W*, , *B*, sowie die ganzen Zahlen *N*, *C* und *JDIM*. Das Resultat wird in *X* und *Z* zurückgegeben. Das Bounded-Knapsack-Problem wird durch die Prozedur *TRANS* in ein 0-1-Knapsack-Problem umgewandelt dann mit *MT1* gelöst und durch die Prozedur *SOL* wieder zurücktransformiert.

```
const
 MAXT = 100;

type
 TVektor = array[0..MAXT] of integer;
 Treal = array[0..100] of real;
 T01Knapsack = record
   n, c, z, TMAX: integer;
   P, W, X: TVektor;
  end;
 TBKnapsack = record
   n, c, z, TMAX: integer;
   B, P, W, X: TVektor;
  end;

procedure WriteV (var V: TVektor; n: integer);
 var
  i: integer;
begin
 for i := 1 to n do begin
   write(V[i] : 1, ',');
  end;
 writeln;
end;

 procedure MT1 (var P, W, X: TVektor; var Z: integer; N, C, JDIM: integer);
{-- implements 0-1 knapsack }
```

```
  label  20, 50, 170, 90, 280, 80, 160, 150, 110, 190, 250, 210, 240, 220, 230,
   270, 300, 310, 320, 330, 340, 370;
  var  CHMT1, IP, LL, LIM, LIM1, MINK, J, KK, LOLD, II, NN, II1, IIN, IU, J1,
   JJ, N1, CH, CHS, DIFF, PROFIT, R, T, A, B: integer;
   XX, MIN, PSIGN, WSIGN, ZSIGN: TVektor;
 begin
  Z := 0;  CH := C;  IP := 0;  CHS := CH;
  for LL := 1 to N do begin
    if (W[LL] > CHS) then  goto 20;
    IP := IP + P[LL];
    CHS := CHS - W[LL];
   end;
20:  LL := LL - 1;
  if (CHS = 0) then  goto 50;
  P[N + 1] := 0;
  W[N + 1] := CH + 1;
  LIM := IP + CHS * P[LL + 2] div W[LL + 2];
  A := W[LL + 1] - CHS;
  B := IP + P[LL + 1];
  LIM1 := B - trunc(A * P[LL] / W[LL]);
  if (LIM1 > LIM) then  LIM := LIM1;
  MINK := CH + 1;
  MIN[N] := MINK;
  for J := 2 to N do begin
    KK := N + 2 - J;
    if (W[KK] < MINK) then  MINK := W[KK];
    MIN[KK - 1] := MINK;
   end;
  for J := 1 to N do  XX[J] := 0;
  Z := 0;
  PROFIT := 0;
  LOLD := N;
  II := 1;
  goto 170;
50:  Z := IP;
  for J := 1 to LL do  X[J] := 1;
  NN := LL + 1;
  for J := NN to N do  X[J] := 0;
  EXIT(MT1);
80:  if (W[II] <= CH) then  goto 90;
  II1 := II + 1;
  if (Z >= CH * P[II1] / W[II1] + PROFIT) then  goto 280;
  II := II1;
  goto 80;
90:  IP := PSIGN[II];
  CHS := CH - WSIGN[II];
  IIN := ZSIGN[II];
  for LL := IIN to N do begin
    if (W[LL] > CHS) then  goto 160;
    IP := IP + P[LL];
    CHS := CHS - W[LL];
   end;
  LL := N;
110:  if (Z >= IP + PROFIT) then  goto 280;
  Z := IP + PROFIT;
  NN := II - 1;
  for J := 1 to NN do  X[J] := XX[J];
  for J := II to LL do  X[J] := 1;
  if (LL = N) then  goto 150;
  NN := LL + 1;
  for J := NN to N do  X[J] := 0;
150:  if (Z <> LIM) then  goto 280;
  EXIT(MT1);
160:  IU := CHS * P[LL] div W[LL];
  LL := LL - 1;
  if (IU = 0) then  goto 110;
  if (Z >= PROFIT + IP + IU) then  goto 280;
170:  WSIGN[II] := CH - CHS;
  PSIGN[II] := IP;
  ZSIGN[II] := LL + 1;
  XX[II] := 1;
  NN := LL - 1;
```

```
  if (NN < II) then  goto 190;
  for J := II to NN do begin
    WSIGN[J + 1] := WSIGN[J] - W[J];
    PSIGN[J + 1] := PSIGN[J] - P[J];
    ZSIGN[J + 1] := LL + 1;
    XX[J + 1] := 1;
   end;
190:  J1 := LL + 1;
  for J := J1 to LOLD do begin
    WSIGN[J] := 0;
    PSIGN[J] := 0;
    ZSIGN[J] := J;
   end;
  LOLD := LL;
  CH := CHS;
  PROFIT := PROFIT + IP;
  if (LL - (N - 2) < 0) then  goto 240
  else if (LL - (N - 2) = 0) then  goto 220 else  goto 210;
210:  II := N;
  goto 250;
