Ein kombinatorisches Standortproblem

Christian Grothoff[1]

Version 1.00
25. Juli 2000

[1] Freiligrathstraße 70, D-42289 Wuppertal, ma0035@stud.uni-wuppertal.de

Vorwort

In dieser Arbeit wird ein Verfahren zur Lösung eines speziellen „Uncapacitated Facility Location" Problems (UFLP) vorgestellt. Bei diesem speziellen Problem müssen die Standorte für die verschiedenen Nachfrager verschiedene Funktionen erfüllen, um die jeweilige Nachfrage abdecken zu können. Jede Funktionalität an einem Standort erhöht die Kosten für alle vom Standort belieferten Nachfrager. Die große Menge möglicher Funktionen führt zu einer extrem großen Menge möglicher Standorte.

Bereits 1976-78 wurde von Erlenkotter ein Verfahren vorgestellt, mit dem das UFLP gelöst werden kann. Damals wurden Probleme üblicherweise mit dutzenden, höchstens jedoch ein paar hundert Facility Locations berechnet.

Hier soll jedoch ein Problem mit 2^n Facility Locations berechnet werden, wobei $n \geq 20$ gilt. Praktische Beispiele wurden mit bis zu $n = 45$ gerechnet. Diese Problemgröße kann, trotz der gestiegenen Hardwareleistung, nicht mit dem ursprünglichen Verfahren behandelt werden.

In den letzen Jahren konnte gezeigt werden, dass das UFLP $\mathcal{NP}$-hart ist.[1] Aktuelle Arbeiten beschäftigen sich mit kontrollierter Approximation an die Optimallösung. Mit einem „Greedy"-Algorithmus konnte eine Näherung an die Optimallösung mit einer Abweichung von höchstens einem Faktor von 2.408 garantiert werden. Falls ein polynominaler Algorithmus existieren sollte, der eine Approximation liefert, die garantiert höchstens um einen Faktor von 1.463 von der Optimallösung abweicht, so wurde gezeigt, dass dann $\mathcal{NP}$-Probleme mit einem Aufwand von $O(n^{\log\log n})$ lösbar sind.[2]

Im ersten Kapitel wird die Aufgabenstellung und Modellbildung erläutert, im zweiten werden heuristische Verfahren vorgestellt, mit denen eine Näherung an die Optimallösung berechnet werden soll. Eine Garantie für die Güte der Approximation kann jedoch nicht gegeben werden. Das dritte Kapitel beschreibt das Verfahren von Erlenkotter mit dem eine exakte Lösung des Problems möglich ist. Im vierten Kapitel wird beschrieben, wie die Heuristiken mit Erlenkotters Verfahren kombiniert werden können, um auch für große Probleme eine Optimallösung zu bestimmen. Im fünften Kapitel werden schließlich numerische Ergebnisse zu den Heuristiken vorgestellt.

Christian Grothoff

[1] vgl. [8] oder [11, S. 126ff]
[2] vgl. [10] und [9]

Motivation

Ein Hersteller von Pralinenmischungen hat 30 verschiedene Pralinen für seine Produkte zur Auswahl. Daher kann er 2^{30} verschiedene Mischungen herstellen, wenn jede Praline höchstens einmal in einer Mischung verwendet werden darf.

Bestellen seine Kunden aus diesen 2^{30} möglichen Kombinationen nur 400 verschiedene Mischungen, so hat der Hersteller ein Problem logistischer Art: Fertigung, Beschriftung, Sortierung, Lagerung und Abrechnung für 400 verschiedene Produkte.

Nun könnte er eine Pralinenschachtel mit allen 30 Pralinen füllen und nur diese verschicken. Alle Kunden wären zufrieden, denn die von ihnen gewünschten Pralinen sind ja enthalten. Nur hat der Fabrikant jetzt das Problem, dass Kunden, die möglicherweise 1.000 Schachteln mit nur 5 Pralinen bestellt haben, 25.000 Pralinen unnötigerweise erhielten. Der hohe Preis würde dazu führen, dass die Produkte nicht mehr wettbewerbsfähig wären.

Also sucht sich der Pralinenfabrikant einen Mathematiker, der ausgehend von den Pralinenpreisen, den bestellten 400 Mischungen und ihren Stückzahlen Produktionspläne erstellen soll, die alle Geschmacksrichtungen der Kunden abdecken und gleichzeitig möglichst wenige überflüssige Pralinen enthalten. Da keine halben Pralinen verpackt werden können, klassifiziert der Mathematiker dies als ein Problem der *ganzzahligen linearen Optimierung*.

Angesichts der theoretischen Lösung erklärt der Fabrikant dem Mathematiker, dass die Pralinen aus geschmacklichen Gründen nicht beliebig kombiniert werden können, bestimmte Kombinationen sind auszuschließen. Außerdem entspräche doch die „große-Herz“-Praline der „kleinen-Herz“-Praline im Geschmack, d.h. wenn der Kunde zwar nur die kleine bestellt hätte, wäre er aber auch mit der großen zufrieden.

Abschließend ist noch das Problem zu lösen, die richtige Verpackung für die Pralinen zu berechnen und die für die Verpackung aufzuwendenden Kosten in die Optimierung mit einzubeziehen.

Inhaltsverzeichnis

Kapitel 1

Aufgabenstellung und Modellbildung

In diesem Kapitel wird zunächst das Problem genauer vorgestellt und ein passendes mathematisches Modell aufgestellt.

1.1 Beschreibung der Problemstellung

Moduln sind Produktteile, die spezielle Aufgaben erfüllen. Bestellt ein Kunde ein Produkt mit gewissen Funktionen, so muss das gelieferte Produkt Moduln enthalten die diesen Funktionen entsprechen. In der Produktion müssen somit in das Produkt die entsprechenden Moduln eingebaut werden. Hohe Ansprüche der Kunden an die individuelle Gestaltung des Funktionsumfangs führen zu einer großen Anzahl verschiedener Produktionsvorgänge und somit zu einem großen **Logistikaufwand**.

Unter der Voraussetzung, dass jeder Kunde Produkte erhält die **mindestens** die von ihm bestellten Funktionen enthalten, kann die Vielfalt der Produktionsvorgänge reduziert werden, indem Produkte geliefert werden, die teilweise auch nicht bestellte Moduln enthalten. Dadurch können dann verschiedene Kundenwünsche durch das gleiche Produkt **abgedeckt** werden.

Beispiel: Ein Bauherr möchte ein Haus mit Balkon, ein anderer ein Haus mit Terrasse. Die Baugesellschaft läßt von einem Architekten ein Haus mit Balkon und Terrasse entwerfen und baut zwei gleiche Häuser.

Diese Verringerung der Produktvielfalt ist von einem Anstieg der Kosten für die erhöhte Stückzahl in der Produktion eingebauter Moduln begleitet. Die Produkte enthalten Moduln die von einzelnen Kunden nicht benutzt werden, den sogenannten **unused content**. Den sinkenden Logistikkosten (weniger verschiedene Produktionsvorgänge) stehen somit höhere Materialkosten gegenüber.

Es stellt sich daher die Frage nach einem kostenoptimalen Produktionsplan. Gesucht wird nach Kombinationen von Moduln, genannt **Varianten**, die alle Kundenwünsche abdecken und die im Preis (Herstellung und Logistik) möglichst günstig sind.

In der Praxis führen **Abhängigkeiten** zwischen den Moduln zu zusätzlichen Schwierigkeiten, so können z.B. nicht immer alle Moduln miteinander kombiniert werden.[1] Solche Abhängigkeiten werden in der folgenden Modellbildung und im theoretischen Teil nicht berücksichtigt. Das im Anhang A dokumentierte Programm kann jedoch mit teilweise sehr komplexen Abhängigkeiten umgehen, die im Abschnitt A.2.1 erläutert werden.

1.2 Modellbildung

In diesem Abschnitt werden die für die Problemspezifikation, -beschreibung und -analyse notwendigen Daten beschrieben. Anschließend wird ein konkretes mathematisches Modell vorgestellt. Abschließend folgt die Beschreibung diverser Algorithmen und Heuristiken, die der Problemlösung dienlich sind.

1.2.1 Eingangsdaten

Modulliste Liste von m Moduln mit Herstellungspreisen M_μ, $\mu = 1, \ldots, m$

Bestellliste Liste $B = \{b_i | i = 1, ..., n\}$ von n Modulkombinationen

$$b = (b_1, \ldots, b_m) \in K := \{k = (k_1, \ldots, k_m) \in \{0,1\}^m\}$$

mit Exemplaranzahl l_b.

Logistikkosten Kosten a_k, die durch die Produktion der Variante k (ohne Materialkosten) entstehen.

1.2.2 Ausgabedaten

Eine **zulässige Lösung** für das Problem besteht aus einer Liste V von p Varianten $v \in K$ die alle Bestellungen **abdecken**, d.h. für alle $b \in B$ existiert ein $v \in V$ mit

$$b_\nu \le v_\nu \quad \text{für } \nu = 1, \ldots, m.$$

Neben der Information, welche Varianten V zu produzieren sind, müssen auch noch die Stückzahlen und die Zuordnung zu den Bestellungen berechnet werden.

Somit ergeben sich folgende Ausgabedaten:

Produktionsplan Liste V mit p zu produzierenden Varianten $v \in K$ mit Stückzahl t_v.

z_k soll angeben, ob die Variante $k \in K$ produziert wird oder nicht. Es sei $z_k := 1$ für $k \in V$ und $z_k := 0$ für $k \notin V$.

Abdeckungsbeitrag Feld $d_{b,k} \in \mathbb{N}$, das angibt, wie häufig die Variante $k \in K$ die Bestellung $b \in B$ abdeckt.[2]

Entscheidungsvariable Feld $w_{b,k} \in \{0,1\}$, das angibt, ob die Variante $k \in K$ produziert wird um die Bestellung $b \in B$ abzudecken. Es gilt $d_{b,k} = l_b \cdot w_{b,k}$ und $w_{b,k} = 0$ für alle $k \notin V$, $b \in B$.

[1] Beispiel: Ein Computer kann nicht gleichzeitig Intel- und Alpha-Prozessoren beinhalten.

[2] Da für eine zulässige Lösung für jede Bestellung b eine Variante $v \in V$ die b mit minimalen Kosten abdeckt existiert, kann der Abdeckungsbeitrag auf die möglichen Werte 0 oder l_b eingeschränkt werden.

1.2.3 Einflußgrößen

Die folgenden Größen sind nützlich für die exakte Definition der Aufgabenstellung sowie für die durchgeführten Berechnungen.

Abdeckungen Menge $F \subset B \times K$, die jene (b,k) enthält, für die $b_\mu \leq k_\mu$ für alle $\mu = 1, \ldots, m$ gilt. F enthält also genau jene Paare (b,k) bei denen die Variante k die Bestellung b abdeckt.

Herstellungskosten $H_k \in \mathbb{N}$ bezeichne für jede Modulkombination $k \in K$ die Herstellungskosten:

$$H_k := \sum_{\mu=1}^{m} k_\mu \cdot M_\mu$$

Materialmehrkosten Die Größe $U_{b,k} \in \mathbb{N}$ gibt für jede Bestellung $b \in B$ und jeder Variante $k \in K$ die Größe des Unused Content an, der entsteht, wenn die Bestellung b durch die Variante k abgedeckt werden soll:

$$U_{b,k} := H_k - H_b$$

Gilt dabei nicht $b \leq k$ so ist $U_{b,k} = \infty$ anzunehmen.

gesamt-Unused Content Summe der tatsächlich anfallenden Materialmehrkosten für den betrachteten Produktionsplan:

$$UC(d) := \sum_{(b,k)\in F} d_{b,k} U_{b,k}$$

1.2.4 Übersicht über die Bezeichnungen

$m \in \mathbb{N}$: Anzahl der Moduln

$M_\mu \in \mathbb{N}$: Preis des μ-ten Moduls, $\mu = 1, \ldots, m$

$K = \{k \in \{0,1\}^m\}$, Menge aller Modulkombinationen

$n \in \mathbb{N}$: Anzahl der verschiedenen Bestellungen

$B = \{b_i | i = 1, ..., n\}$ Menge aller bestellten Modulkombinationen

$l_b \in \mathbb{N}$: Stückzahl in der $b \in B$ bestellt wurde

$v_j \in K$: zu produzierende Variante, $j = 1, \ldots, p$

$V = \{v_j | j = 1, ..., p\}$ Menge der zu produzierenden Varianten

z_k 1 für $k \in V$ und 0 für $k \notin V$

$p \in \mathbb{N}$: Anzahl der zu produzierenden Varianten, $p = \sum_{k\in K} z_k$

t_v: Stückzahl, in der $v \in V$ zu produzieren ist.

$F = \{(b,k) \in B \times K | b_\mu \leq k_\mu$ für alle $\mu = 1, \ldots, m\}$

$a_k \in \mathbb{N}$: Kosten, die durch die Produktion der Variante $k \in K$ entstehen.

$d_{b,k} \in \{0, l_b\}$: Stückzahl der Variante $k \in K$ die für die Bestellung $b \in B$ produziert wird.

$w_{b,k} = \frac{d_{b,k}}{l_b}$

$H_k = \sum_{\mu=1}^{m} k_\mu \cdot M_\mu,\ k \in K$

$U_{b,v} = H_v - H_b,\ v \in V,\ b \in B$

$UC(d) = \sum_{(b,k)\in F} d_{b,k} U_{b,k}$

1.2.5 Motivation des mathematischen Modells

Ausgehend von den obigen Definitionen kann das Ziel wie folgt formuliert werden:

- Um die Gesamtkosten zu minimieren, muss die Summe aus Herstellungskosten und den Logistikkosten minimiert werden.
- Die Produktionskosten setzen sich aus dem genutzen und dem ungenutzten Anteil zusammen. Da der genutzte Anteil sich aus den Bestellungen ergibt und konstant ist, genügt es den Unused Content zu betrachten.
- Ziel ist es also die Summe aus Logistikkosten und Unused Content-Kosten zu minimieren. Dabei ist zu berücksichtigen, dass alle Bestellungen durch die produzierten Varianten abgedeckt werden.

1.2.6 Formulierung des Modells

Das folgende mathematische Modell übersetzt die obigen Forderungen. Da die Varianten v zu wählen sind, ist statt über die $v \in V$ über alle denkbaren Varianten $k \in K$ zu summieren. Welche Varianten gewählt wurden, kennzeichnet $z_k \in \{0,1\}$.

$$\text{minimiere} \quad \sum_{(b,k)\in F} w_{b,k} \cdot l_b \cdot U_{b,k} + \sum_{k\in K} a_k z_k \tag{1.1}$$

$$\text{so dass} \quad \sum_{(b,k)\in F} w_{b,k} = 1 \text{ alle } b \in B \tag{1.2}$$

$$w_{b,k} \leq z_k \text{ alle } (b,k) \in F \tag{1.3}$$

$$w_{b,k} \in \{0,1\} \text{ alle } (b,k) \in F \tag{1.4}$$

$$z_k \in \{0,1\} \text{ alle } k \in K \tag{1.5}$$

(1.1) Es werden alle Kosten aufsummiert. Logistikkosten $a_k z_k$ plus die Unused-Content-Kosten für alle die Bestellungen $b \in B$ multipliziert mit der Anzahl der Bestellungen l_b für das $k \in K$, das als b abdeckende Variante gewählt wurde ($w_{b,k} = 1$).

(1.2) Durch diese Forderung wird gewährleistet, dass jede Bestellung genau von einer produzierten Variante abgedeckt wird.

(1.3) Eine Variante k kann nur dann zur Abdeckung beitragen ($w_{b,k} = 1$), wenn sie auch produziert wird ($z_k = 1$).

Die Aufgabe ist somit eine lineare ganzzahlige Optimierungsaufgabe mit einer großen Anzahl von Variablen und Restriktionen.

Das Modell hat die Form eines *uncapacitated facility location* Problems, für das einige Lösungsverfahren bekannt sind.

Eine Vereinfachung kann erreicht werden, indem

$$c_{b,k} := l_b \cdot U_{b,k} \tag{1.6}$$

gesetzt wird. Die Problemformulierung lautet dann:

$$\begin{cases} \text{minimiere} & \sum_{(b,k)\in F} w_{b,k} c_{b,k} + \sum_{k\in K} a_k z_k \\ \text{so dass} & \sum_{(b,k)\in F} w_{i,k} = 1 \text{ alle } b \in B \\ & w_{b,k} \leq z_k \text{ alle } (b,k) \in F \\ & w_{b,k} \in \{0,1\} \text{ alle } (b,k) \in F \\ & z_k \in \{0,1\} \text{ alle } k \in K \end{cases} \tag{1.7}$$

Dieses Modell wird insbesondere die Grundlage für die theoretische Betrachtung im Kapitel 3 sein.

1.2.7 Ein alternatives Modell

Da die a_k nicht immer zu ermitteln sind (keine Datengrundlage, weil sehr viele Faktoren zu berücksichtigen sind), wird auf die Betrachtung dieses Teils der Zielfunktion zunächst verzichtet. Unter der Annahme, dass $a_k = a$ konstant für alle $k \in K$, kann ein Problem formuliert werden, aus dessen Lösung dann bei bekanntem a die Optimallösung des ursprünglichen Problems unmittelbar folgt.

Gesucht werde ein kostenminimaler Produktionsplan unter der Vorgabe eine bestimmte Variantenanzahl p zu produzieren. Kann eine Optimallösung für jede Zahl p angegeben werden, so ist es (bei bekanntem a) einfach, das tatsächliche Optimum zu ermitteln. Dazu ist nur $a \cdot p$ auf die Kosten der Optimallösungen zu addieren und das Minimum unter den so gebildeten Werten zu suchen.

Damit nicht alle möglichen Variantenanzahlen p untersucht werden müssen, ist es aufgrund von Erfahrungen mit dem konkreten Problem sinnvoll, den zu untersuchenden Bereich von p auf ein Intervall $[p_u, p_o]$ einzuschränken.

Bei vorgewähltem p besteht die **alternative Aufgabestellung** dann darin, eine **kostenminimale disjunkte Zerlegung** $Z = \{B_1, B_2, \ldots, B_p\}$ der Bestellliste $B = \bigcup B_\zeta$ zu finden. Die zu produzierenden Varianten ergeben sich als **einhüllende Modulkombinationen** $E_\zeta \in K$ der jeweiligen Teillisten B_ζ, die all jene Moduln enthalten, die in mindestens einer der Bestellungen $b \in B_\zeta$ vorkommen:

$$E_{\zeta,\mu} = \max_{b\in B_\zeta}\{b_\mu\}$$

Dass sich so eine zulässige Lösung ergibt, folgt unmittelbar aus der Definition der E_ζ. Die Kosten der Zerlegung entsprechen dem Unused Content der entsteht, wenn die Bestellung $b \in B_\zeta$ durch E_ζ abgedeckt wird. Dass sich die Optimallösung immer so darstellen läßt, besagt das folgende

Lemma 1.1 *Sei V die Liste der zu produzierenden Varianten im Optimalfall, d.h. $V = \{k \in K | z_k = 1\}$. Sei $v \in V$ beliebig und*

$$B_v := \{b \in B | w_{b,v} = 1\}.$$

Ferner sei

$$E_v := \max\{b \in B_v\}.$$

Dann gilt $v = E_v$.

Beweis: Da nach Definition $w_{b,v} = 1$ für alle $b \in B$, muss v alle Varianten $b \in B_v$ abdecken (eine Optimallösung muss zulässig sein). Damit folgt

$$\begin{aligned} & v_\mu \geq b_\mu \text{ für alle } b \in B_v \\ \Rightarrow & v_\mu \geq \max_{b \in B_v} b_\mu \\ & = E_\mu \end{aligned}$$

Angenommen $v > E_v$. Dann könnte v durch E_v ersetzt werden. Die Lösung bliebe zulässig, da E_v nach Konstruktion alle Bestellungen abdeckt, die v abdeckt. Gleichzeitig würden die Kosten jedoch echt kleiner (es war $M_\mu > 0$ vorausgesetzt). Widerspruch. Es folgt die Behauptung.

□

Ausgehend von dieser Feststellung erhält man zwei grundlegende Arbeitstechniken. Durch wiederholte Anwendung dieser Techniken kann im Prinzip jede Optimallösung konstruiert werden.

Gegeben seien die Bestellliste B mit n Bestellungen $b \in B$ der Stückzahl l_b, eine Variantenliste V mit Stückzahlen t_v und Abdeckungsbeiträgen $d_{b,k}$, $b \in B$, $v \in V$. Als zulässige Startlösungen sind entweder die Bestellliste als Variantenliste ($V = B$, $d_{b,b} = l_b$, $d_{b,k} = 0$ für $k \notin B$) bzw. die Einhüllende bezogen auf die gesamte Bestellliste als einzige Variante ($V = \{E_B\}$, $d_{b,E_B} = l_b$, $d_{b,k} = 0$ sonst) denkbar.

„Kleben“

Gesucht wird eine neue Variantenliste V_-, die eine Variante weniger enthält als die alte Liste V. Dazu sind zwei bestehende Varianten v_1 und v_2 zu einer neuen Variante v_3 zusammenzufassen:

$$v_3 := \max\{v_1, v_2\}$$

Die Varianten v_1 und v_2 werden aus der alten Variantenliste entfernt. Es gilt $d_{b,v_3} = d_{b,v_1} + d_{b,v_2}$ für alle $b \in B$.

„Spalten“

Gesucht wird eine neue Variantenliste V_+, die genau eine Variante mehr enthält als die alte Liste V. Dazu ist eine Variante v, der mindestens zwei Bestellungen b_i, $i \in I$ zugeordnet sind,[3] in zwei neue Varianten v_1 und v_2 aufzuspalten, die

[3] I sei die Menge der Indizes der Bestellungen, die v zugeordnet sind

die Bestellungen b_{j_1}, $j_1 \in J_1$ bzw. b_{j_2}, $j_2 \in J_2$ abdecken:

$$\begin{aligned} v &= \max_{i \in I}\{b_i\} \\ I &= J_1 \cup J_2 \\ J_1 \cap J_2 &= \emptyset \\ v_1 &:= \max_{j_1 \in J_1}\{b_{j_1}\} \\ v_2 &:= \max_{j_2 \in J_2}\{b_{j_2}\} \end{aligned}$$

1.3 Dynamische Optimierung

Das alternative Modell kann als Aufgabe der endlichen dynamischen Optimierung[4] formuliert werden. Die folgende Modellbildung soll sich dabei an der Verfahrensweise beim „Kleben“ orientieren. Für „Spalten“ ist eine analoge Modellbildung möglich.

1.3.1 Das Standardproblem der diskreten dynamischen Optimierung

Das allgemeine dynamische Modell der diskreten dynamischen Optimierung lautet wie folgt:

$$\begin{array}{llcll} \text{minimiere} & \sum_{j=2}^{p} UC_j(V_{j-1}, y_j) & & & \\ & V_j & = & f_j(V_{j-1}, y_j) & j = 2, \ldots, p \\ \text{so dass} & V_j & \in & X_j & j = 1, \ldots, p \\ & y_j & \in & S_j(V_{j-1}) & j = 2, \ldots, p \end{array}$$

Dabei sind $UC_j : X_j \times S_j(V_{j-1}) \to \mathbb{R}$ Kostenfunktionen. f_j, X_j und S_j sind vorgegeben. Jeder Übergang des Modells von Stufe zu Stufe ist mit Kosten $UC_j(V_{j-1}, y_j)$ verbunden.

V_j ist die Zustandsvariable, die den Zustand des Modells in der j-ten Stufe, $j = 1, \ldots, p$, beschreibt. $[1, p]$ wird als **Planungszeitaum** bezeichnet. Die Folge $(V_j)_{j=1,\ldots,p}$ heißt Zustandsfolge. Der Anfangszustand V_1 ist vorgegeben.

Für festes j, $j = 2, \ldots, p$, heißt $y_j \in S_j(V_{j-1})$ Entscheidungsvariable. Der Bereich $S_j(V_{j-1})$ in dem y_j variieren darf, heißt **Steuerbereich**. Die Folge $(y_j)_{j=2,\ldots,p}$ wird als **Steuervektor**, **Entscheidungsfolge** oder **Politik** bezeichnet. Diese beschreibt den Übergang des Modells von Stufe $j-1$ nach Stufe j.

Eine Übergangsfunktion $f_j(V_{j-1}, y_j)$ beschreibt, wie der Zustand des Modells in der j-ten Stufe vom vorhergehenden Zustand V_{j-1} und der j-ten Entscheidung abhängt:

$$V_j = f_j(V_{j-1}, y_j) \qquad j = 2, \ldots, p$$

[4] vgl. z.B. [6, S. 27ff]

Das Bellmansche Optimalitätsprinzip

Auf dem folgenden Satz (ohne Beweis) baut das übliche Lösungsverfahren für das Standardproblem auf:

Satz 1.2 (Optimalitätsprinzip von Bellman) *Es gibt eine optimale Politik* $(y_j, \ldots, y_p)$ *eines auf der Stufe* j, $2 \le j \le p$ *eines* p*-stufigen Prozesses beginnenden* $(p-j+1)$*-stufigen Teilprozesses, die nur von dem Wert des Zustandsvektors* V_{j-1} *zu Beginn der Stufe* j *und nicht explizit von den vorhergehenden Entscheidungen* $y_2, \ldots, y_{j-1}$ *des Gesamtprozesses abhängig ist.*

Das Optimalitätsprinzip kann noch plakativer formuliert werden:[5]

> „Jeder Teilweg eines optimalen Weges oder Teilweges ist optimal.“

Die Bellmansche Funktionalgleichungsmethode

Bezeichnet $\pi_j^*(V_{j-1})$ den **optimalen Wert** der Zielfunktion, wenn nur der Teilprozess betrachtet wird, der bei V_{j-1} beginnt und für den mindestens ein Prozessablauf existiert, der bei V_{j-1} beginnt und der diesen Wert hat. Dann gilt:

$$\pi_j^*(V_{j-1}) = \min \left\{ UC_j(V_{j-1}, y_j) + \pi_{j+1}^*(V_j) \;\middle|\; \begin{array}{c} y_j \in S_j(V_{j-1}), \\ f_j(V_{j-1}, y_j) \in X_j \end{array} \right\} \tag{1.8}$$

$\pi_j^* : X_{j-1} \to \mathbb{R}$ heisst die j-te Wertfunktion des Prozesses. Durch rekursive Berechnung können ausgehend von einem vorgegebenen Anfangs- bzw. Endzustand V_1 bzw. V_p die Wertfunktionen berechnet und durch Rückwärtsrechnung die optimale Politik bestimmt werden.

Um später leichter auf das gegebene Problem übergehen zu können, soll nun der Algorithmus für Stufen $j = p, \ldots, n$ $(p < n)$, ausgehend von einem vorgegebenem Element B für den Zustand V_n beschrieben werden.

> **Da die folgende Formulierung bereits auf das nachfolgend betrachtete Problem angeglichen wurde, sind die Bezeichner nicht mehr mit der obigen Formulierung der Aufgabestellung (die mehr die traditionelle Form hatte) kompatibel. Insbesondere wurde die Richtung der Optimierung geändert. Auch die Bedeutung von n und p ist anders.**

1. Es sei $X_n = \{B\}$. Setze $j := n - 1$ und $\pi_n^*(B) = 0$.
2. Solange $j > p$ führe aus:
   (a) Für alle $V_j \in X_j$ berechne gemäß
   $$\pi_{j-1}^*(V_j) = \min \left\{ UC_j(V_j, y_{j-1}) + \pi_j^*(V_{j+1}) \;\middle|\; \begin{array}{c} y_{j-1} \in S_{j-1}(V_j), \\ f_{j-1}(V_j, y_{j-1}) \in X_{j-1} \end{array} \right\}$$
   den Wert von $\pi_{j-1}^*(V_j)$ unter Benutzung von π_j^*.
   (b) Merke für jedes $V_j \in X_j$ die Stelle $y_{j-1} =: \eta_{j-1}^*(V_j)$ an der das Optimum jeweils angenommen wurde.

[5] [7, S. 24]

   (c) Senke j um 1 ab.

3. Bestimme das Minimum V^* von π_p^*.

4. Bestimme ausgehend von $V^* = V_p^*$ die optimalen Entscheidungen aus den Funktionen η_j nach

$$\begin{aligned} y_{j-1}^* &= \eta_{j-1}(V_j^*) \\ V_{j-1}^* &= f_{j-1}(V_j^*, y_{j-1}) \end{aligned}$$

   für $j = p, \ldots, n$.

1.3.2 Übersetzung des Modells

Basierend auf Lemma 1.1 kann das Problem wie folgt als **endliche diskrete dynamische Optimerungsaufgabe** beschrieben werden.

Planungszeitraum

Als Planungszeitraum ist dabei die Anzahl der Varianten im Produktionsplan zu wählen. Der Planungszeitraum läuft dann von n Varianten ($V = B$ als Startlösung) bis zu p Varianten (p vorgegebene Variantenzahl für die ein optimaler Produktionsplan zu erstellen ist). Die Zwischenstufen sind jeweils Produktionspläne mit j Varianten, $p \leq j \leq n$.

Entscheidungsprozess

Auf der j-ten Stufe ist jeweils die Entscheidung zu treffen, welche zwei Varianten zusammengefasst werden sollen. Dadurch entsteht ein Produktionsplan der zur $j-1$-ten Stufe (oder Periode) gehört. Der **Zustand** des betrachteten Systems wird somit durch den Produktionsplan (der wiederum angibt, welche Bestellungen zusammengefasst wurden) beschrieben.

Zulässigkeit

Dass sich durch diese Entscheidungsfolge und diese Festlegung des Planungszeitraums jede Optimallösung mit p Varianten darstellen läßt, besagt das Lemma 1.1 zusammen mit der Tatsache, das „Kleben“ nach Definition nichts anderes ist, als das Bilden der einhüllenden Modulkombination der beteiligten Bestellungen.

Kostenfunktion

Die Kosten der j-ten Stufe entsprechen dem zusätzlichen Unused Content der durch das „Kleben“ der beiden ausgewählten Varianten entsteht.

1.3.3 Das Problem

Leider kann das beschriebene Verfahren nicht direkt auf das Problem angewendet werden. Die Menge der im Algorithmus im Schritt 2a zu beachtenden Zustände ist zu groß. Ausgehend von $n = m$ Bestellungen $b = e_\mu$,[6] einem relativ

[6] D.h. jede Bestellung enthält nur ein (hier: das μ-te) Modul, alle Bestellungen sind verschieden.

leicht zu diskutierendem Extrem, enthält X_j mindestens $\binom{n}{j}$ verschiedene V_j. Zu viele, um für jede Stufe alle Möglichkeiten zu betrachten.

1.3.4 Begrenzte Enumeration

Die Enumeration (also die Betrachtung aller Möglichkeiten) muss somit beschränkt werden. Da das Minimum in der Gleichung

$$\pi^*_{j-1}(V_j) = \min \left\{ UC_j(V_j, y_{j-1}) + \pi^*_j(V_{j+1}) \,\middle|\, \begin{array}{c} y_{j-1} \in S_{j-1}(V_j), \\ f_{j-1}(V_j, y_{j-1}) \in X_{j-1} \end{array} \right\}$$

in den meisten Fällen wohl an Stellen V_{j+1} angenommen wird, für die $\pi^*_j(V_{j+1})$ bereits relativ klein war, entsteht die Idee, nur die V_{j+1} mit den kleinsten $\pi^*_j(V_{j+1})$ weiter zu betrachten. Wird diese Anzahl auf eine feste kleine natürliche Zahl κ beschränkt, so entstehen **κ-gepufferte** Verfahren in denen die **Enumeration** auf die κ „besten“ Varianten **beschränkt** wird.

Es entstehen die im Abschnitt 2.2 vorgestellten **Heuristiken**, die allerdings die Optimalität nicht mehr garantieren können, da nicht mehr über alle Möglichkeiten minimiert (bzw. maximiert) wurde.

Im Kapitel 4 wird daher ein Ansatz vorgestellt wie die Optimalität noch nachträglich bewiesen bzw. hergestellt werden kann. Dazu wird das im Kapitel 3 beschriebene Verfahren zur Lösung des Standortproblems benutzt.

Kapitel 2

Heuristische Lösungsverfahren

In diesem Kapitel werden die Verfahren und Prozeduren vorgestellt, die hinter dem in Kapitel A dokumentierten Programm stehen und die neben dem Verfahren von Erlenkotter die Grundlage für die Lösung des Problems bilden.

2.1 Handwerkszeug

Als Handwerkszeug dienen konkrete Prozeduren die zentrale Teilaufgaben im Rahmen der Anwendung der Heuristiken erledigen.

2.1.1 Maximale Modulkombinationen

Ein Hilfsverfahren für das „Spalten“ besteht darin, in einer vorgegebenen Teilmenge B_ζ der Bestellliste B die Menge der **maximalen Modulkombinationen** $X \subset B_\zeta$ zu finden. Die Elemente dieser Menge sind dadurch gekennzeichnet, dass ihre Elemente paarweise keine Ungleichungsrelationen erfüllen (es gilt also weder $x \leq y$ noch $y \leq x$ für alle $x, y \in X$) und die alle $b \in B_\zeta$ **dominieren**, d.h. es gibt für alle $b \in B_\zeta$ ein $x \in X$ mit $b \leq x$.

Die maximalen Modulkombinationen können gefunden werden, indem alle $b \in B_\zeta$ einmal durchlaufen werden und dabei X erzeugt wird:

1. Setze $X := \emptyset$
2. Für alle $b \in B_\zeta$ führe aus:
   (a) Für alle $x \in X$ führe aus:
      i. $b \leq x$, dann: nächstes b
      ii. $b \geq x$, dann:
         A. $X := (X - \{x\}) \cup \{b\}$
         B. nächstes b
   (b) wenn alle $x \in X$ betrachtet: $X := X \cup \{b\}$.

Lemma 2.1 *Das Verfahren liefert die Menge der gesuchten maximalen Modulkombinationen X von B_ζ.*

Beweis: Zu zeigen ist:

1. am Ende des Verfahrens existiert für alle $b \in B_\zeta$ ein $x \in X$ mit $b \leq x$.
2. für alle $x, y \in X$ gilt weder $x \leq y$ noch $y \leq x$.

Im Laufe des Verfahrens werden nur dann neue Varianten b in X aufgenommen, wenn für alle bisherigen $x \in X$ weder $b \leq x$ noch $b \geq x$ gilt (bzw. die Variante $x \leq b$ wird aus X entfernt). Damit ist die zweite Bedingung erfüllt.

Angenommen die erste Bedingung sei verletzt. Dann existiert kein $x \in X$ mit $b \leq x$. Aber im Laufe des Verfahrens muss zu einem bestimmten Zeitpunkt ein $x \in X$ existiert haben mit $b \leq x$, denn jedes $b \in B_\zeta$ wird im Laufe des Verfahrens betrachtet. Somit gibt es drei Möglichkeiten:

1. Es existiert bereits ein $x \in X$ mit $b \leq x$. Damit gibt es zumindest zu diesem Zeitpunkt ein $x \in X$ mit $x \geq b$.
2. Es existiert ein $x \in X$ mit $b \geq x$. Dann wird x durch b ersetzt (Schritt 2(a)iiA). Zu diesem Zeitpunkt existiert dann $b \in X$ und für b gilt natürlich insbesondere $b \leq b$.
3. Es existiert kein $x \in X$ mit $b \leq x$ oder $b \geq x$. Auch in diesem Fall wird (im Schritt 2b) b in X aufgenommen.