220:  if (CH < W[N]) then  goto 230;
  CH := CH - W[N];
  PROFIT := PROFIT + P[N];
  XX[N] := 1;
230:  II := N - 1;
  goto 250;
240:  II := LL + 2;
  if (CH >= MIN[II - 1]) then  goto 80;
250:  if (Z >= PROFIT) then  goto 270;
  Z := PROFIT;
  for J := 1 to N do  X[J] := XX[J];
  if (Z = LIM) then  EXIT(MT1);
270:  if (XX[N] = 0) then  goto 280;
  XX[N] := 0;
  CH := CH + W[N];
  PROFIT := PROFIT - P[N];
280:  NN := II - 1;
  if (NN = 0) then  EXIT(MT1);
  for J := 1 to NN do begin
    KK := II - J;
    if (XX[KK] = 1) then  goto 300;
   end;
  EXIT(MT1);
300:  R := CH;
  CH := CH + W[KK];
  PROFIT := PROFIT - P[KK];
  XX[KK] := 0;
  if (R < MIN[KK]) then  goto 310;
  II := KK + 1;
  goto 80;
310:  NN := KK + 1;
  II := KK;
320:  if (Z >= PROFIT + CH * P[NN] / W[NN]) then  goto 280;
  DIFF := W[NN] - W[KK];
  if (DIFF < 0) then  goto 370  else if DIFF = 0 then  goto 330  else  goto 340;
330:  NN := NN + 1;
  goto 320;
340:  if (DIFF > R) then  goto 330;
  if (Z >= PROFIT + P[NN]) then  goto 330;
  Z := PROFIT + P[NN];
  for J := 1 to KK do  X[J] := XX[J];
  JJ := KK + 1;
  for J := JJ to N do  X[J] := 0;
  X[NN] := 1;
  if (Z = LIM) then  EXIT(MT1);
  R := R - DIFF;
  KK := NN;
  NN := NN + 1;
  goto 320;
370:  T := R - DIFF;
  if (T < MIN[NN]) then  goto 330;
  if (Z >= PROFIT + P[NN] + T * P[NN + 1] / W[NN + 1]) then  goto 280;
```

```
  CH := CH - W[NN];
  PROFIT := PROFIT + P[NN];
  XX[NN] := 1;
  II := NN + 1;
  WSIGN[NN] := W[NN];
  PSIGN[NN] := P[NN];
  ZSIGN[NN] := II;
  N1 := NN + 1;
  for J := N1 to LOLD do begin
    WSIGN[J] := 0;
    PSIGN[J] := 0;
    ZSIGN[J] := J;
   end;
  LOLD := NN;
  goto 80;
 end;

 procedure SOL (var B, XT, X: Tvektor; N, JDIM: integer);
{ translater from 0-1-Knapsack to bounded-form }
  label  10, 20;
  var  NT, J, ISUM, ID: integer;
 begin
  NT := 0;
  for J := 1 to N do begin
    ISUM := 0;
    ID := 1;
    X[J] := 0;
10:    NT := NT + 1;
    ISUM := ISUM + ID;
    X[J] := X[J] + ID * XT[NT];
    ID := ID * 2;
    if (ID + ISUM <= B[J]) then  goto 10;
    if (ISUM = B[J]) then  goto 20;
    ID := B[J] - ISUM;
    NT := NT + 1;
    X[J] := X[J] + ID * XT[NT];
20:   end;
 end;

 procedure TRANS (var P, W, B, PT, WT: TVektor; var NT: integer; N, JDIM:
integer);
{ translates bounded-Knapsack in a 0-1-form }
  label  30, 10, 20;
  var  JDMAX, J, ISUM, ID: integer;
 begin
  JDMAX := JDIM - 3;
  NT := 0;
  for J := 1 to N do begin
    ISUM := 0;
    ID := 1;
10:    NT := NT + 1;
    if (NT > JDMAX) then  goto 30;
    PT[NT] := P[J] * ID;
    WT[NT] := W[J] * ID;
    ISUM := ISUM + ID;
    ID := ID * 2;
    if (ID + ISUM <= B[J]) then  goto 10;
    if (ISUM = B[J]) then  goto 20;
    ID := B[J] - ISUM;
    NT := NT + 1;
    if (NT > JDMAX) then  goto 30;
    PT[NT] := P[J] * ID;
    WT[NT] := W[J] * ID;
20:   end;
  EXIT(TRANS);
30:  NT := -5;
 end;

procedure MTB2 (var P, W, X, B: TVektor; var Z: integer; N, C, JDIM: integer);
{-- bounded knapsack }
  label  10;
  var
```

```
   RD8: Treal;
   ID1, ID2, ID3, ID4, ID5: TVektor;
   ID6, ID7, NT: integer;
 begin
  Z := 0;
  TRANS(P, W, B, ID1, ID2, NT, N, JDIM);
  if (NT > 0) then  goto 10;
  Z := -5;
  EXIT(MTB2);
10:  MT1(ID1, ID2, ID3, Z, NT, C, JDIM);
  SOL(B, ID3, X, N, JDIM);
end;
```