Somit existiert an einer Stelle im Verfahren ein $x \in X$ mit $b \leq x$. Aber eine Variante x wird im Verfahren nur genau dann aus X entfernt, wenn sie gleichzeitig durch eine größere Variante $y \geq x$ ersetzt wird. Für dieses y gilt dann aber auch $y \geq b$. Es folgt die erste Bedingung.

□

2.1.2 gepuffertes Kleben in begrenzter Enumeration

Für vorgegebene Listen E^p von κ' Produktionplänen V mit je p Produktionsvarianten v_j werden alle Möglichkeiten geprüft, diese durch Zusammenfassen („Kleben") von je zwei Varianten neuen Produktionsplänen V' mit je $p-1$ Produktionsvarianten zu kombinieren. Die höchstens $\kappa \geq \kappa'$ (vom Preis her) besten neuen Listen werden weiterverwendet. Im Pseudocode:

1. Setze $E^{p-1} = \emptyset$.
2. Für jede der gegebenen κ' Produktionpläne V aus E^p führe aus:

   Für $j = 1$ bis $p-1$ führe aus:

   Für $l = j+1$ bis p führe aus:[1]

   (a) Bilde eine neue Variante v^{jl} gemäß

   $$v^{jl}_\mu := \max\{v_{j,\mu}, v_{l,\mu}\} \tag{2.1}$$

   für alle Moduln $\mu = 1, \ldots, m$.

[1] insgesamt: für alle Paare von Varianten in V

(b) Ersetze im Produktionsplan V die beiden Varianten v_j und v_l durch die neue Variante v^{jl}.[2] Setze $d_{b,v^{jl}} := d_{b,v_j} + d_{b,v_l}$ und $w_{b,v_j} = w_{b,v_l} := 0$. Der neue Plan heiße V'.

(c) Berechne die Kosten des neuen Produktionsplans V' (ohne Logistikkosten):

$$\sum_{(b,k)\in F} w_{b,k} \cdot l_b \cdot U_{b,k}$$

(d) Füge den neuen Produktionsplan in E^{p-1} ein.

(e) Ist die Anzahl der Produktionspläne in E^{p-1} größer als κ, so entferne den teuersten Produktionsplan aus E^{p-1}.

3. Ergebnis ist E^{p-1}.

Danach enthält E^{p-1} höchstens[3] κ Produktionspläne mit $p-1$ Varianten. Das Verfahren kann bis E^1 fortgesezt werden.

2.1.3 gepuffertes Spalten in begrenzter Enumeration

Vorgegebene κ' Produktionspläne mit p Varianten v_j sollen höchstens in $\kappa \geq \kappa'$ neue Produktionspläne mit je $p+1$ Varianten v'_j **unterteilt** werden. Die höchstens $\kappa \geq \kappa'$ von den Kosten her besten Listen werden, wie schon beim Kleben, weiterverwendet.

Für das „Spalten“ wird zunächst eine Variante v ausgewählt, die mindestens zwei (verschiedene) Bestellungen abdeckt. Da im folgenden nicht alle Unterteilungsmöglichkeiten der jeweiligen Variante geprüft werden, bietet es sich an, zumindest alle Varianten auszuprobieren. Zum **Unterteilen** einer Variante bieten sich dann zwei Prozeduren an:

maximale Modulkombinationen

Die erste Möglichkeit den Teilungsprozess sinnvoll durchzuführen ist, maximale Modulkombinationen in der Menge der Bestellungen zu finden, die von der zu teilenden Variante $v \in V$ abgedeckt werden.

Gesucht werden zunächst zwei beliebige verschiedene maximale Modulkombinationen. Führt das in Abschnitt 2.1.1 beschreibene Verfahren jedoch nur zu einer einzigen maximalen Modulkombination, so ist diese aus der Menge herauszunehmen und dann erneut zu suchen. Dann findet man mindestens eine weitere Modulkombination (wenn die Anzahl der Kombinationen überhaupt größer als 1 war).

Von diesen zwei Kombinationen nimmt man eine (im Sonderfall die zuletzt gefundene) und fügt diese der Variantenliste hinzu. Die zweite neue Variante entsteht aus allen durch die erste nicht abgedeckten Bestellungen, die vorher jedoch von v abgedeckt wurden.

---

[2] Dies ist zulässig, denn v^{jl} deckt nach Konstruktion alle Bestellungen ab, die v_j oder v_l abdeckten.

[3] Höchstens, da z.B. für E^1 nur genau ein Produktionsplan möglich ist.

0-1-Zerlegung

Diese Methode untersucht zunächst alle Moduln M der vorgegebenen Variante v auf das Einsparungspotential an unused content, wenn eine Spaltung von v so vorgenommen wird, dass alle Bestellkombinationen, die das μ-te Modul benutzen, μ beliebig vorgegeben, zu v' zusammengefasst werden und der Rest zu v'':

$$v' := \max\{b | b \in B, b_\mu = 1, w_{b,v} = 1\}$$
$$v' := \max\{b | b \in B, b_\mu = 0, w_{b,v} = 1\}$$

Diese Unterteilung macht nur dann Sinn, wenn die Variante v sowohl Bestellungen abdeckt, die das μ-te Modul enthalten, als auch die, die es nicht enthalten (weder v' noch v'' dürfen „leer“ sein).

Unter allen möglichen Unterteilungen jeder in Frage kommenden Varianten v werden die $\kappa \geq \kappa'$ besten weiterverwendet.

Beim „Spalten“ entstehen neue Varianten mit geringeren Kosten. Daher kann es passieren, dass Bestellungen von einer dieser Varianten günstiger abgedeckt werden, als von der, der sie bislang zugeordnet sind. Daher ist nach dem Unterteilen der Variante eine **Anpassung** der Zuordnungen für die Varianten v' und v'' vorzunehmen.

Anpassung

Für die Anpassung der Varianten ist für alle Bestellungen $b \in B$ zu prüfen, ob diese durch eine der neuen Varianten v abgedeckt werden. Wenn ja ist zu prüfen, ob diese Abdeckung kostengünstiger ist, als die bestehende günstigste Abdeckung durch eine Variante $x \in V$ $(w_{b,x} = 1)$.

Ist dies ebenfalls der Fall, so muss die Bestellung „umziehen“. Beim „Umziehen“ ist die Bestellung b zu der Variante v hinzuzufügen ($d_{b,v}$ auf l_b setzen und t_v um l_b erhöhen) sowie aus der Liste der x zugeordneten Bestellungen zu entfernen ($d_{b,x}$ auf 0 abzusenken, t_x um l_b absenken).

Durch diesen „Umzug“ kann sich auch die Variante x, von der b bislang abgedeckt wurde, ändern. Denn da x jetzt b nicht mehr abdecken muss, können möglicherweise Moduln aus x entfernt werden. Daher ist x neu zu berechnen:

$$x := \max\{b' | b' \in B, b' \neq b, w_{b,x} = 1\}$$

Wurde x dadurch verkleinert, muss die Anpassungsprozedur erneut für x durchlaufen werden.

Beispiel: Betrachtet werde ein Beispiel mit sechs Moduln und vier Bestellungen. Alle Moduln sollen den Preis $M_\mu = 1$ haben. Bestellt seien $a = 011110$, $b = 001011$, $c = 111001$ und $d = 000001$ $(a, b, c, d \in B)$.

Als zu produzierende Varianten sollen zunächst $x = 011111$ und $y = 111001$ betrachtet werden $(x, y \in V)$. x soll a und b, y die Bestellungen c und d abdecken:

$$a, b \to x \qquad c, d \to y$$

Durch „Spalten“ von x entstehe dann die Variante $v = b = 001011$, die neu in die Liste der zu produzierenden Varianten, V, aufgenommen wird.

Durch die Hinzunahme von v zu V wird die Anpassung für v und x notwendig. Die Bestellung b wurde beim „Spalten“ der Variante v zugeordnet.
Anpassung von x:
Durch die Zuordnung von v zu b wird eine Eins in $x = 011111$ überflüssig. x kann zu $x' = a = 011110$ reduziert werden. Die Abdeckung der verbliebenen Bestellungen (hier nur a) bleibt dabei bestehen.
Anpassung von v:
Die Bestellung d wird von der neuen Variante v besser abgedeckt als von y (um 1 geringere Kosten). Also muss auch d „umziehen“. Hier endet die Anpassung, da trotz des „Umzuges“ von d die Variante y nicht reduziert werden kann.
Die endgültige Zuordnung ist somit $a \to x'$, $b, d \to v$, $c \to y$.

Eine „Anpassung“ kann auch beim „(Schnell-)Kleben“ die gefundene Lösung verbessern. Hier sind zwar die Kosten der neu hinzugefügten Variante immer größer als die der beiden Varianten aus denen die neue Variante entstanden ist, dennoch können die Kosten der neuen Variante geringer sein als die anderer Varianten im Produktionsplan.
Beispiel: Betrachtet werde ein Beispiel mit vier Moduln und vier Bestellungen. Es sei $a = 1111$, $b = 0111$, $c = 0110$ und $d = 0011$. Vor dem „Kleben“ seien $x = 1111$, $y = 0110$ und $z = 0011$ die zu produzierenden Varianten. Die Zuordnung der Bestellungen zu diesen Varianten ist $a, b \to x$, $c \to y$ und $d \to z$.
Durch Verkleben von y und z entsteht die Variante $v = 0111$, die Variante b zieht um, es folgt $a \to x$, $b, c, d \to v$.

Ablaufplan beim Spalten

Die Methode „Spalten“ läuft somit insgesamt wie folgt ab:

1. Für alle bekannten $\kappa \geq \kappa'$ Produktionspläne V betrachte alle Varianten $v \in V$ die mindestens zwei Bestellungen abdecken.

2. Für alle diese Varianten betrachte mögliche Unterteilungen (maximalen Modulkombinationen oder 0-1-Zerlegung).

3. Für alle diese Unterteilungen betrachte den entstehenden Produktionsplan und führe eine Anpassung für die neu entstandenen Varianten durch.

4. Die $\kappa \geq \kappa'$ besten so entstandenen Produktionspläne sind das Ergebnis.

2.1.4 Variantenauswählende Methoden

Ist eine (kleine) Menge **potentieller Varianten** vorgegeben, so können Verfahren konstruiert werden, die sich darauf beschränken, günstige Varianten aus dieser Menge auszuwählen. Die Abdeckungsverhältnisse können in diesem Fall mit einer Kostenmatrix vorab berechnet werden, da die Liste der im Verfahren auftauchenden Varianten beschränkt ist. Analog zur Aufwärts-Abwärtsbewegung beim „Kleben“ und „Spalten“ entstehen zwei Grundtypen.

1. **Variante hinzufügen:** Es wird berechnet, welche Variante aus der Liste potentieller Produktionsvarianten die stärkste Absenkung des Unused Content bewirkt, wenn sie in den Produktionsplan aufgenommen würde.

2. **Variante entfernen:** Für jede Variante des Produktionsplans wird berechnet, welche Mehrkosten bei Entfernung dieser Variante aus dem Produktionsplan entstehen. Die von der Variante bislang abgedeckten Bestellungen b sind der kostengünstigsten im Produktionsplan verbleibenden, b abdeckenden Variante zuzuordnen. Deckt keine der verbleibenden Varianten die Bestellung ab, so ist die Entfernung der Variante mit „unendlich" hohen Kosten zu belegen.[4] Die Variante, deren Entfernung die geringsten Mehrkosten verursacht, wird aus dem Produktionsplan genommen.

Die beiden Grundtypen können auch κ-gepuffert durchgeführt werden. Dazu sind jeweils die $\kappa \geq \kappa'$ besten Möglichkeiten Varianten hinzuzufügen bzw. zu entfernen zu betrachten.

Kostenmatrix erstellen

Eine Kostenmatrix $c_{b,k}$, die für jede Bestellung b angibt, wieviel es kosten würde (einschließlich Stückzahlen) sie durch die Produktionsvariante k abzudecken, kann mit dem folgenden Algorithmus berechnet werden.

Vorgelegt seien Bestellungen $b \in B$ mit Stückzahlen l_b und potentielle Varianten $k \in K$. Berechnet wird die Matrix $(c_{b,k})_{b \in B, k \in K}$.

Für $b \in B$ führe aus:

Für $k \in K$ führe aus:

Ist $b_\mu \leq k_\mu$ für $\mu = 1, \ldots, m$ (deckt k die Bestellung b ab), so bilde

$$c_{b,k} := l_b \cdot \sum_{\mu=1}^{m} (k_\mu - b_\mu),$$

ansonsten setze $c_{b,k} = \infty$.[5]

Beispiel

Vorgegeben sei eine Bestellliste bestehen aus 8 Bestellungen für 4 Moduln.

Preis		10	5	3	2
Index	*Anzahl*	m_1	m_2	m_3	m_4
1	50	0	1	1	0
2	30	1	0	1	1
3	20	0	0	1	1
4	10	1	1	0	1
5	8	0	1	0	0
6	6	1	0	0	1
7	4	1	0	1	0
8	2	0	1	0	1

[4] Ob die Kostenzuweisung „∞" die günstigeste Methode der Behandlung dieses Falles ist, hängt von der Implementierung ab. Ggf. kann es günstiger sein, einfach zur nächsten Variante überzugehen bzw. falls keine einzige Variante mehr entfernt werden kann mit dem Ergebnis „kein Produktionsplan" (mit weniger Varianten) zu terminieren.

[5] Wie ∞ am günstigsten zu kodieren ist, hängt von den späteren Algorithmen ab, die den Fall entsprechend behandeln müssen.

Es sei $K = B$. Dann ergibt sich folgende Kostenmatrix (leere Felder stehen für ∞):

	1	2	3	4	5	6	7	8
1	0				24			
2		0	200			18	8	
3			0					
4				0	96	30		20
5					0			
6						0		
7							0	
8					16			0

Die Tabelle ist wie folgt zu lesen:

	$b \in B$
$k \in K$	$c_{b,k}$

Prozedur: Variante Hinzufügen

Mit Hilfe der Kostenmatrix $c_{b,k}$ kann dann die Prozedur „Variante Hinzufügen“ wie folgt beschrieben werden:

1. Ausgangspunkt sei ein Produktionsplan V mit p Varianten $v \in V$.

2. Für jede Variante k aus der Menge der potentiellen Varianten, die nicht bereits im Produktionsplan vorkommt, führe aus:

   (a) Setze $gewinn_k = 0$.

   (b) Für alle $b \in B$ prüfe ob $c_{b,k} < \infty$. Falls ja, bestimme das $v_b \in V$, für das $w_{b,v_b} = 1$ gilt, d.h. bestimme jene der Varianten, die b bisher abdeckt.
   Erhöhe $gewinn_k$ um $\max\{0, c_{b,v_b} - c_{b,k}\}$.

3. Bestimme jenes $k_0 \in K$ ($k_0 \notin V$) mit $gewinn_{k_0}$ maximal. Existiert kein $k \in K$ mit $gewinn_k > 0$ **stopp:** kein Produktionsplan mit mehr Varianten sinnvoll.

4. Füge k_0 zu V hinzu, ordne alle Bestellungen $b \in B$ mit $c_{b,v_b} > c_{b,k}$ der neuen Variante k_0 zu.[6]

5. Ergebnis ist ein neuer Produktionsplan V mit $p + 1$ Varianten.

Bemerkung: Die obige Prozedur liefert ausgehend von einem Produktionsplan V mit p Varianten den kostengünstigsten Produktionsplan mit genau einer neuen Variante (aus der Menge der potentiellen Varianten) ohne Änderung der „alten“ Variantenliste.
Dies wird sichergestellt indem für jede potentielle Variante die Kostensenkung für jede einzelne Bestellung vorab berechnet wird. Die Variante mit der insgesamt größten Kostenabsenkung wird verwendet.
Da keine „Anpassung“ (vgl. Seite 14) durchgeführt wird, ist das Ergebnis nicht zwingend der günstigste Produktionsplan mit $p+1$ Varianten. Denn wenn die p

[6] D.h. setze $w_{b,v_b} = 0$, $w_{b,k} = 1$.

Varianten aus dem ursprünglichen Produktionsplan auf die für die Abdeckung im neuen Produktionsplan noch notwendigen Moduln reduziert werden, könnten die Kosten möglicherweise weiter reduziert werden.

Prozedur: Variante Entfernen

Analog läuft „Varianten Entfernen“ ab:

1. Ausgangspunkt sei ein Produktionsplan V mit $p > 1$ Varianten $v \in V$.
2. Für jede Variante $v \in V$ führe aus:
   (a) Setze $kosten_v = 0$.
   (b) Für alle $b \in B$ für die $w_{b,v} = 1$ gilt, bestimme jenes $v_b \in V - \{v\}$, für dass c_{b,v_b} minimal ist.
   Erhöhe $kosten_v$ um $c_{b,v_b} - c_{b,v}$.
3. Bestimme jenes $v_0 \in V$ mit $kosten_{v_0}$ minimal. Existiert kein $v \in V$ mit $kosten_v < \infty$ **stopp:** keine Variante aus dem Produktionsplan kann mehr entfernt werden.
4. Entferne v aus V. Ordne alle Varianten mit $w_{b,v} = 1$ den alternativen Varianten v_b zu.
5. Ergebnis ist ein neuer Produktionsplan V mit $p - 1$ Varianten.