GENVARIANTS

GenVariants generiert alle Varianten eines 1-1-Zuschnittproblems. Das Resultat wird in der Matrix *a[]* zurückgegeben.

```
const TMAX = 100;
type
  TVektor = array[0..TMAX] of integer;
  TMatrix = array[0..TMAX, 0..TMAX] of integer;

procedure GenVariants (l: TVektor; m, La: integer; var a: TMatrix; var n:
integer);
   { generates all variantes for the cutting-stock problem : Terno S. 60 }
   { input is l,La,m; output is a,n }
 var
   i, j, p, k, t, R: integer;   Z, v: TVektor;
begin
 k := 1;  i := 1;  t := 0;  R := La;

 while true do begin
   repeat
    a[i, k] := R div l[i];
    if (a[i, k] > 0) and (i < m) then begin
       t := t + 1;   Z[t] := i;   R := R - a[i, k] * l[i];
     end;
    i := i + 1;
   until i > m;

   v[k] := R - a[m, k] * l[m];
   if t = 0 then begin
      n := k;   exit(GenVariants);
    end;

   k := k + 1;   j := Z[t];
   for p := 1 to j - 1 do   a[p, k] := a[p, k - 1];
   a[j, k] := a[j, k - 1] - 1;
   R := R + l[j];
   i := j + 1;
   if a[j, k] = 0 then  t := t - 1;
  end;
end;
```

RANKP AND RANKPINV

Die beiden Funktionen produzieren Mappings zwischen ganzen Zahlen und Permutationen bestehend aus n Elementen. Mit n=4 haben wir z.B. folgendes Mapping:

Zahl	Permutation

```
0       1 2 3 4
1       1 2 4 3
2       1 3 2 4
...     ...
23      4 3 2 1
```

```
type  TPerm = array[0..255] of byte;  { table structure to permute }

procedure InitV (var v: TPerm);
  var  i: integer;
begin
  for i := 0 to 255 do v[i] := 0;
end;

procedure RankP (n: longint; var d: longint; var p: TPerm);
 { d is a binary representation of a permutation p containing n elements }
 { given a permutation p and n, this returns d (inverse function of RankPInv) }
 {---- ref.: AKL S.G., S.146 }
 var
  i, j, k, a, b: longint;
  s: TPerm;
begin
  for i := 1 to n do begin
    d := -i;
    for j := 1 to i - 1 do
      if p[i] < p[j] then  d := d + 1;
    s[i] := p[i] + d;
  end;
  d := s[n];
  i := 1;
  for j := n - 1 downto 1 do begin
    i := (n - j) * i;   d := d + s[j] * i;
  end;
  {d := d + 1;}
end;

procedure RankPInv (n, d: longint; var p: TPerm);
 { d is a binary representation of a permutation p containing n elements }
 { given d and n, this returns a permutation p (inverse function of RankP) }
 {---- ref.: AKL S.G., S.146 }
 var
  i, j, k, a, b: longint;
  s: TPerm;
begin
  {d := d - 1;}
  InitV(s);
  a := 1;
  for i := n - 1 downto 1 do  a := a * i;
  for i := 1 to n do begin
    b := trunc(d / a);
    d := d - a * b;
    if n > i then  a := a div (n - i);
    k := 0;
    j := 0;
    while k < b + 1 do begin
      j := j + 1;
      if s[j] = 0 then  k := k + 1;
    end;
    p[i] := j;
    s[j] := 1;
  end;
end;
```

RANDOM UNIT

Dieses Programm wird im genetischen Algorithmus *SgA* verwendet. Es liefert den benötigten Zufallsgenerator. Das Unit besteht aus vier Funktionen. Die Prozedur

randSeeder(x) initialisiert den Zufallsgenerator, der mitgegebene Parameter *x* muss eine Zahl in Bereich] 0,1 [sein. Die Funktion *rand01* liefert eine uniforme Zufallszahl im Intervall] 0,1 [, während *randInt(a,b)* eine ganzzahlige, uniform verteilte Zufallszahl im Intervall] a, b [zurückgibt. Die Funktion *flip(p)* gibt mit einer bestimmten Wahrscheinlichkeit *p* TRUE und mit der Gegenwahrscheinlichkeit *1-p* FALSE zurück.

```
unit Random;

interface
  function rand01: real;   { uniform random in [0,1] }
  function flip (probability: real): boolean;
  function randInt (low, high: integer): integer;
  procedure randSeeder (RandomSeed: real);

implementation

  var
    RandomTable: array[1..55] of real;
    current: integer;

  procedure CreateNext55;
    var
      i: integer;
      new_random: real;
  begin
    for i := 1 to 24 do begin
         new_random := RandomTable[i] - RandomTable[i + 31];
         if (new_random < 0.0) then
            new_random := new_random + 1.0;
         RandomTable[i] := new_random;
      end;
    for i := 25 to 55 do begin
         new_random := RandomTable[i] - RandomTable[i - 24];
         if (new_random < 0.0) then
            new_random := new_random + 1.0;
         RandomTable[i] := new_random;
      end;
  end;

  procedure CreateFirst55 (RandomSeed: real);
    var
      i, j: integer;
      new_random, prev_random: real;
  begin
    RandomTable[55] := RandomSeed;
    new_random := 1.0e-9;
    prev_random := RandomSeed;
    for i := 1 to 54 do begin
         j := 21 * i mod 55;
         RandomTable[j] := new_random;
         new_random := prev_random - new_random;
         if (new_random < 0.0) then
            new_random := new_random + 1.0;
         prev_random := RandomTable[j];
      end;
    CreateNext55; CreateNext55; CreateNext55;
    current := 0;
  end;

  function rand01: real;   { uniform random in [0,1] }
  begin
    current := current + 1;
    if (current > 55) then begin
         current := 1;
         CreateNext55;
      end;
```

```
    rand01 := RandomTable[current];
  end;

  function flip (probability: real): boolean;
  begin
    if (probability = 1.0) then
      flip := true
    else
      flip := (rand01 <= probability);
  end;

  function randInt (low, high: integer): integer;
    var
      i: integer;
  begin
    if (low >= high) then
      i := low
    else begin
         i := trunc(rand01 * (high - low + 1) + low);
         if (i > high) then
            i := high;
      end;
    randInt := i;
  end;

  procedure randSeeder (RandomSeed: real); { must be >0.0 and <1.0 }
  begin
    if (RandomSeed <= 0.0) or (RandomSeed >= 1.0) then begin
         writeln('Fatal error: RandomSeed'); halt;
      end;
    CreateFirst55(RandomSeed);
  end;

end.
```