Bemerkung: Die obige Prozedur liefert ausgehend von einem Produktionsplan V mit p Varianten den kostengünstigsten Produktionsplan bei dem genau eine Variante entfernt und die restlichen Varianten belassen werden. Kann auf diese Weise kein zulässiger Produktionsplan gebildet werden, so wird dies vom Algorithmus festgestellt.
Dies wird sichergestellt indem für jede Variante des Plans der durch die Entfernung der Variante (und die damit entstehenden Mehrkosten für die Abdeckung der dieser Variante zugeordneten Bestellungen) der Kostenanstieg vorab berechnet wird. Die Variante mit den insgesamt geringsten Mehrkosten wird verwendet.
Entsteht in allen Fällen ein Produktionsplan bei dem mindestens eine Bestellung nicht mehr durch die verbliebenen Varianten abgedeckt ist, so entstehen „minimale Mehrkosten“ von ∞ mit der Konsequenz, dass „keine Variante kann entfernt werden“ gemeldet wird.

2.2 Heuristiken

In diesem Abschnitt wird eine Reihe von Heuristiken vorgestellt, die auf das Problem anwendbar sind. Alle Heuristiken arbeiten κ-gepuffert, d.h. es liegen zu jeder Zeit bis zu κ Produktionspläne für **jede** sinnvolle Variantenanzahl p vor. Sinnvoll sind Variantenanzahlen zwischen 1 und n oder, falls ein Intervall für die Variantenanzahl aufgrund von Erfahrungen gegeben ist, zwischen p_o und p_u.

Die durch eine der Heuristiken erstellten neuen Produktionspläne werden anschließend in diese Struktur einsortiert, d.h. die neu entstandenen Produktionspläne werden mit den alten Plänen (für die entsprechende Variantenanzahl) verglichen, die κ besten werden beibehalten.

Dadurch können sich die Heuristiken gegenseitig positiv beeinflussen, da alle auf Ergebnissen der anderen Heuristiken aufbauen und selber wieder (kompatible) Ergebnisse in den Prozess einbringen.

Bemerkung:

1. Beim Einsortieren neuer Produktionspläne in die Struktur wäre weiterhin ein „simulated-anealing“ denkbar. Die Regel „die κ-besten überleben“ bräuchte nur per Zufallsgenerator (teilweise) außer Kraft gesetzt werden, so dass auch ungünstigere (Zwischen-)ergebnisse eine Chance erhalten.[7]

2. Sind die Logistikkosten a_k für alle Varianten $k \in K$ bekannt, so können die entsprechenden Logistikkosten $z_k a_k$ in den vier vorgestellten Methoden („Kleben“, „Spalten“, Varianten hinzufügen, Varianten entfernen) zu den Herstellungskosten für den Produktionsplan addiert werden und so die Optimierung direkt mit den Gesamtkosten durchgeführt werden.

3. Sind in der Struktur bereits κ Produktionspläne für eine Variantenzahl p enthalten, so kann der größte auftretende Produktionspreis unter diesen Plänen als Schranke für die Berechnung weiterer Produktionspläne mit p Varianten dienen.

   Ist durch eine Abschätzung vorhersehbar, dass der durch eine der vier Methoden gebildete Produktionsplan höhere Kosten haben wird, als der κ-beste bekannte Produktionsplan, so kann auf eine exakte Berechnung verzichtet werden. Die zu untersuchende Kombination wurde durch die Schranke „ausgelotet“.[8]

2.2.1 Variantenerzeugende Heuristiken

Diese Heuristiken verwenden die Methoden „Kleben“ und „Spalten“ um aus bestehenden Kombinationen neue zu erzeugen.

Klebe-Heuristik

Mit der „Klebe“-Methode (aus Abschnitt 2.1.2) werden neue Produktionspläne in einem Intervall $[p_u, p_o]$ erzeugt:

1. Ausgangspunkt seien $1 \leq \kappa' \leq \kappa$ Produktionspläne mit p_o Varianten.

2. Setze $p = p_o$

3. Solange $p \geq p_u$ und $\kappa' \geq 0$:

[7] Im Programm wurde darauf verzichtet.

[8] Entsprechende Abschätzungsmöglichkeiten werden im Anhang bei der Dokumentation der entsprechenden Methoden beschrieben.

(a) Für alle κ' Produktionspläne mit p Varianten erzeuge mit der „Klebe“-Methode $0 \leq \kappa'' \leq \kappa$ neue Produktionspläne mit Variantenanzahl $p-1$.

(b) Setze $\kappa' := \kappa''$, $p := p-1$.

Schnellklebe-Heuristik

Berücksichtigt man beim „Kleben“ nur $\frac{g \cdot (g-1)}{2}$ Möglichkeiten (indem die beiden Schleifen der „Klebe“-Methode nur bis $g-1$ bzw. g statt $n-1$ bzw. n durchlaufen werden), kann die Rechenzeit für das „Kleben“ reduziert werden (nicht mehr $O(n^2)$, sondern $O(g^2) = O(1)$, da g nicht von der Problemgröße abhängt). Dafür sind gewisse Kombinationen nicht mehr möglich, die beim normalen „Kleben“ auftreten könnten.

Es stellte sich die Frage, welche Kombinationen berücksichtigt werden sollten. Dazu soll zunächst eine Bewertung aller auftretenden Varianten eingeführt werden.

Der **Wert** einer Variante kann sich auf die Stückzahl mit der sie produziert wird, auf den Preis der Variante oder auf das Produkt Stückzahl mal Preis beziehen. Die Stückzahl mit der eine Variante produziert wird wächst durch Hinzunehmen weiterer abzudeckender Bestellungen an. Da dabei gleichzeitig der Preis der jeweiligen Variante ansteigt, sollten keine großen Unterschiede zwischen diesen Bewertungsmöglichkeiten bestehen. Da die Produktbildung auch einen gewissen Aufwand darstellt, wurde die Stückzahl als Bewertungskriterium verwendet.

Ausgehend von dieser Bewertung der Varianten wurden drei Möglichkeiten die Kombinationsmöglichkeiten einzuschränken untersucht:

1. Die Varianten werden nach ihrem **Wert** sortiert. Die mit den g höchsten Werten sind für das Ergebnis ausschlaggebend, werden also auf Kombinationsmöglichkeiten untersucht.

2. Die Varianten werden nach ihrem **Wert** sortiert. Die mit den g kleinsten Werten haben keinen großen Einfluß auf das Ergebnis, sondern nur auf die Variantenanzahl. Der Anstieg des Unused Content durch „Verkleben“ dieser Varianten ist daher geringer als beim Verkleben von Varianten mit höheren Kosten. Diese werden daher beim „Schnellkleben“ berücksichtigt.

3. Es erfolgt keine Sortierung; es werden immer die g ersten Varianten „verklebt“, die gerade am Anfang der Datenstruktur stehen.

In der Praxis ergab sich, dass das 2. Verfahren gut ist. „Keine Sortierung“ ist eindeutig das schlechteste und langsamste Verfahren. Ursache dafür könnte die im Algorithmus entstehende Sortierung sein, die immer die gleichen Kombinationen zu „verkleben“ versucht. Zusammen mit der dadurch entstehenden späten Auslotung wird die Rechenzeit so stark erhöht, dass der Wegfall der Sortierphase nicht mehr zur Kompensation ausreicht.

Das schlechte Abschneiden des 1. Verfahrens („große Werte verkleben“) beruht vermutlich darauf, dass der Wert von zwei verklebten Varianten größer ist, als der Wert der Varianten allein. Daher werden durch dieses Verfahren nach kurzer Zeit immer nur noch eine neue Variante mit den $g-1$ bereits früher „verklebten“ Varianten kombiniert. Es entstehen $g-1$-Supervarianten (die sehr teuer sind und sehr viele Bestellungen abdecken), was insbesondere bei $g << p$ kritisch ist (p ist die Anzahl der Varianten des Produktionsplanes für den die gefundenen Kosten verglichen wurden). So lieferte für $p = 50$, $g = 100$ das 1. Verfahren ungefähr die gleichen Herstellungskosten wie das 2. Verfahren. Das 2. Verfahren ergab jedoch mit $g = 100$ ungefähr die gleichen Herstellungskosten wie für $g = 10$. Das 2. Verfahren ist daher dem ersten überlegen, da ohne Verschlechterung des Ergebnises der g-Wert[9] kleiner gewählt werden kann.

Insgesamt kann gesagt werden, dass die „Schnellklebe“-Heuristik die beste der untersuchten Heuristiken ist, um schnell gute Lösungen zu finden. Diese können anschließend vor allem als obere Schranke für andere Verfahren verwendet werden.

Spalten-Heuristik

Die „Spalten“-Heuristik arbeitet analog zur „Kleben“-Heuristik, nur dass diesmal bei $p = p_u$ begonnen und die Methode „Spalten“ verwendet wird:

1. Ausgangspunkt seien $1 \leq \kappa' \leq \kappa$ Produktionspläne mit p_u Varianten.
2. Setze $p = p_u$
3. Solange $p \leq p_o$ und $\kappa' \geq 0$:
   (a) Für alle κ' Produktionspläne mit p Varianten erzeuge mit der „Spalten“-Methode $0 \leq \kappa'' \leq \kappa$ neue Produktionspläne mit Variantenanzahl $p + 1$.
   (b) Setze $\kappa' := \kappa''$, $p := p + 1$.

Da die „Spalten“-Methode nicht alle möglichen Unterteilungen berücksichtigt, könnte auch von „Schnellspalten“ (in Analogie zum „Schnellkleben“, wo auch nicht alle möglichen Kombinationen untersucht werden) gesprochen werden. Diesem Namen wird die Heuristik in der Praxis jedoch nicht gerecht.[10]

Die beiden möglichen Unterteilungen beim „Spalten“ lieferten bei den numerischen Experimenten Beiträge zur Optimallösung. Somit kann keine der beiden Möglichkeiten als überflüssig charakterisiert werden.

2.2.2 Variantenauswählende Heuristiken

Betrachtet man nur eine vorgegebene (kleine) Liste potentieller Varianten, so können Verfahren darauf reduziert werden, Varianten zum Produktionsplan hinzuzunehmen bzw. herauszunehmen, wobei jeweils auf Abdeckung und Kosten zu achten ist.

Abdeckung und Kosten (bzw. Gewinn beim Hinzufügen einer Variante) können vorab einmalig durchgerechnet werden, da die Anzahl der Varianten in diesem Fall begrenzt ist. Der dazugehörige Algorithmus wurde in Abschnitt 2.1.4 vorgestellt.

Die entstehenden Heuristiken lehnen sich an die in der Literatur unter dem Namen **Add und Drop** bekannten Eröffnungsverfahren an.[11] Im Unterschied

[9] im Programm auch mit ***goal*** bezeichnet
[10] Sie ist sehr langsam.
[11] vgl. z.B. [5, Seite 38ff]

zu den dort beschriebenen Verfahren kann bei dem gegebenen Problem jedoch nur eine Teilmenge aller Standorte (hier als potentielle Varianten bezeichnet) betrachtet werden. Da außerdem die Fixkosten a_k für den Standort $k \in K$ als nicht bekannt unterstellt wurden, ist kein Abbruchkriterium für die beiden Heuristiken vorhanden. Es kann nicht festgestellt werden, ab wann das Hinzufügen bzw. Wegnehmen weiterer Standorte (bzw. bei uns: potentielle Varianten) die Kosten nicht weiter absenkt. Daher wird nicht bei einer Variantenanzahl p „irgendwo in der Mitte“ abgebrochen, sondern erst, wenn entweder die Kosten[12] auf ∞ ansteigen bzw. wenn durch Hinzufügen weiterer Varianten die Kosten nicht mehr abgesenkt werden können (oder wenn ein vorgegebener Bereich $[p_u, p_o]$ verlassen wird).

Die Heuristiken „Varianten hinzufügen“ bzw. „Varianten entfernen“ laufen in Analogie zu den „Kleben“- bzw. „Spalten“-Heuristiken ab:

1. Ausgangspunkt seien $1 \leq \kappa' \leq \kappa$ Produktionspläne mit p_u bzw. p_o Varianten.

2. Setze $p = p_u$ bzw. $p = p_o$

3. Solange $p \leq p_o$ bzw. $p \geq p_u$ und $\kappa' \geq 0$:

   (a) Für alle κ' Produktionspläne mit p Varianten erzeuge durch Hinzufügen bzw. Entfernen von Varianten $0 \leq \kappa'' \leq \kappa$ neue Produktionspläne mit Variantenanzahl $p+1$ bzw. $p-1$.[13]

   (b) Setze $\kappa' := \kappa''$, $p := p+1$ bzw. $p := p-1$.

Im Unterschied zu den „Kleben“- und „Spalten“-Heuristiken besteht bei der „Varianten hinzufügen“-Heuristik die Möglichkeit, dass sich die gefundenen Lösungen gegenseitig beeinflussen. Beim „Kleben“ und „Spalten“ werden die κ Produktionspläne zunächst völlig isoliert betrachtet. Erst beim Einsortieren der κ besten Produktionspläne in die Struktur werden die Ergebnisse in Beziehung gesetzt. Der eigentliche Prozess der Konstruktion von Produktionsplänen bezieht sich somit immer nur auf einen der κ Produktionpläne.

Bei der „Varianten hinzufügen“-Heuristik hingegen können sich die Produktionspläne gegenseitig beeinflussen. Da eine **globale** Liste potentieller Varianten geführt wird (die aus **allen** bekannten Produktionsplänen gebildet wird), erfolgt bei der Konstruktion eines Produktionsplanes insbesondere auch die Betrachtung von Varianten der **anderen** Produktionspläne. Somit kann die Heuristik „Varianten hinzufügen“ Produktionspläne in gewisser Weise „mischen“.

In numerischen Ergebnissen zeigte sich, dass die variantenauswählenden Heuristiken mit der Bestellliste als Menge der potentiellen Varianten bereits gute Ergebnisse liefern. Da diese Vorgehensweise dann jedoch die meisten Varianten ausschließt, sollte es mit variantenerzeugenden Verfahren, z.B. mit der „(Schnell-)Klebe“-Heuristik, kombiniert werden, um weiter verbesserte Ergebnisse zu erzielen.

[12] Steigen die Kosten für eine Variante auf ∞, so steigen auch die Kosten für den gesamten Produktionsplan auf ∞. Abgebrochen wird, wenn „Varianten Entfernen“ für **jede** der Varianten $v \in V$ unendliche Kosten (d.h. keine Abdeckung mehr möglich) verursacht.

[13] Entsprechende Prozeduren wurden im Abschnitt 2.1.4 beschrieben.

2.3 Anwendungsstrategie

In diesem Abschnitt soll eine mögliche Strategie zur Anwendung der vorgestellten Heuristiken erläutert und motiviert werden.

2.3.1 Ausgangslösung

Als Ausgangslösung bieten sich die im Lemma 1.1 eingeführte Einhüllende E_B (bezogen auf die Bestellliste B) oder die gesamte Bestellliste als Produktionsplan an.

$V = \{E_B\}$ hat den Vorteil, dass bei hohen Standortkosten durch „Spalten“ bzw. „Varianten hinzufügen“ in wenigen Schritten eine erste Lösung für eine geringe Anzahl p von Standorten (Varianten) gefunden werden kann, und zwar unabhängig von der Anzahl der Bestellungen.

$V = B$ hat den Vorteil, dass die „Schnellklebe“-Heuristik angewendet werden kann. Diese liefert sehr schnell Produktionspläne für alle denkbaren p. Diese können dann insbesondere auch den anderen Heuristiken zur Auslotung dienen. Da die „Schnellklebe“-Heuristik im Vergleich „Spalten“ und „Varianten hinzufügen“ nur sehr wenig Zeit kostet, ist somit $V = B$ die bessere Startlösung.

2.3.2 Schnellkleben

Ein weiterer Vorteil dieser Vorgehensweise ist, dass anhand der von der „Schnellklebe“-Heuristik gelieferten Produktionspläne ggf. der Bereich $[p_u, p_o]$, in dem optimiert werden soll, festgesetzt werden kann, da die Heuristik eine erste Abschätzung der Kosten liefert.

Außerdem wird auf diese Weise garantiert, dass für jede (sinnvolle) Variantenanzahl p zumindest ein Produktionsplan bekannt ist, von dem aus die anderen Heuristiken starten können.

2.3.3 Kleben

Nach der Festlegung eines Arbeitsbereiches $[p_u, p_o]$ sollte dann die „Klebe“-Heuristik angewendet werden. Durch Betrachtung aller Kombinationen (beim „Schnellkleben“ wurde nur ein Teil der Kombinationsmöglichkeiten zugelassen) können ggf. die Lösungen der „Schnellklebe“-Heuristiken noch verbessert werden. Auch benötigt diese Heuristik (auch auf Grund der durch das Schnellkleben entstandenen Schranken) nur wenig Rechenzeit.

Durch die Festlegung eines Arbeitsbereiches spielt auch hier die Anzahl der Bestellungen für die Rechenzeit nur eine untergeordnete Rolle.

2.3.4 Iteriertes Optimieren

Nun sollten die Heuristiken angewendet werden, um in einem kleinen vorgegebenem Intervall $[p_u, p_o]$ besonders gute Produktionspläne zu erhalten. Da die Anwendung eines einzelnen Typs von Heuristiken schnell keine weitere Verbesserung liefert, müssen alle vier Grundtypen („Spalten“, „Kleben“, „Varianten entfernen“, „Varianten hinzufügen“) iteriert angewendet werden. Der Aufwand hier ist groß, da auch die „langsamen“ Heuristiken verwendet werden müssen.