EINFACHER, GENETISCHER ALGORITHMUS

Das folgende Program SgA vereinigt alle Elemente eines genetischen Algorithmus. Dabei wurde die Effizienz der klaren Übersichtlichkeit geopfert. Das Program kann auf einfache Optimierungsfunktionen angewandt werden. Die Zielfunktion muss in der Funktion ObjFunc implementiert werden. Im Listing ist die Funktion $f(x)=x^n$ mit n=10 implementiert.

```
{ --- simplest genetic Algorithm --- }
{ --- simplified from: GOLDBERG D.E., Genetic Alogithm in Search, Opt. &
Machine}
{ --- Lerning, Addison-Wesley Publishing Company, Inc., New York, 1989.}
{ --- pp. 343}

program SgA;

  uses
    Random;

  const
    MAXPOPULATION = 100;
    MAXCHROMOSOM = 30;

  type
    allele = boolean;
    chromosome = array[1..MAXCHROMOSOM] of allele;
    individual = record
          chrom: chromosome;
          x: real;
          fitness, partSum1: real;
       end;
    population = record
```

```
        generation: integer;
        max, min, avg, sumFitness: real;
        M: array[1..MAXPOPULATION] of individual;
     end;

var
   oldPop, newPop: population;
   pCross, pMutation: real;
   nCross, nMutation: integer;
   popSize, chromSize: integer;
   MaxGeneration: integer;
   best: individual;

function ObjFunc (x: real): real;
   const
      coef = 1073741823.0;
      n = 10;
   function power (x: real; n: integer): real;
      var
         i: integer;
         p: real;
   begin
      p := 1;
      for i := 1 to n do
         p := p * x;
      power := p;
   end;
begin
   ObjFunc := power(x / coef, n);
end;

function decode (var chrom: chromosome): real;
   var
      i: integer;
      accum, powerOf2: real;
begin
   accum := 0.0;
   powerOf2 := 1;
   for i := 1 to chromSize do begin
         if chrom[i] then
            accum := accum + powerOf2;
         powerOf2 := powerOf2 * 2;
      end;
   decode := accum;
end;

procedure InstallParameters;
begin
   pCross := 0.6;         { crossover rate }
   pMutation := 0.0333;   { muation rate }
   chromSize := 30;       { size of chromosome }
   popSize := 30;         { population size }
   MaxGeneration := 8;    { run 8 Generations }

   randSeeder(0.11);      { try 0.1 }
   nMutation := 0;        { counts the mutations }
   nCross := 0;           { counts the crossovers }
end;

procedure UpdatePopulationStatistics (var pop: Population);
   var
      i: integer;
begin
   with pop do begin
         max := M[1].fitness;
         min := max;
         sumFitness := max;
         M[1].partSum1 := max;
         for i := 2 to popSize do
            with pop.M[i] do begin
                     sumFitness := sumFitness + fitness;
                     partSum1 := sumFitness;
```

```
                        if fitness > max then begin
                                    max := fitness;
                                    if fitness > best.fitness then
                                          best := pop.M[i];
                              end;
                        if fitness < min then
                              min := fitness;
                  end;
            avg := sumFitness / popSize;
         end;
   end;

   procedure ReportPopulationStatistics (var pop: Population; how: integer);
  { how=0 only summary, how=1 all chromosomes }
      var
         i: integer;

      procedure writeChrom (var chrom: chromosome);
         var
            i: integer;
      begin
         for i := chromSize downto 1 do
            if chrom[i] then
               write('1')
            else
               write('0');
      end;

   begin
      writeln;
      with pop do begin
            writeln('gen:', generation, ' max:', max : 6 : 5, ' avg:',
                    avg : 6 : 5, ' best:', best.fitness : 6 : 5, ' min:',
                    min : 6 : 5, ' Fit:', sumFitness : 6 : 5, nCross : 4,
                    nMutation : 4);
            if how = 0 then
               exit(ReportPopulationStatistics);
            for i := 1 to popSize do
               with pop.M[i] do begin
                           writeChrom(chrom);
                           writeln(x : 10 : 5, ' ', fitness : 10 : 5);
                     end;
         end;
   end;

   procedure CreateStartPopulation (var pop: population);
      var
         i, j: integer;
   begin
      pop.generation := 1;
      for i := 1 to popSize do
         with pop.M[i] do begin
               for j := 1 to chromSize do
                  chrom[j] := flip(0.5);
               x := decode(chrom);
               fitness := ObjFunc(x);
            end;
      best := pop.M[1];
      UpdatePopulationStatistics(pop);
      ReportPopulationStatistics(pop, 0);
   end;

   function select (var pop: Population): integer;
  { select an individual for crossover, return the popualtion count }
      var
         rand, partSum: real;
         i: integer;
   begin
      with pop do begin
            partSum := 0.0;
            i := 0;
            rand := rand01 * sumFitness;
```

```
          repeat
             i := i + 1;
             partSum := partSum + M[i].fitness;
          until (partSum >= rand) or (i = popSize);
          select := i;
       end;
  end;

  procedure crossover (var parent1, parent2, child1, child2: chromosome);
{ produce two children from two parents }
     var
        i, crossPoint: integer;

     function mutation (alleleval: allele): allele;
        var
           mutate: boolean;
     begin
        mutate := flip(pMutation);
        if mutate then begin
              nMutation := nMutation + 1;
              mutation := not alleleval;
           end
        else
           mutation := alleleval;
     end;

  begin
     if flip(pCross) then begin
           crossPoint := randInt(1, chromSize - 1);
           nCross := nCross + 1;
        end
     else
        crossPoint := chromSize;
     for i := 1 to crossPoint do begin
           child1[i] := mutation(parent1[i]);
           child2[i] := mutation(parent2[i]);
        end;
     if crossPoint <> chromSize then
        for i := crossPoint + 1 to chromSize do begin
              child1[i] := mutation(parent2[i]);
              child2[i] := mutation(parent1[i]);
           end;
     UpdatePopulationStatistics(newPop);
      {pMutation:=pMutation/2;}
  end;

  procedure MakeNewGeneration (var oldPop, newPop: population);
     var
        father, mother: integer; {individuals}
        i: integer;
  begin
     newPop.generation := oldPop.generation + 1;
     i := 1;
     repeat
        father := select(oldPop);
        mother := select(oldPop);
        crossover(oldPop.M[father].chrom, oldPop.M[mother].chrom,
newPop.M[i].chrom,
                  newPop.M[i + 1].chrom);
        with newPop.M[i] do begin
              x := decode(chrom);
              fitness := ObjFunc(x);
           end;
        with newPop.M[i + 1] do begin
              x := decode(chrom);
              fitness := ObjFunc(x);
           end;
        i := i + 2;
     until (i > popSize);
  end;

begin   {---- main program ---}
```

```
  InstallParameters;
  CreateStartPopulation(oldPop);
  repeat
     MakeNewGeneration(oldPop, newPop);
     ReportPopulationStatistics(newPop, 0);
     oldPop := newPop;
  until newPop.generation > MaxGeneration;
end.
```