Insbesondere die Anzahl der Bestellungen beeinflußt die Laufzeit von „Spalten" und „Varianten hinzufügen" stark.

Kann durch iteriertes Anwenden der variantenauswählenden und variantenerzeugenden Heuristiken keine ausreichend gute Lösung gefunden werden, so sollte der κ-Wert und/oder das Intervall $[p_u, p_o]$ vergrößert werden.[14]

2.3.5 Ablaufplan

Insgesamt ergibt sich folgender Ablauf:

1. Startlösung $V = B$.
2. Schnellkleben.
3. Intervall $[p_u, p_o]$ wählen.
4. Kleben im Intervall $[p_u, p_o]$
5. Solange noch Rechenzeit[15] vorhanden **und** die Kosten der Produktionspläne zu hoch **und** sich mindestens ein Produktionsplan seit dem letzten Durchlauf geändert hat:
   (a) Varianten entfernen (im Intervall $[p_u, p_o]$).
   (b) Varianten hinzufügen (im Intervall $[p_u, p_o]$).
   (c) Kleben (im Intervall $[p_u, p_o]$).
   (d) Spalten (im Intervall $[p_u, p_o]$).
6. Falls Lösungen zu schlecht: κ vergrößern, gehe zu 2

Da „Kleben", „Spalten" und „Varianten entfernen" für den gleichen Produktionsplan immer die gleichen Ergebnisse liefern brauchen diese Heuristiken nur für die geänderten Produktionspläne durchgeführt werden.

Die „Varianten hinzufügen"-Heuristik hingegen hängt auch noch von der Liste der potentiellen Varianten ab. Daher ist diese auf alle Produktionspläne erneut anzuwenden, wenn sich die Liste der potentiellen Varianten geändert hat.

2.4 Beispiel

In diesem Abschnitt soll das auf Seite 16 eingeführte Beispiel entsprechend der vorgeschlagenen Strategie durchgerechnet werden. Die κ-besten Produktionspläne für jede Variantenanzahl werden in einem Tableau gespeichert. Dabei wird jeder Produktionsplan durch die in ihm enthaltenen Varianten identifiziert. Die Varianten werden durch den dem Nibble, der die Modulkombination beschreibt, zugeordnetem Hexadezimalwert beschrieben:

[14] Beide Maßnahmen vergrößern die Vielfalt der Varianten. Für das Intervall $[1, n]$ und $k = 2^m$ garantiert bereits „Kleben" das Auffinden einer Optimallösung.

[15] Das Kriterium „Rechenzeit" ist theoretisch nicht notwendig (aufgrund der ganzzahligen Eingangsdaten ist Endlichkeit garantiert).

Kombination				Nibble
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	A
1	0	1	1	B
1	1	0	0	C
1	1	0	1	D
1	1	1	0	E
1	1	1	1	F

Ausgangsdaten

Als Liste der potentiellen Varianten L sei zunächst die Bestellliste B angenommen. Startlösung sei ebenfalls B. Weiterhin sei $\kappa = 2$ gewählt. Im Tableau werden die Stückzahlen als Indizes und die UC-Kosten in Klammern notiert. Damit ergibt sich für das Starttableau folgende Situation:

p	1. Produktionsplan	2. Produktionsplan
1		
2		
3		
4		
5		
6		
7		
8	$6_{50}, B_{30}, 3_{20}, D_{10}, 4_8, 9_6, A_4, 5_2(0)$	

Die Liste der potentiellen Varianten ist somit $L = \{3, 4, 5, 6, 9, A, B, D\}$.

Schnellkleben

Für das Schnellkleben sei $g = 3$ gewählt. Die Liste der potentiellen Varianten soll nach den Stückzahlen sortiert vorliegen. Für $g = 3$ werden dann die Kombinationsmöglichkeiten $5_2^7 + A_4^{13} = F_6^{20}$, $5_2^7 + 9_6^{12} = D_8^{17}$ sowie $A_4^{13} + 9_6^{12} = B_{10}^{15}$ untersucht.[16] Die UC-Kosten betragen für die drei Möglichkeiten $UC(5 + A) = 28 + 26 = 54$, $UC(5 + 9) = 20 + 30 = 50$ und $UC(A + 9) = 18 + 8 = 26$. Die

[16] Angabe in der Form $Variante_{Anzahl}^{Kosten}$

Übersicht über die κ besten Produktionspläne ändert sich wie folgt:

p	1. Produktionsplan	2. Produktionsplan
1		
2		
3		
4		
5		
6		
7	$6_{50}, B_{30}, 3_{20}, B_{10}, D_{10}, 4_8, 5_2(26)$	$6_{50}, B_{30}, 3_{20}, D_{10}, 4_8, D_8, A_4(50)$
8	$6_{50}, B_{30}, 3_{20}, D_{10}, 4_8, 9_6, A_4, 5_2(0)$	

Die Doppelnennung von B tritt auf, da durch das Schnellkleben die Kombinationsmöglichkeiten $B + 5$ bzw. $B + 9$ nicht untersucht wurden. Hier zeigt sich der Nachteil beim Schnellkleben. Möglicherweise werden sehr günstige Kombinationen nicht untersucht. Die Liste der potentiellen Varianten ist nun $L = \{3, 4, 5, 6, 9, A, B, D\}$.

Im nächsten Schritt werden für den Produktionsplan 7/1 die Kombinationen $5_2^7 + 4_8^5 = 5_{10}^7$, $5_2^7 + D_{10}^{17} = D_{12}^{17}$ und $4_8^5 + D_{10}^{17} = D_{18}^{17}$ betrachtet. Die UC-Kosten betragen $UC(5+4) = 26 + 16 = 42$, $UC(5+D) = 26 + 20 = 46$ und $UC(4+D) = 26 + 96 = 122$.

Für den Produktionsplan 7/2 werden die Kombinationen $A_4^{13} + D_8^{17} = F_{30}^{20}$, $4_8^5 + D_8^{17} = D_{16}^{17}$ sowie $A_4^{13} + 4_8^5 = E_{21}^{18}$ betrachtet. Die UC-Kosten betragen $UC(A+D) = 50 + 28 + 24 = 102$, $UC(4+D) = 50 + 55 = 105$ und $UC(A+4) = 50 + 20 + 104 = 174$.

Letztendlich folgt:

p	1. Produktionsplan	2. Produktionsplan
1	$F_{130}(1302)$	
2	$7_{80}, F_{50}(502)$	$F_{80}, 6_{50}(702)$
3	$F_{50}, 6_{50}, D_{30}(402)$	$F_{50}, 6_{50}, B_{30}(552)$
4	$6_{50}, B_{30}, 7_{30}, F_{20}(252)$	$6_{50}, F_{30}, B_{30}, 3_{20}(252)$
5	$6_{50}, B_{30}, 3_{20}, F_{20}, 5_{10}(122)$	$6_{50}, B_{30}, 3_{20}, D_{20}, B_{10}(142)$
6	$6_{50}, B_{30}, 3_{20}, D_{10}, B_{10}, 5_{10}(42)$	$6_{50}, B_{30}, 3_{20}, B_{10}, D_{12}, 4_8(46)$
7	$6_{50}, B_{30}, 3_{20}, B_{10}, D_{10}, 4_8, 5_2(26)$	$6_{50}, B_{30}, 3_{20}, D_{10}, 4_8, D_8, A_4(50)$
8	$6_{50}, B_{30}, 3_{20}, D_{10}, 4_8, 9_6, A_4, 5_2(0)$	

Die Liste der potentiellen Varianten ist danach

$$L = \{3, 4, 5, 6, 7, 9, A, B, D, F\}.$$

Kleben

Nach dem Kleben (im Intervall $[1, 8]$) sieht die Situation wie folgt aus:

p	1. Produktionsplan	2. Produktionsplan
1	$F_{130}(1302)$	
2	$7_{80}, F_{50}(502)$	$7_{70}, F_{60}(602)$
3	$7_{80}, B_{40}, D_{10}(272)$	$7_{70}, B_{40}, D_{20}(342)$
4	$6_{50}, B_{40}, D_{20}, 3_{20}(142)$	$6_{50}, B_{40}, 7_{30}, D_{10}(172)$
5	$6_{50}, B_{40}, 3_{20}, D_{10}, 5_{10}(42)$	$6_{50}, B_{30}, 3_{20}, F_{20}, 5_{10}(122)$
6	$6_{50}, B_{34}, 3_{20}, D_{10}, 5_{10}, 9_6(24)$	$6_{50}, B_{30}, 3_{20}, D_{10}, B_{10}, 5_{10}(42)$
7	$6_{50}, B_{34}, 3_{20}, D_{10}, 4_8, 9_6, 5_2(8)$	$6_{50}, B_{30}, 3_{20}, D_{10}, 5_{10}, 9_6, A_4(16)$
8	$6_{50}, B_{30}, 3_{20}, D_{10}, 4_8, 9_6, A_4, 5_2(0)$	

Varianten entfernen

Ab jetzt soll nur noch das Intervall $[1, 4]$ betrachtet werden.

Im Plan 4/1 kann nur die Variante 3_{20}^5 (abgedeckt durch B^{15}) entfernt werden. Alle anderen Varianten können durch die verbleibenden Varianten nicht mehr abgedeckt werden. Die Entfernung von 3 erhöht den unused-content um $20 \cdot 10 = 200$. Es entsteht keine Lösung die besser wäre als die beiden bekannten Lösungen für $p = 3$.

Im Plan 4/2 kann ebenfalls die Variante 6_{50}^8 (abgedeckt durch 7^{10}) entfernt werden. Dies erhöht den unused-content um $50 \cdot 2 = 100$. Die so entstehende Variante mit UC-Kosten 272 ist jedoch bereits bekannt. Weniger offensichtlich ist, dass auch die Variante 7 entfernt werden kann. Diese von 7^{10} bisher abgedeckten Bestellungen (3_{20}^5, 4_8^5 und 5_2^7) werden von den verbleibenden Varianten abgedeckt ($3_{20}^5 \to B^{15}$, $4^5 \to 6^8$, $5^7 \to D^{17}$). Es entsteht der Produktionsplan B_{60}, 6_{58} und D_{12} mit unused-content $172 + 20 \cdot (15 - 10) + 8 \cdot (8 - 10) + 2 \cdot (17 - 10) = 270$.

Bei den Produktionsplänen 3/1 und 3/2 kann kein Fortschritt erzielt werden. 2/1 und 2/2 brauchen (da der Produktionsplan für $p = 1$ eindeutig ist) nicht einmal geprüft werden. Es folgt das Tableau:

p	1. Produktionsplan	2. Produktionsplan
1	$F_{130}(1302)$	
2	$7_{80}, F_{50}(502)$	$7_{70}, F_{60}(602)$
3	$B_{60}, 6_{58}, D_{12}(270)$	$7_{80}, B_{40}, D_{10}(272)$
4	$6_{50}, B_{40}, D_{20}, 3_{20}(142)$	$6_{50}, B_{40}, 7_{30}, D_{10}(172)$

Varianten hinzufügen

Beim „Varianten hinzufügen“ ändert sich nichts mehr.

Abbruch des Verfahrens

Das Verfahren würde nun mit „Kleben“ und „Spalten“ weiterlaufen und das Tableau weiter verbessern. Danach würden sich die obigen Schritte (ggf. mit größerem Wert κ) wiederholen bis zufriedenstellende Ergebnisse erreicht werden (oder keine Rechenzeit mehr zur Verfügung steht).

Für $a_k = 250$ für alle $k \in K$ folgt unmittelbar, dass $7_{80}, F_{50}$ mit Gesamtkosten $2 \cdot 250 + 502 = 1002$ die Optimallösung im obigen Tableau ist. Die Rechnung im nächsten Kapitel zeigt, dass in diesem Fall die Heuristiken die globale Optimallösung gefunden haben.

Kapitel 3

Das Verfahren von Erlenkotter

In diesem Kapitel soll das Verfahren von Erlenkotter vorgestellt werden. Das Verfahren von Erlenkotter ist das beste bekannte Verfahren zur Lösung des „Uncapacitated Facility Location“ Problems.[1]

3.1 Modell

Zu Beginn sei das in Abschnitt 1.2.6 entwickelte Modell (1.7) von Seite 5 zitiert:

$$\begin{cases} \text{minimiere} & \sum_{(b,k)\in F} w_{b,k} c_{b,k} + \sum_{k\in K} a_k z_k \\ \text{so dass} & \sum_{(b,k)\in F} w_{b,k} = 1 \text{ alle } b \in B \\ & w_{b,k} \leq z_k \text{ alle } (b,k) \in F \\ & w_{b,k} \in \{0,1\} \text{ alle } (b,k) \in F \\ & z_k \in \{0,1\} \text{ alle } k \in K \end{cases}$$

Dabei sind a_k und $c_{b,k} \in \mathbb{N}$ vorgegeben. F besteht aus all jenen Kombinationspaaren (b,k), bei denen $b_\mu \leq k_\mu$ für alle $\mu = 1, \dots, m$. Das Problem wird relaxiert, indem $w_{b,k} \geq 0$ und $z_k \geq 0$ anstelle der Ganzzahligkeitsbedingungen gefordert wird. F bestehe aus allen denkbaren Kombinationen $F = B \times K$.[2] Es folgt:

$$\begin{cases} \text{minimiere} & \sum_{b\in B} \sum_{k\in K} w_{b,k} c_{b,k} + \sum_{k\in K} a_k z_k \\ \text{so dass} & \sum_{k\in K} w_{b,k} = 1 \text{ alle } b \in B \\ & z_k - w_{b,k} \geq 0 \text{ alle } b \in B, k \in K \\ & w_{b,k} \geq 0 \text{ alle } b \in B, k \in K \\ & z_k \geq 0 \text{ alle } k \in K \end{cases} \tag{3.1}$$

[1] vgl. [3, S. 388] oder auch [11, S. 126]

[2] Nicht-Abdeckungen sind dabei (in Analogie zu den Big-M-Verfahren) durch sehr hohe Einträge in der $c_{b,k}$-Matrix darzustellen.

3.2 Dualisierung

Um die Ableitung der dualen Aufgabe deutlich zu machen, sei hier zunächst das Tableau der primalen Aufgabe angegeben:[3]

	$b \in B, k \in K$	$k \in K$		
	$w_{b,k}$	z_k		
min	$c_{b,k}$	a_k	0	
$0 = v_i$	$\delta_{i,b}$	0	1	$i \in B$
$u_{i,j}$	$\delta_{i,b} \cdot \delta_{j,k}$	$-\delta_{j,k}$	0	$i \in B, j \in K$

(3.2)

Dieses Problem kann umgeformt werden zu:

	$b \in B, k \in K$	$k \in K$		
	$w_{b,k}$	z_k		
min	$c_{b,k}$	a_k	0	
v_i^1	$\delta_{i,b}$	0	1	$i \in B$
v_i^2	$-\delta_{i,b}$	0	-1	$i \in B$
$u_{i,j}$	$\delta_{i,b} \cdot \delta_{j,k}$	$-\delta_{j,k}$	0	$i \in B, j \in K$

(3.3)

Das duale Tableau lautet dann:

	$i \in B$	$i \in B$	$i \in B, j \in K$		
	v_i^1	v_i^2	$u_{i,j}$		
max	1	-1	0	0	
$w_{b,k}$	$-\delta_{i,b}$	$\delta_{i,b}$	$-\delta_{i,b}\delta_{j,k}$	$c_{b,k}$	$b \in B, k \in K$
z_k	0	0	$\delta_{j,k}$	a_k	$k \in K$

(3.4)

Es folgt:

	$i \in B$	$i \in B, j \in K$		
	v_i^*	$u_{i,j}$		
max	1	0	0	
$w_{b,k}$	$\delta_{b,i}$	$-\delta_{i,b}\delta_{j,k}$	$c_{b,k}$	$b \in B, k \in K$
z_k	0	$\delta_{j,k}$	a_k	$k \in K$

(3.5)

wobei v_i^* nicht vorzeichenbeschränkt ist. In Gleichungsform lautet die dualisierte Aufgabe somit:

$$\left\{ \begin{array}{ll} \text{maximiere} & \sum_{b \in B} v_b \\ \text{so dass} & \sum_{b \in B} u_{b,k} \leq a_k \text{ alle } k \in K \\ & v_b - u_{b,k} \leq c_{b,k} \text{ alle } b \in B, k \in K \\ & u_{b,k} \geq 0 \text{ alle } b \in B, k \in K \end{array} \right. \tag{3.6}$$

3.3 Vereinfachung

Die Aufgabe (3.6) kann vereinfacht werden. Die Zielfunktion hängt nur von v ab. Daher ändert sich eine Lösung $(v_b, u_{b,k})$ nicht, wenn $u_{b,k}$ auf einen kleineren, zulässigen Wert gesetzt wird. Aufgrund der Bedingungen $v_b - u_{b,k} \leq c_{b,k}$ (oder

[3] Dabei sei $\delta_{a,b} = 1$ falls $a = b$ und $\delta_{a,b} = 0$ falls $a \neq b$. $v_i = 0$ ist im Sinne der Big-M-Verfahren zu interpretieren.

äquivalent $u_{b,k} \geq v_b - c_{b,k}$) sowie $u_{b,k} \geq 0$ ergibt sich als kleinster zulässiger Wert für $u_{b,k}$:

$$u_{b,k} := \max\{0, v_b - c_{b,k}\} \tag{3.7}$$

Diese Setzung (für eine zulässige Lösung $(v_b, u_{b,k})$) ändert nichts an der Zulässigkeit der Lösung (da die Bedingung $\sum_{b\in B} u_{b,k} \leq a_k$ dadurch, dass $u_{b,k}$ auf den kleinstmöglichen Wert *erniedrigt* wird, nicht verletzt werden kann).

Die Aufgabe (3.6) wird vereinfacht zu:

$$\begin{cases} \text{maximiere} & z_D = \sum_{b\in B} v_b, \\ \text{so dass} & \sum_{b\in B} \max\{0, v_b - c_{b,k}\} \leq a_k \text{ alle } k \in K \end{cases} \tag{3.8}$$

Satz 3.1 *Für Optimallösungen von* (3.1) *und* (3.8) *gilt immer:*

$$z_k \left(a_k - \sum_{b\in B} \max\{0, v_b - c_{b,k}\} \right) = 0 \qquad k \in K \tag{3.9}$$

$$(z_k - w_{b,k}) \max\{0, v_b - c_{b,k}\} = 0 \qquad k \in K, b \in B \tag{3.10}$$

Beweis: Angenommen $w^*_{b,k}$ und z^*_k seien eine Optimallösung von (3.1), v^*_b eine Optimallösung von (3.8). Nach dem Dualitätssatz sind dann die Zielfunktionswerte gleich, d.h.

$$\begin{aligned}
0 &= Z_P(w^*_{b,k}, z^*_k) - Z_D(v^*_b) \\
&= \sum_{b\in B}\sum_{k\in K} w^*_{b,k} c_{b,k} + \sum_{k\in K} a_k z^*_k - \sum_{b\in B} v^*_b \\
&= \sum_{b\in B}\sum_{k\in K} w^*_{b,k} c_{b,k} + \sum_{k\in K} a_k z^*_k - \sum_{b\in B} v^*_b - \sum_{b\in B}\sum_{k\in K} u_{b,k} w^*_{b,k} + \sum_{b\in B}\sum_{k\in K} u_{b,k} w^*_{b,k} \\
&\quad - \sum_{k\in K} z^*_k \cdot \left(- \sum_{b\in B} u^*_{b,k} \right) + \sum_{k\in K} z^*_k \cdot \left(- \sum_{b\in B} u^*_{b,k} \right) \\
&\quad - \sum_{b\in B} v^*_b \cdot \left(\sum_{k\in K} w^*_{b,k} \right) + \sum_{b\in B} v^*_b \cdot \left(\sum_{k\in K} w^*_{b,k} \right) \\
&= \sum_{b\in B}\sum_{k\in K} u^*_{b,k} \cdot (z^*_k - w^*_{b,k}) + \underbrace{\sum_{b\in B}\sum_{k\in K} \underbrace{w^*_{b,k}}_{\geq 0} \cdot (c_{b,k} + u^*_{b,k} - v^*_b)}_{\geq 0} \\
&\quad - \sum_{b\in B} v^*_b \cdot \underbrace{\left(1 - \sum_{k\in K} w^*_{b,k} \right)}_{=0} + \sum_{k\in K} z^*_k \cdot \left(a_k - \sum_{b\in B} u^*_{b,k} \right) \\
&\geq \sum_{b\in B}\sum_{k\in K} u^*_{b,k} \cdot (z^*_k - w^*_{b,k}) + \sum_{k\in K} z^*_k \cdot \left(a_k - \sum_{b\in B} u^*_{b,k} \right) \\
&= \sum_{k\in K} \left(z_k \left(a_k - \sum_{b\in B} \max\{0, v_b - c_{b,k}\} \right) \right) \\
&\quad + \sum_{k\in K}\sum_{b\in B} ((z_k - w_{b,k}) \max\{0, v_b - c_{b,k}\})
\end{aligned}$$

Da alle Summanden ≥ 0 sind, folgen die angegebenen Bedingungen.

□

Mit Hilfe dieser Bedingungen kann zu **jeder** vorgegebenen (3.8)-zulässigen Lösung v eine Lösung $(z_k^+, w_{b,k}^+)$ zu (3.1) angegeben werden.

Dazu ist v zunächst durch die Prozedur aus Abschnitt 3.4 zu verbessern. Das Ergebnis sei v^+. Für v^+ sei die Menge $K^+ \subset K$ definiert als

$$K^+ := \left\{ k \in K : \sum_{b \in B} \max\{0, v_b^+ - c_{b,k}\} = a_k \right\}. \tag{3.11}$$

Für die von der Prozedur gelieferte Lösung v^+ gilt dann,[4] dass für jedes $b \in B$ ein $k \in K^+$ existiert, so dass

$$v_b^+ \geq c_{b,k}. \tag{3.12}$$

Weiterhin wurde der Zielfunktionswert von v durch Anwendung der Prozedur höchstens erhöht (also verbessert).

Für jedes $b \in B$ sei die Variante $k^+(b) \in K^+$ festgelegt, die kostenminimal ist:

$$c_b^+ := c_{b,k^+(b)} := \min\{c_{b,k} : k \in K^+\} \tag{3.13}$$

Lemma 3.2 *Bei dieser Konstruktion gilt:*

$$v_b^+ - c_b^+ \geq 0 \tag{3.14}$$

Beweis: Sei $k_0 \in K^+$ so gewählt, dass c_{b,k_0} die Voraussetzung (3.12) erfüllt. Dann gilt:

$$\begin{aligned}
& v_b^+ - c_b^+ \\
=& v_b^+ - \min\{c_{b,k} : k \in K^+\} \\
\geq& v_b^+ - c_{b,k_0} \\
\geq& 0 \qquad \text{nach Voraussetzung (3.12)}
\end{aligned}$$

□

Satz 3.3 *Definiert man nun*

$$z_k^+ := \begin{cases} 1, & k \in K^+ \\ 0, & \textit{sonst} \end{cases}$$

$$w_{b,k}^+ := \begin{cases} 1, & k = k^+(b), b \in B \\ 0, & \textit{sonst} \end{cases}$$

so ist damit eine (3.1)*-zulässige Lösung festgelegt, die die Bedingung* (3.9) *erfüllt. Die Bedingungen* (3.10) *sind genau dann verletzt, wenn* $v_b^+ > c_{b,k}$ *gilt für ein* $k \in K^+ - \{k^+(b)\}$.

[4]nach Satz 3.10, der die Prozedur charakterisiert

Beweis:

1. (3.1)-Zulässigkeit ist gegeben, denn

$$\sum_{k \in K} w_{b,k}^{+} = 1 \qquad \text{für alle } b \in B$$

ist klar nach Definition von $w_{b,k}^{+}$.

$$z_k^{+} - w_{b,k}^{+} \geq 0 \qquad \text{für alle } b \in B, k \in K$$

ist klar, da $k = k^{+}(b)$ nur für $k \in K^{+}$ sein kann.

$$w_{b,k}^{+} \geq 0 \qquad z_k \geq 0$$

sind trivial nach Konstruktion.

2. (3.9) ist nach Definition von z_k^{+} äquivalent zu

$$a_k = \sum_{b \in B} \max\{0, v_b - c_{b,k}\}$$

für alle $k \in K^{+}$. Dies ist aber gerade die Definition von K^{+}.

3. (3.10) lautet

$$(z_k - w_{b,k}) \cdot \max\{0, v_b - c_{b,k}\} = 0.$$

Da $z_k^{+} - w_{b,k}^{+} \neq 0$ für $k \in K^{+} - \{k^{+}(b)\}$, $b \in B$ nach Definition, folgt die Behauptung unmittelbar.

□

Der Satz korrespondiert mit dem folgenden Lemma, das den Unterschied zwischen dem Zielfunktionswert z_D^{+} der dualen Lösung und dem Zielfunktionswert z_P^{+} der primalen Lösung beziffert.

Lemma 3.4 *Erfüllt die Lösung v^{+} die Forderung* (3.12) *und wurde die dazugehörige primale Lösung gemäß dem Satz 3.3 konstruiert, so gilt:*

$$z_P^{+} - z_D^{+} = \sum_{b \in B} \sum_{\substack{k \in K^{+} \\ k \neq k^{+}(b)}} \max\{0, v_b^{+} - c_{b,k}\} \tag{3.15}$$

Beweis:

$$
\begin{aligned}
z_D^+ &= z_D^+ + \sum_{k\in K^+}\left(a_k - \sum_{b\in B}\max\{0, v_b^+ - c_{b,k}\}\right)\\
&= z_D^+ + \sum_{k\in K^+} a_k - \sum_{b\in B}\sum_{k\in K^+}\max\{0, v_b^+ - c_{b,k}\}\\
&= z_D^+ + \sum_{k\in K^+} a_k - \sum_{b\in B}\sum_{\substack{k\in K^+\\ k\neq k^+(b)}}\max\{0, v_b^+ - c_{b,k}\} - \sum_{b\in B}\max\{0, \underbrace{v_b^+ - c_{b,k^+(b)}}_{\geq 0}\}\\
&= \sum_{b\in B} v_b^+ + \sum_{k\in K^+} a_k + \sum_{b\in B}(c_b^+ - v_b^+) - \sum_{b\in B}\sum_{\substack{k\in K^+\\ k\neq k^+(b)}}\max\{0, v_b^+ - c_{b,k}\}\\
&= \underbrace{\sum_{\substack{k\in K\\ b\in B}} c_{b,k}w_{b,k}^+ + \sum_{k\in K} a_k z_k^+}_{=z_P^+} - \sum_{\substack{b\in B\\ k\in K^+}}\max\{0, v_b^+ - c_{b,k}\}
\end{aligned}
$$

□

Für eine dual zulässige Lösung v, die gemäß dem im Abschnitt 3.4 beschriebenem Algorithmus in eine (3.12) genügende Lösung v^+ überführt wurde, ist somit die Bedingung

$$\sum_{b\in B}\sum_{\substack{k\in K^+\\ k\neq k^+(b)}}\max\{0, v_b^+ - c_{b,k}\} = 0 \tag{3.16}$$

hinreichend dafür, dass eine gemäß obiger Vorschrift konstruierte primale Lösung die Bedingungen (3.9) und (3.10) erfüllt, und somit optimal ist.

Ziel der Überlegungen, eine optimale Lösung von (3.1) zu finden, sollte also sein, den Term

$$\sum_{b\in B}\sum_{\substack{k\in K^+\\ k\neq k^+(b)}}\max\{0, v_b^+ - c_{b,k}\}$$

möglichst klein zu machen. Wird er Null kann mit dem Satz 3.3 die Optimallösung der primalen Aufgabe konstruiert werden.

3.4 Anstiegsprozedur

Im folgenden wird nun eine Prozedur benannt, die eine dual zulässige Lösung ermittelt, auf deren Grundlage wie oben beschrieben eine (primale) Lösung von (3.1) berechnet werden kann.

Das Ziel ist also aus einer (3.8)-zulässigen Lösung v eine **zulässige** Lösung v^+ mit **besserem** (oder gleichem) Zielfunktionswert zu konstruieren, die die **Forderung** (3.12) **erfüllt**.

Zur geeigneten Formulierung seien die Kosten für jedes $b \in B$ geordnet. Dazu sei für jedes $b \in B$ eine **Ordnung** auf K definiert. Die entsprechenden Elemente $k \in K$ können dann mit entsprechenden Zahlen $j \in \mathbb{N}$ identifiziert werden:

$$c_{b,1} \leq c_{b,2} \leq \ldots \leq c_{b,j_0} \leq c_{b,j_0+1} = \infty,$$

wobei c_{b,j_0+1} künstlich eingeführt seien, um Fallunterscheidungen zu vermeiden.

Als Anfangslösung der Prozedur kann man $v_b = c_{b,1}$ wählen (alle $b \in B$), wenn keine andere zulässige Lösung vorgelegt wird. Es gilt immer $c_{b,1} < \infty$ da sonst keine zulässige Lösung existieren würde (jede Bestellung muß von mindestens einer Variante abgedeckt werden). Die Prozedur wird so formuliert, dass nur eine (willkürlich vorgewählte, nicht leere) Teilmenge $B^+ \subset B$ für Verbesserungen zugelassen wird.[5]

3.4.1 Prozedur: Dualer Anstieg

1. Ausgangspunkt ist eine dual zulässige Lösung v für die o.B.d.A. $v_b \geq c_{b,1}$ für alle $b \in B$ gilt.[6] Es sei
$$s_k := a_k - \sum_{b \in B} \max\{0, v_b - c_{b,k}\} \tag{3.17}$$
für alle $k \in K$. Für jedes $b \in B$ setze [7]
$$k(b) := \min\{k \in K | v_b < c_{b,k}\}. \tag{3.18}$$

2. Setze Flagge $\delta := 0$. Für alle $b \in B^+$ (in einer frei gewählten Reihenfolge) führe durch:
   (a) Setze $\triangle_b := \min\{s_k | v_b - c_{b,k} \geq 0, k \in K\}$.[8]
   (b) Ist $\triangle_b > c_{b,k(b)} - v_b$, setze $\triangle_b := c_{b,k(b)} - v_b$, $\delta := 1$ und erhöhe $k(b)$ um 1.
   (c) Senke s_k um $\triangle_b$ ab für jedes $k \in K$, für das $v_b - c_{b,k} \geq 0$ und erhöhe dann v_b um $\triangle_b$.

3. Ist $\delta = 1$ gehe zu 2, sonst **stop**.

3.4.2 Analyse der Prozedur

Lemma 3.5 *In der Prozedur gilt bei Schritt 2a immer*
$$\min\{k \in K | v_b \leq c_{b,k}\} \leq k(b) \leq \min\{k \in K | v_b < c_{b,k}\}.$$

Beweis: $k(b)$ wird im ersten Schritt entsprechend definiert. Daher genügt es, die Änderung von $k(b)$ im Schritt 2b sowie die Änderungen von v_b im Schritt 2c zu betrachten.

1. In Schritt 2b wird möglicherweise $k(b)$ um 1 erhöht, was die zweite Ungleichung verletzen könnte. In diesem Fall wird aber im Schritt 2c v_b um $\triangle_b = c_{b,k(b)} - v_b$ erhöht. Damit gilt dann für das alte $k(b)$
$$v_b = c_{b,k(b)}.$$

[5] Die Forderung, dass für jedes $b \in B$ ein $k \in K^+$ existiert, so dass $v \geq c_{b,k}$ ist dann allerdings nur für $b \in B^+$ garantiert erfüllt.

[6] Sonst erhöhe v_b auf $c_{b,1}$. Die Zulässigkeit ändert sich dadurch nicht.

[7] Gegenüber der ursprünglichen Formulierung von Erlenkotter wurde $v_b \leq c_{b,k}$ durch $v_b < c_{b,k}$ ersetzt und der nachfolgende Schritt „gilt dabei $v_b = c_{b,k}$, so setze $k(b) := k(b) + 1$" dadurch überflüssig gemacht.

[8] Die Menge $k \in K$ mit $v_b - c_{b,k} \geq 0$ ist nach Voraussetzung nicht leer.

Da die $c_{b,k}$ als geordnet vorausgesetzt waren folgt

$$v_b \leq c_{b,k(b)+1} < c_{b,k_0},$$

wobei k_0 das kleinste k mit $v_b < c_{b,k}$ sei. Ein solches existiert, da $c_{b,j+1} = \infty$ gesetzt wurde. Da aufgrund der Anordnung der $c_{b,k}$ dann $k(b)+1 \leq k_0$ sein muss, bleibt die zweite Ungleichung nach der Erhöhung gültig.

2. Die erste Ungleichung könnte ungültig werden, wenn v_b um $\triangle_b$ erhöht wird und dadurch $v_b > c_{b,k(b)}$ wird. Dies ist jedoch nicht möglich, denn die Erhöhung von v_b wird im Schritt 2b auf ein $\triangle_b$ beschränkt, das maximal eine Erhöhung auf $v_b = c_{b,k(b)}$ ergibt.

□

Lemma 3.6 *s_k gibt in der Prozedur jederzeit die Differenz*

$$s_k = a_k - \sum_{b \in B} \max\{0, v_b - c_{b,k}\}$$

für alle $k \in K$ an und es gilt immer $s_k \geq 0$.

Beweis: s_k wird im ersten Schritt genau so definiert. Aufgrund der Zulässigkeit von v gilt an dieser Stelle $s_k \geq 0$. Daher genügt es, die (einzige) Änderung von s_k und v_b im Schritt 2c um $\triangle_b$ zu betrachten.
s_k ist immer ≥ 0, denn es werden immer nur die s_k um $\triangle_b$ verkleinert für die $v_b \geq c_{b,k}$ gilt, wobei die Absenkung $\triangle_b$ im Schritt 2a auf den kleinsten dieser s_k-Werte beschränkt wurde:

$$\triangle_b = \min\{s_k | v_b - c_{b,k} \geq 0, k \in K\}$$

Damit folgt bereits $s_k \geq 0$.
Schritt 2b stellt nun sicher, dass die Menge der $k \in K$ mit $v_b - c_{b,k}$ über die der Minimierungsprozess geführt wurde, durch die Änderung von v_b nicht „wesentlich“ geändert wird. Die Erhöhung wird auf einen Wert beschränkt, so dass für jedes k, für das vorher $v_b < c_{b,k}$ galt, nach der Erhöhung höchstens $v_b = c_{b,k}$ gilt.
Das kleinste k, auf das die Prozedur „aufpassen“ muss, wird durch $k(b)$ beschrieben. Denn nach Lemma 3.5 gilt für alle $k > k(b)$

$$v_b \leq c_{b,k(b)} \leq c_{b,k}.$$

Alle $k < k(b)$ hingegen sind unkritisch, denn es gilt bereits

$$v_b \geq c_{b,k}.$$

Daher wird die Erhöhung auf

$$\triangle_b \leq c_{b,k(b)} - v_b.$$

beschränkt. Durch diese Wahl von $\triangle_b$ ist somit sichergestellt, dass für kein $k \in K$ vor der Erhöhung $v_b < c_{b,k}$ und danach $v_b > c_{b,k}$ gilt.