PMX_CROSSOVER OPERATOR

Diese Prozedur implementiert den PMX-Crossover-Operator (Goldberg 1989).

```
function posV (which: byte; var v: TPerm; n: integer): integer;
 var  i: integer;
begin
  i := 0;
  repeat
    i := i + 1;
  until (i > n) or (which = v[i]);
  posV := i;
end;

procedure swapV (from1, to1: integer; var v: TPerm);
 var  temp: byte;
begin
  temp := v[from1];
  v[from1] := v[to1];
  v[to1] := temp;
end;

procedure PMX_crossOver (low, high: integer; var old1, old2, new1, new2: TPerm;
                         n: integer);
  { crossover operator for permutation, Partially Mapped Crossover from Goldberg
}
  { in: Proceedings GA 1985 p. 154ff }
  { old1 and old2 are the parents, new1 and new2 are the newly created children
}
  { from both parents the chrunck in the range (low,high) is copied to their
children }

 var  i, hi_test: integer;
begin
  hi_test := high + 1;    {hi_test:=high mod n + 1 }
  if (hi_test > n) then  hi_test := 1;
  new1 := old1;
  new2 := old2;
  if ((low <> high) and (low <> hi_test)) then begin
    i := low;
    while (i <> hi_test) do begin
      swapV(i, posV(old1[i], new2, n), new2);
      swapV(i, posV(old2[i], new1, n), new1);
      i := i + 1;
      if (i > n) then  i := 1;
    end;
  end;
end;
```

REFERENCES

GENETISCHE ALGORITHMEN

ACKLEY D.H., [1987], A Connectionist Machine for Genetic Hillclimbing, Kluwer Academic Publ. Boston.

BLÜMECKE T., [1991], Wunder der Evolution, Optimierung mit Evolutionsstrategien und genetischen Algorithmen, c't Magazin, Heft 12.

DAVIS L. (ed.), [1991], Handbook of Genetic Algorithms,Van Nostrand Reinhold, New York.

DAVIS L. (ed.), [1987], Genetic Algorithms and Simulated Annealing, Pitman, London.

DAWKINS R., [1987], The Blind Watchmaker, W.W. Norton & Comp, New York.

GOLDBERG D.E., [1989], Genetic Algorithms in Search, Optimization & Machine Learning, Addison-Wesley.

GREFENSTETTE J.J., [1985], Proceedings of an International Conference on Genetic Algoritms and Their Applipations, July 24-26 at Carnegie-Mellon University Pittsburgh, PA.

ZUSCHNITT- UND ANDERE PROBLEME

BAKER B.S., COFFMAN E.G., RIVEST R.L., [1980], Orthogonal Packings in Two Dimensions, SIAM Journal on Computing, Vol.9, No.4, Nov 1980, S. 846-855.