Die Änderung von v_b um $\triangle_b$ kann sich somit nur entweder vollständig auf s_k auswirken (es gilt $v_b - c_{b,k} \geq 0$ und s_k wird um $\triangle_b$ abgesenkt) oder gar nicht (vorher gilt $v_b - c_{b,k} < 0$ und nachher gilt $v_b - c_{b,k} \leq 0$, s_k bleibt unverändert).

□

Lemma 3.7 *Die beschriebene Prozedur führt höchstens zu einer Erhöhung des Zielwertes der dual zulässigen Startlösung v.*

Beweis: v_b wird in der Prozedur höchstens um Werte $\triangle_b$ erhöht, und nach Lemma 3.6 ist $\triangle_b \geq 0$.

□

Korollar 3.8 *In der Prozedur ist v zu jeder Zeit eine* (3.8)*-dual zulässige Lösung.*

Beweis: Zu Beginn der Prozedur ist nach Voraussetzung v dual zulässig, es gilt also

$$\sum_{b \in B} \max\{0, v_b - c_{b,k}\} \leq a_k \text{ für alle } k \in K$$

Dies ist äquivalent zu

$$a_k - \sum_{b \in B} \max\{0, v_b - c_{b,k}\} \geq 0 \text{ für alle } k \in K$$

Nach Lemma 3.6 gilt in der Prozedur jederzeit

$$s_k = a_k - \sum_{b \in B} \max\{0, v_b - c_{b,k}\} \text{ für alle } k \in K$$

und $s_k \geq 0$. Die Behauptung folgt unmittelbar.

□

Lemma 3.9 *Die Prozedur terminiert nach endlich vielen Schritten.*

Beweis: Es genügt zu zeigen, dass für alle b irgendwann δ nicht mehr gleich 1 gesetzt wird. δ wird aber nur genau dann gleich 1 gesetzt, wenn $\triangle_b > c_{b,k(b)} - v_b$ gilt.
In diesem Fall wird jedoch jedesmal $k(b)$ um 1 erhöht. Damit steigt $c_{b,k(b)}$ nach spätestens j Durchgängen auf ∞ an. v_b ist jedoch durch $a_k + c_{b,1}$ beschränkt, denn wenn $v_b > a_k + c_{b,1}$ wäre, folgt $s_k \leq a_k - (v_b - c_{b,1}) < 0$.
Somit gilt nach spätestens j Durchgängen $\triangle_b < c_{b,k(b)} - v_b$ und δ kann nicht mehr auf 1 gesetzt werden. Das Verfahren bricht ab.

□

Satz 3.10 *Gilt $B^+ = B$, so erfüllt die duale Lösung v bei Abbruch der Prodedur die Bedingungen, die an die Konstruktion einer zugehörigen primalen Lösung gestellt wurden, insbesondere existiert für jedes $b \in B$ ein $k \in K^+$, das die Gleichung* (3.12) *erfüllt.*

Beweis: Nach Korollar 3.8 kann die Prozedur nur mit einer zulässigen dualen Lösung enden.
K^+ ist definiert als die Menge der $k \in K$, für die

$$a_k = \sum_{b \in B} \max\{0, v_b - c_{b,k}\}$$

gilt. Nach Lemma 3.6 ist somit $s_k = 0$ äquivalent zu $k \in K^+$.
Angenommen es gäbe ein $b \in B$ mit $v_b < c_{b,k}$ für alle $k \in K^+$. Dann wird im Schritt 2a $\triangle_b$ auf einen Wert > 0 gesetzt (da $s_k > 0$ für $k \notin K^+$).
Danach gilt entweder $\triangle_b > c_{b,k(b)} - v_b$ und es wird $\triangle_b$ auf $c_{b,k(b)} - v_b$ beschränkt. Dabei ist $c_{b,k(b)} - v_b > 0$ und es wird weiter $\delta = 1$ gesetzt. Also kann die Prozedur noch nicht beendet sein.
Gilt hingegen $\triangle_b \leq c_{b,k(b)} - v_b$, so wird in Schritt 2c v_b echt angehoben. Obiges läuft so lange, bis letztendlich $v_b < c_{b,k}$ gilt.

$\square$

3.4.3 Beispiel

Im folgenden soll das Beispiel aus Abschnitt 2.1.4 behandelt werden. Im Arbeitstableau werden folgende Werte geführt:

	b	s
k	$c_{b,k}$	s_k
v	v_b	z_D

z_D ist dabei der duale Zielfunktionswert. Es gelte $a_k = 250$ für alle $k \in K$. $c_{b,k}$ ist die bereits im Abschnitt 2.1.4 berechnete Kostenmatrix.

	1	2	3	4	5	6	7	8	s		
1	0				24				250^{0}	0^{1}	0^{9}
2		0	200			18	8		250^{0}	0^{2}	0^{10}
3			0						250^{0}	50^{3}	50^{11}
4				0	96	30		20	250^{0}	0^{4}	0^{12}
5					0				250^{0}	234^{5}	226^{13}
6						0			250^{0}	232^{6}	232^{14}
7							0		250^{0}	242^{7}	242^{15}
8					16			0	250^{0}	230^{8}	$222^{13,16}$
	0	0	0	0	0	0	0	0	0		
v	250^{1}	250^{2}	200^{3}	250^{4}	16^{5}	18^{6}	8^{7}	20^{8}		1012	
	250^{9}	250^{10}	200^{11}	250^{12}	24^{13}	18^{14}	8^{15}	20^{16}			1020

Potenzen wurden im Tableau benutzt, um die Reihenfolge der Berechnung kenntlich zu machen. Werte ohne Potenz sind Ausgangsdaten. Bei der Initialisierung produzierte Daten wurden mit der Potenz 0 gekennzeichnet. Im Algorithmus erfolgt nach dem letzten gezeigten Durchlauf noch einer, da $\delta = 1$ im

Schritt 13 gesetzt wurde. In diesem letzten Durchlauf ändert sich wegen der 0 am Ende der ersten Zeile nichts mehr.

Die zugehörige primale Lösung kann unmittelbar abgelesen werden:

$$K^+ = \{1, 2, 4\}$$

Die optimalen Abdeckungen sind demnach:

b	1	2	3	4	5	6	7	8
c_b^+	0	0	200	0	24	18	8	20
$k^+(b)$	1	2	2	4	1	2	2	4

Die primale Lösung kostet

$$\underbrace{3 \cdot 250}_{=z_k a_k} + \underbrace{200 + 24 + 18 + 8 + 20}_{w_{b,k} \cdot c_{b,k}} = 1020.$$

Damit ist $z_D = z_P$ und die Optimallösung der Aufgabe mit beschränkter Variantenliste gefunden.

Als Motivation für den nächsten Abschnitt wird das Beispiel nun so erweitert, dass eine Dualitätslücke entsteht. Dazu wird K um zwei Varianten erweitert: $k(9) = 1111$, $k(10) = 0111$. Aufbauend auf der Beobachtung aus Lemma 3.4 wird im nächsten Abschnitt die Lösung weiter verbessert.

	1	2	3	4	5	6	7	8	s		
1	0				24				250	150	0
2		0	200			18	8		250	100	0
3			0						250	150	50
4				0	96	30		20	250	220	70
5					0				250	234	226
6						0			250	232	232
7							0		250	242	242
8					16			0	250	244	236
9	600	150	300	30	120	48	28	26	250	250	0
10	100		100		40			6	250	250	0
	0	0	0	0	0	0	0	0	0		
v	100	150	100	30	16	18	8	6		428	
	250	250	200	180	24	18	8	6			936

Die zugehörige primale Lösung lautet diesmal ($K^+ = \{1, 2, 9, 10\}$):

b	1	2	3	4	5	6	7	8
c_b^+	0	0	100	30	24	18	8	6
$k^+(b)$	1	2	10	9	1	2	2	10

Die Kosten betragen diesmal $4 \cdot 250 + 186 = 1186$. Die optimale Lösung liegt somit im Intervall $[936, 1186]$.

3.5 Lösungsverbesserung

3.5.1 Einschränkung von K^+

Die erste Verbesserung bezieht sich auf die Konstruktion einer primalen Lösung aus der dualen Lösung v^+.

Bislang wurden als Produktionsvarianten die Menge K^* aller $k \in K$ verwendet, für die nach Abschluß des Anstiegsverfahrens $s_k = 0$ gilt (bisher mit K^+ bezeichnet). Die Anforderung an die Menge kann jedoch ggf. mit einer kleineren Menge $K^+ \subset K^*$ erfüllt werden. Da für jedes $k \in K^+$ Kosten a_k anfallen, kann dadurch ggf. die Lösung verbessert werden.

Im folgenden werde die Menge K^+ wie folgt konstruiert:

1. alle **wesentlichen** Varianten aus K^* werden in K^+ einbezogen, d.h. alle $k \in K^*$, für die es bei irgendeinem $b \in B$ nur ein einziges $k \in K$ gibt mit $v_b^+ \geq c_{b,k}$.
2. sind alle wesentlichen Varianten in K^+ aufgenommen, so wird die Menge B systematisch durchsucht, ob es ein $b \in B$ gibt, für das **kein** $k \in K^+$ existiert mit $v_b^+ \geq c_{b,k}$. Dann wird ein $\overline{k} \in K^*$ in K^+ aufgenommen für das $c_{b,\overline{k}}$ minimal ist. Nach Satz 3.10 gilt dann $v_b^+ \geq c_{b,\overline{k}}$.

Nach Erlenkotter erweist sich diese einfache Auswahlstrategie (die nicht zwingend die beste Menge $K^+ \subset K^*$ liefert) als günstig, da sie schnell eine gute Lösung findet.

3.5.2 Verkleinerung der Dualitätslücke

Das folgende Verfahren versucht, die in Lemma 3.4 bezifferte Dualitätslücke

$$z_P^+ - z_D^+ = \sum_{b \in B} \sum_{\substack{k \in K^+ \\ k \neq k^+(b)}} \max\{0, v_b^+ - c_{b,k}\}$$

zu verkleinern, indem die bei Abbruch des Anstiegsverfahrens erhaltene duale Lösung nachträglich noch verbessert wird.

Dazu wird ein $b \in B$ gesucht, für das die Komplementaritätsbedingung (3.10)

$$(z_k - w_{b,k}) \max\{0, v_b - c_{b,k}\} = 0 \qquad k \in K, b \in B$$

verletzt ist, für das es also mindestens zwei $k \in K^+$ gibt mit $v_b^+ > c_{b,k}$. Wird dieses v_b^+ abgesenkt, werden mindestens zwei Schlupfvariablen wieder positiv und geben Raum für die Erhöhung anderer v_b.

Die folgende Prozedur stellt sicher, dass die durch die Anhebung der Schlupfvariablen verursachte Absenkung des dualen Zielfunktionswerts anderweitig wieder rückgängig gemacht wird, dass sich z_D also höchstens erhöht.

Für die Formulierung der Prozedur sind folgende Definitionen hilfreich:

$$K_b^* := \{k \in K^* | v_b \geq c_{b,k}\},$$
$$K_b^+ := \{k \in K^+ | v_b > c_{b,k}\}$$

für alle $b \in B$. Ferner werde der „zweitbeste“ c-Wert bezeichnet mit

$$\tilde{c}_b := c_{b,\tilde{k}(b)} := \min_{\substack{k \in K^+ \\ k \neq k^+(b)}} c_{b,k}$$

(sofern $|K_b^+| > 1$).[9] Der größte relevante c-Wert sei

$$\overline{c}_b := c_{b,\overline{k}(b)} := \max_{k \in K}\{c_{b,k} | v_b > c_{b,k}\}. \tag{3.19}$$

[9] $k^+(b)$ wurde auf Seite 32, Gleichung (3.13) eingeführt.

Schließlich sei für alle $k \in K$

$$B_k^+ := \{b | K_b^* = \{k\}\}.$$

3.5.3 Prozedur: Duale Anpassung

Für alle $b \in B$ führe durch:

1. gilt $|K_b^+| > 1$ und $B_{k^+(b)}^+ \cup B_{\tilde{k}(b)}^+ \neq \emptyset$, so führe aus:

   (a) Für jedes $k \in K_b^+$ erhöhe s_k um $v_b - \overline{c}_b$ und senke v_b auf den Wert $\overline{c}_b$ ab.

   (b) i. Setze $B^+ = B_{k^+(b)}^+ \cup B_{\tilde{k}(b)}^+$ und führe das duale Anstiegsverfahren durch.

   ii. Setze $B^+ = B^+ \cup \{b\}$ und führe erneut das duale Anstiegsverfahren durch.

   iii. Setze $B^+ = B$ und wiederhole nochmals das duale Anstiegsverfahren.

   (c) Hat v_b noch nicht wieder seinen ursprünglichen Wert erreicht, gehe zurück zu 1.[10]

3.5.4 Analyse der Prozedur

Lemma 3.11 *Ist v noch nicht Optimallösung, so ist die Bedingung $|K_b^+| > 1$ für mindestens ein $b \in B$ erfüllt.*

Beweis: Da v nicht Optimallösung ist, besteht eine Dualitätslücke wie in Lemma 3.4 angegeben. Damit existiert mindestens ein $b \in B$ mit

$$\sum_{\substack{k \in K^+ \\ k \neq k^+(b)}} \max\{0, v_b^+ - c_{b,k}\} > 0$$

In der Anordnung der $c_{b,k}$ wurde das kleinste $k \in K^+$ mit $k \neq k^+(b)$ mit $\tilde{k}(b)$ bezeichnet. Also muss zumindest gelten

$$v_b^+ - c_{b,\tilde{k}(b)} > 0$$

(denn für alle anderen $k \in K^+$ gilt $c_{b,\tilde{k}(b)} \leq c_{b,k}$).
Es folgt $\tilde{k}(b), k^+(b) \in K_b^+$, also $|K_b^+| > 1$.

□

Die Menge K_b^+ ist die Menge der $k \in K^+$ durch die v_b „beschränkt" wurde. Wenn $|K_b^+| > 1$ enthält diese Menge zumindest $k^+(b)$ und $\tilde{k}(b)$. Diese sind die kleinsten $c_{b,k}$-Werte, wenn für mindestens zwei Werte die Differenz $v_b^+ - c_{b,k}$ positiv ist, dann sind diese somit darunter.

[10] Dieser Schritt führt dazu, das v_b solange behandelt wird, bis eine weitere Anhebung auf keinen Fall mehr eine weitere Verbesserung bewirken kann.

Die Mengen $B^+_{k^+(b)} \cup B^+_{\tilde{k}(b)}$ sind damit eine ausgezeichnete Teilmenge jener $b' \in B$, für die durch die Absenkung der zu b gehörigen s_k neuer Spielraum für die Erhöhung der $v_{b'}$ entsteht.

Bemerkung: Es ist **nicht** garantiert, dass $B^+_{k^+(b)} \cup B^+_{\tilde{k}(b)} \neq \emptyset$ für irgendein $b \in B$. Die Prozedur kann somit versagen, d.h. trotz bestehender Dualitätslücke keine Verbesserung ergeben. Daher ist das im nächsten Abschnitt vorgestellte Branch & Bound-Verfahren ein notwendiger Bestandteil des Verfahrens.

Das folgende Lemma beschreibt, dass die Prozedur, sofern sie überhaupt etwas verändert, zumindest keine Verschlechterung bewirkt.

Lemma 3.12 *Die Änderung von s_k und v_b im Schritt 1a erhält die Beziehung*

$$s_k = a_k - \sum_{b \in B} \max\{0, v_b - c_{b,k}\}$$

und ermöglicht die Erhöhung von mindestens einem anderen Wert des Vektors v.

Beweis: Nach Definition ist $c_{b,\overline{k}}$ der größte Wert für den gilt $v_b > c_{b,k}$. Wird daher v_b auf $c_{b,\overline{k}} = \overline{c}_b$ abgesenkt, so gilt für alle $k \in K$ für die vorher $v_b > c_{b,k}$ galt danach zumindest noch $v_b \geq c_{b,k}$.
Die entsprechenden s_k-Werte müssen daher alle um die Differenz $v_b - \overline{c_b}$ angehoben werden damit die Beziehung

$$s_k = a_k - \sum_{b \in B} \max\{0, v_b - c_{b,k}\}$$

gültig bleibt.
Unter diesen k-Werten gibt es mindestens zwei für die vorher $s_k = 0$ galt. Dies sind wie oben erläutert $k^+(b)$ und $\tilde{k}(b)$. Die bislang durch einen der dazugehörigen s_k-Werte ausschließlich Beschränkten v_b sind durch die Mengen $B^+_{k^+(b)}$ und $B^+_{\tilde{k}(b)}$ angegeben, die nach der Voraussetzung aus dem Schritt 1 zumindest ein Element $b' \in B^+_{k^+(b)} \cup B^+_{\tilde{k}(b)}$ enthalten. Das dazugehörige $v_{b'}$ kann somit angehoben werden.

□

Lemma 3.13 *Der Zielfunktionswert wird durch die Prozedur insgesamt nicht verschlechtert, d.h. $\sum_{b \in B} v_b$ wird höchstens größer.*

Beweis: Schritt 1(b)ii stellt sicher, dass von den $v_{b'}$ nicht genutztes Potential durch die Absenkung von v_b nicht zu einer Verschlechterung des Zielfunktionswertes führt. Denn wenn nicht alle s_k-Werte im Schritt 1(b)i auf 0 abgesenkt wurden, wird nun die verbleibende Differenz genutzt um v_b anzuheben.
Eine mögliche Absenkung der s_k-Werte im Schritt 1(b)i hat aber ein gleichzeitiges mindestens gleich großes Ansteigen der Gesamtsumme der $\sum_{b \in B} v_b$ zur Folge. Es folgt die Behauptung.