BAKER B.S., BROWN D.J., KATSEFF H.P., [1981], A 5/4 Algoritm for Two-Dimensional Packing, in: Journal of Algorithms 2, p.348-368.

BEASLEY J.E., [1985a], Algorithms for the Unconstrainted Two-Dimensional Guillotine Cutting, in: Journal of the Operational Research Society, Vol 36(4), p.297-306.

BEASLEY J.E., [1985b], Bounds for Two-Dimensional Cutting, in: Journal of the Operational Research Society, Vol 36(1), p.71-74.

BEASLEY J.E., [1985c], An Exact Two-Dimensional Non-Guillotine Cutting Tree Search Procedure, in: Operations Research, Vol 33(1), p.49-64.

COFFMAN E.G., GAREY M.R., JOHNSON D.S., [1984], Approximation algorithms for the bin-packing - an updated survey, in: Ausiello G et al. (eds.), Approximation algorithms for Computer System Design, Wien, 1984.

COFFMAN E.G., GAREY M.R., JOHNSON D.S., TARJAN R.E., [1980a], Performance Bounds for level-oriented two-dimensional packing algorithms, SIAM Journal on Computing, Vol.9, No.4, Nov 1980, S. 808-826.

CHRISTOFIDES N., WHITLOCK C., [1977], An Algorithm for Two-Dimensional Cutting Problems, in: Operations Research, Vol. 25(1), p.30-44.

DYCKHOFF H., WÄSCHER G. (eds), [1990], Cutting and Packing, Special Issue of European Journal of Operational Research, Vol 44, No. 2, January 25, 1990, North-Holland, Amsterdam.

FARLEY A.A., [1988], The Gilmore-Gomory Approach, in: OR-Spektrum (1988) 10,2 :p.113-123).

FARLEY A.A., [1990], Two dimensional cutting stock situations, in: European Journal of Operational Research 44:2, Jan 1990, p.239-246.

GARDNER M., [1979], Mathematische Spielereien, Packungsprobleme mit Quadraten, in: Spektrum der Wissenschaften 1979, Heft 12. Dazu auch 1980 Heft 1 S.9, Heft 3 S. 5 und 1981 Heft 1 S.11.

GILMORE P.C., GOMORY R.E., [1961], A Linear Approach to the Cutting Stock Problem, in: Operations Research 9, p.849-859.

GILMORE P.C., GOMORY R.E., [1963], A Linear Approach to the Cutting Stock Problem (Part II), in: Operations Research 11, p.863-887.

GILMORE P.C., GOMORY R.E., [1965], Multistage cutting stock problems of two and more dimensions, in: Operations Research 13, p.94-120.

GILMORE P.C., GOMORY R.E., [1966], The theory and computation of knapsack functions, in: Operations Research 14, p.1045-1074.

HERZ J.C., [1972], A Recursive Computing Procedure for Two-Dimensional Stock Cutting, in: IBM Journal of Research Development 16, p.462-469.

HINXMAN A.I., [1980], The trim-loss and assortment problems: A survey, in: European Journal of Operational Reserach, 5(1), p. 8-18.

KRÖGER B., [1991], Elegant tiefstapeln, ein genetischer Algorithmus für ein Packproblem, MC Magazin, Mai 1991.

MARTELLO S., TOTH P., [1990], Knapsack Problems, Algorithms and Computer Implementation, John Wiley & Sons, Chichester.

NELISSEN J., [1991], Die Optimierung zweidimensionaler Zuschnittprobleme, Schriften zur Informatik und angewandten Mathematik, Hrsg.: Merkwitz J., Oberschelp W., Schinzel B., Rheinisch-Westfälische Technische Hochschule, Aachen,Bericht Nr.150, Oktober 1991.

TERNO J., LINDEMANN R., SCHEITHAUER G., [1987], Zuschnittprobleme und ihre praktische Lösung, Verlag Harri Deutsch, Frankfurt a.M.

WANG P.Y., [1983, Two Algorithms for Constrained Two-Dimensional Cutting Stock Problems, in: Operations Research 31, p.573-586.

5/1/93: see also

VAN DER WILT A., [1992], Transformation of two-dimensional unconstrained cutting problems into a set of one-dimensional knapsack problems, in: Belgian Journal of Operations Research, Statistics and Computer Science Vol. 32 (3,4), p.127-138.