□

Lemma 3.14 *Die Prozedur endet mit einer dual zulässigen, die Gleichung (3.12) erfüllenden Lösung v.*

Beweis: Da in der Prozedur s_k nur angehoben wird, nach Lemma 3.12 die Beziehung zwischen s_k und v_b erhalten bleibt und die Anwendung der Anstiegsprozedur nach Korollar 3.8 ebenfalls die Zulässigkeit erhält, ist v jederzeit zulässige Lösung.
Nach Satz 3.10 liefert die Anwendung von 1(b)iii eine dualen Lösung die die Gleichung (3.12) erfüllt.

□

3.5.5 Beispiel

Das letzte Beispiel aus Abschnitt 3.4.3 soll nun mit diesem Verfahren fortgesetzt werden. Es ist $K^+ = K^* = \{1, 2, 9, 10\}$. Für $b = 1$ gilt $K_1^+ = \{1, 10\}$ und $B_1^+ = \{5\}$ und $B_{10}^+ = \{10\}$.

	1	2	3	4	5	6	7	8	s					
1	0				24				0	150	134	78	54	4
2		0	200			18	8		0	0	0	0	0	0
3			0						50	50	50	50	50	50
4				0	96	30		20	70	70	70	64	40	40
5					0				226	226	210	154	130	130
6						0			232	232	232	232	232	232
7							0		242	242	242	242	242	242
8					16			0	236	236	206	144	120	120
9	600	150	300	30	120	48	28	26	0	0	0	0	0	0
10	100		100		40			6	0	150	136	74	50	0
	250	250	200	180	24	18	8	6	936					
	100	250	200	180	24	18	8	6						
	100	250	200	180	40	18	8	20						
	100	250	200	180	96	18	8	26						
	100	250	200	180	120	18	8	26						
	150	250	200	180	120	18	8	26						952

An dieser Stelle ist der Algorithmus mit dem Fall $b = 1$ zu Ende, aber noch nicht beendet! Die duale und die primale Lösung haben sich jedoch schon angenähert, denn die zugehörige primale Lösung lautet ($K^* = K^+ = \{2, 9, 10\}$):

b	1	2	3	4	5	6	7	8
c_b^+	100	0	100	30	40	18	8	6
$k^+(b)$	10	2	10	9	10	2	2	10

Die Kosten betragen dabei $3 \cdot 250 + 302 = 1052$.

Die Untersuchung für $b = 2$ führt zu $K_2^+ = \{2, 9\}$, $B_2^+ = \{6, 7\}$ und $B_9^+ = \emptyset$. Das Verfahren läuft wie folgt weiter:

	1	2	3	4	5	6	7	8	s				
1	0				24				4	4	4	4	4
2		0	200			18	8		0	100	68	50	0
3			0						50	50	50	50	50
4				0	96	30		20	40	40	40	22	0
5					0				130	130	130	130	130
6						0			232	232	220	202	202
7							0		242	242	222	172	172
8					16			0	120	120	120	120	120
9	600	150	300	30	120	48	28	26	0	100	100	50	28
10	100		100		40			6	0	0	0	0	0
	150	250	200	180	120	18	8	26	952				
	150	150	200	180	120	18	8	26					
	150	150	200	180	120	30	28	26					
	150	150	200	180	120	48	78	26					
	150	150	200	202	120	48	78	26					974

Die zugehörige primale Lösung lautet ($K^* = K^+ = \{2, 4, 10\}$):

b	1	2	3	4	5	6	7	8
c_b^+	100	0	100	0	40	18	8	6
$k^+(b)$	10	2	10	4	10	2	2	10

Die Kosten betragen dabei unverändert $3 \cdot 250 + 272 = 1022$. Für die nächsten $b \in B$ gilt $|K_3^+| = 1$, $|K_4^+| = 1$ aber $K_5^+ = \{4, 10\}$ mit $B_{10}^+ = \{1\}$, $B_4^+ = \{4\}$. Also läuft das Verfahren weiter:

	1	2	3	4	5	6	7	8	s		
1	0				24				4	28	4
2		0	200			18	8		0	0	0
3			0						50	50	50
4				0	96	30		20	0	24	0
5					0				130	154	154
6						0			202	202	202
7							0		172	172	172
8					16			0	120	144	144
9	600	150	300	30	120	48	28	26	28	28	4
10	100		100		40			6	0	24	0
	150	150	200	202	120	48	78	26	974		
	150	150	200	202	96	48	78	26			
	174	150	200	226	96	48	78	26			998

Die primale Lösung bleibt unverändert. Es gilt $K_6^+ = \{2, 4\}$ mit $B_4^+ = \{4\}$,

$B_2^+ = \{7\}$.

	1	2	3	4	5	6	7	8	s			
1	0				24				4	4	4	4
2		0	200			18	8		0	18	18	4
3			0						50	50	50	50
4				0	96	30		20	0	18	14	0
5					0				154	154	154	154
6						0			202	220	220	206
7							0		172	172	172	172
8					16			0	144	144	144	144
9	600	150	300	30	120	48	28	26	4	4	0	0
10	100		100		40			6	0	0	0	0
	174	150	200	226	96	48	78	26	998			
	174	150	200	226	96	30	78	26				
	174	150	200	230	96	30	78	26				
	174	150	200	230	96	44	78	26				998

Die zugehörige primale Lösung lautet ($K^+ = \{9, 10\}$):

b	1	2	3	4	5	6	7	8
c_b^+	100	150	100	30	40	48	28	6
$k^+(b)$	10	9	10	9	10	9	9	10

Die Gesamtkosten liegen damit bei $2 \cdot 250 + 502 = 1002$.

Es gilt $K_7^+ = \{9\}$, $K_8^+ = \{4, 10\}$ mit $B_4^+ = \{6\}$, $B_{10}^+ = \{1\}$.

	1	2	3	4	5	6	7	8	s			
1	0				24				4	4	0	0
2		0	200			18	8		4	4	0	0
3			0						50	50	50	50
4				0	96	30		20	0	6	2	0
5					0				154	154	154	154
6						0			206	206	202	202
7							0		172	172	172	172
8					16			0	144	144	144	142
9	600	150	300	30	120	48	28	26	0	0	0	0
10	100		100		40			6	0	6	2	0
	174	150	200	230	96	44	78	26	998			
	174	150	200	230	96	44	78	20				
	178	150	200	230	96	48	78	20				
	178	150	200	230	96	48	78	22				1002

Die primale Lösung bleibt unverändert. Da die Dualitätslücke geschlossen wurde, ist das Verfahren damit beendet.

Erweitert man das letzte Tableau um die fehlenden Varianten (vgl. Seite 16), so zeigt sich, dass die gefundene Lösung eine globale Optimallösung ist:

	1	2	3	4	5	6	7	8	s
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	0
1000									250
0010									250
0001									250
1100					80				234
1110	500				104		20		192
	178	150	200	230	96	48	78	22	1002

Das Programm im Anhang kommt innerhalb weniger Sekunden zu diesem Ergebnis.[11]

3.6 Branch & Bound

Die bisher beschriebenen Verfahren liefern primale und duale Schranken für die Optimallösung des gestellten Problems. Es bietet sich daher an, eine vollständige Optimierung im Rahmen eines Branch & Bound-Verfahrens durchzuführen.[12]

3.6.1 Verzweigung

Die Verzweigung wird durchgeführt, indem zu gegebener Testaufgabe für ein ausgewähltes $k \in K$ alternativ die Restriktionen $z_k = 0$ oder $z_k = 1$ hinzugenommen werden. $z_k = 0$ realisiert man im dualen Verfahren am einfachsten, indem $a_k = \infty$ gesetzt wird. $z_k = 1$ kann mit $a_k = 0$ realisiert werden.

$a_k = 0$ garantiert nicht in allen Fällen, dass der Standort k tatsächlich gewählt wird (es folgt nicht zwingend $z_k = 1$). Für die Verzweigung genügt dies dennoch, denn jede auf Basis von $a_k = 0$ gefundene Lösung mit $z_k = 0$ ist in Bezug auf alle Zuordnungen von Bestellungen (und insbesondere damit auch in Bezug auf alle Kosten) mit der gleichen Lösung nur mit $z_k = 1$ identisch.

Für die Wahl von k zur Verzweigung kann das beschriebene duale Verfahren auf die Testaufgabe angewendet werden. Stellt sich heraus, dass die gefunden Lösung die Komplementaritätsbedingungen verletzt, so muss ein b existieren, für das die Bedingungen verletzt sind. Dann kann (wie D. Erlenkotter vorschlägt) $k = k^+(b)$ zur Verzweigung ausgewählt werden.

Die Verzweigung $z_k = 0$ sollte zuerst untersucht werden, da hier die zusätzliche Auslotungsmöglichkeit auftritt, dass keine Abdeckung mehr möglich ist. Ob für die Verzweigungen insgesamt LIFO oder FIFO die bessere Strategie ist, hängt wohl vom zur Verfügung stehenden Speicher ab, FIFO benötigt tendenziell mehr Speicher.

3.6.2 Auslotung

Jede Hinzunahme einer Restriktion zu einer primalen Testaufgabe erhöht den optimalen Zielfunktionswert potentiell. Auslotung liegt daher dann vor, wenn

[11] Passend formulierte Eingangsdaten wurden als Beispiel „C“ im *data/*-Verzeichnis abgelegt.

[12] zu Branch & Bound Verfahren allgemein vergleiche [1, Seite 247ff] oder auch [4, 1. Kapitel]

eine untere Schranke für den optimalen Zielfunktionswert zu hoch ist, wenn also der aktuelle duale Zielfunktionswert größer oder gleich dem besten primalen Zielfunktionswert ist.

Ferner liegt Auslotung vor, wenn durch Entfernen zu vieler potentieller Varianten ($z_k = 0$) keine Abdeckung der Bestellmenge mehr möglich ist. Dies äußert sich jedoch im Algorithmus dadurch, dass für eine (oder mehrere) Bestellungen in der jeweiligen Spalte keine Zeile k mit einem Eintrag $c_{b,k} < \infty$ existiert. Formal folgt $v_b = \infty$. Somit tritt der Zielfunktionswert ∞ auf.

Beim formalen Rechnen mit Big-M (statt ∞) tritt somit Auslotung auf. Der Fall braucht nicht gesondert behandelt werden.

3.6.3 Ein Algorithmus

Das Branch & Bound-Verfahren könnte z.B. wie folgt aussehen:

1. Gestartet wird mit einem Problem P_0, bei dem alle z_k frei wählbar sind. Die Liste der offenen Knoten sei am Anfang $LIST := \{P_0\}$.

2. Wähle ein Problem P aus $LIST$. Ist $LIST$ leer, **stopp**. Löse das Problem P mit dem Verfahren von Erlenkotter. Entferne P aus $LIST$.

3. Ist der duale Zielfunktionswert für das Problem P größer als die durch die beste bekannte Optimallösung gegebene Schranke, gehe zu 2.

4. Überprüfe ob die gefundene Lösung die Komplementariatätsbedingungen erfüllt. Falls

   ja überprüfe, ob die Lösung besser ist als die beste bisher gefundene Lösung. Korrigiere ggf. ist die Schranke.

   nein Wähle ein b, für daß die Bedingungen verletzt sind und ein zugehöriges, bisher nicht festgelegtes $k = k^+(b)$. Für dieses k füge in $LIST$ ein:

   (a) Das Problem P_-, dass entsteht, wenn in P $z_k = 0$ gesetzt wird (oder äquivalent $a_k = \infty$).

   (b) Das Problem P_+, dass entsteht, wenn in P $z_k = 1$ gesetzt wird (oder $a_k = 0$).

5. Gehe zu 2.

3.6.4 Die Variantenmenge K

Bleibt die Frage zu klären, wie die Menge der potentiellen Varianten K sukzessive ausgeweitet werden kann, so dass letztendlich eine optimale Lösung des Variantenoptimierungsproblems gefunden wird. Weil $|K| = 2^m$ die Anzahl der denkbaren Varianten sehr schnell sehr groß wird, sprengt die vollständige Menge K die Kapazitätsgrenzen.

Näherungen an die Optimallösung sollten zunächst mit einer Teilmenge $L \subset K$ bestimmt werden, insbesondere um schnell gute Schranken zu erhalten. Die Frage, welche k dann in L aufgenommen werden sollen, um ein gegen die Optimallösung möglichst schnell konvergentes Verfahren zu erhalten, wird im nächsten Kapitel behandelt.

Kapitel 4

Ein vollständiges Verfahren

4.1 Idee

Heuristiken liefern mögliche Optimallösung

Die vorgestellten Heuristiken („Kleben“, „Spalten“, „Varianten entfernen“, „Varianten hinzufügen“ und das Verfahren von Erlenkotter) liefern für ein vorgegebenes κ eine Liste potentieller Varianten L (alle Varianten die im Laufe des Verfahrens betrachtet wurden) sowie einen Produktionsplan für den sich die Dualitätslücke im Erlenkotter-Verfahren schließt.

Das Problem bei dieser Lösung ist, dass die Optimalität nur für eine Teilmenge $L \subset K$ der tatsächlich denkbaren Varianten gewährleistet ist. Dieses Problem entsteht durch die begrenzte Enumeration (vgl. Abschnitt 1.3). Daher ist die so gefundene Lösung bezüglich aller denkbaren Varianten $k \in K$ auf Optimalität zu prüfen.

Warmstart falls keine Optimalität vorliegt

Stellt diese Überprüfung fest, dass die Optimalität nicht gegeben ist, so muss dann durch Wahl größerer Werte für κ die Liste der für jeden Wert von p betrachteten Produktionspläne und damit letztlich die Liste der potentiellen Varianten vergrößert werden.

Für einen entsprechenden Warmstart muss nur κ erhöht und die bisherige Struktur der bis zu κ Produktionspläne pro Variantenanzahl $p \in [p_u, p_o]$ beibehalten werden.

Lemma 4.1 *Ein genügend großes Intervall $[p_u, p_o]$ vorausgesetzt (es genügt, das Intervall im Laufe des Verfahrens auf $[1, n]$ zu vergrößern), kann durch wiederholte Wechsel zwischen Heuristiken (mit jeweils vergrößertem κ) und Versuchen, die Optimalität nachzuweisen, die Optimallösung in endlich vielen Schritten gefunden und als solche erkannt werden.*[1]

Beweis: κ wächst in jedem der Iterationsschritte um mindestens 1 (da $\kappa \in \mathbb{N}$). Bei sehr großen κ-Werten erfolgt der Übergang zur vollständigen Enumeration. Offensichtlich wird für $\kappa > 2^m$ beim Kleben (beginnend bei der Bestellliste

[1] Die Rechenzeit kann der praktischen Durchführung im Wege stehen.

B) jede Möglichkeit die Bestellungen in p Varianten zu gruppieren in Betracht gezogen und kein Produktionsplan verworfen.
Nach Lemma 1.1 muss dann die Optimallösung im Verfahren auftauchen; das Verfahren bricht ab.

□

Optimalität bezüglich K nachweisen

Satz 4.2 *Liegt eine Liste potentieller Varianten L (diese enthalte die Einhüllende E_B) und ein zugehöriger Produktionsplan mit geschlossener Dualitätslücke vor, so ist zum Nachweis der Optimalität nur zu zeigen, dass durch Hinzunahme aller Varianten $k \in K$ im Verfahren von Erlenkotter keine negativen Schlupfe s_k entstehen würden. Diese Lösung ist dann optimal bezüglich der Variantenliste K.*

Beweis: Sind für alle Varianten $k \in K$ alle $s_k \geq 0$, so ist die Lösung dual zulässig. Nach Voraussetzung war die Dualitätslücke vor der Hinzunahme aller Varianten $k \in K$ geschlossen. Somit genügt es zu zeigen, dass die Dualitätslücke durch die neuen Varianten $k \in K$ sich nicht öffnet.
Da der Produktionsplan durch die Hinzunahme weiterer Varianten zur Liste potentieller Varianten nicht geändert wird, bleibt die primale Lösung unverändert. Aber auch die v_b bleiben unverändert, da eine Absenkung weder notwendig ($s_k \geq 0$) noch möglich ist (v_b war vorher Optimallösung bezüglich der alten Variantenliste, also durch $s_k = 0$, $k \in L$ bereits beschränkt). Anders ausgedrückt, das Verfahren wäre genauso verlaufen, wenn die „neuen“ Varianten von Anfang an dabei gewesen wären.
Somit bleibt die Dualitätslücke geschlossen und die Lösung ist Optimallösung bezüglich aller Varianten $k \in K$.

□

Nachweis ohne Betrachtung aller s_k, $k \in K$

Ziel dieses Kapitels ist nun zu zeigen, wie ohne **alle** Schlupfe s_k zu berechnen, die Optimalität nachgewiesen werden kann. Dazu wird eine spezielle Enumeration der $k \in K$ vorgeschlagen, die dann ähnlich wie beim Branch & Bound anhand eines Auslotungskriteriums begrenzt wird.

Falls keine Optimalität vorlag, findet das vorgestellte Verfahren eine Variante $k \in K$ mit negativem Schlupf s_k.

Ausnutzung des Nachweises der Suboptimalität

Diese Variante s_k ist in die Liste der potentiellen Varianten aufzunehmen und erneut zu versuchen mit dem Erlenkotter-Verfahren (und ggf. den Heuristiken[2]) eine Optimallösung bezüglich dieser erweiterten Liste potentieller Varianten zu finden.

[2] Die neue Variante kann mit der Heuristik „Varianten hinzufügen“ in bestehende Produktionspläne integriert werden.

Insofern dient das vorgestellte Verfahren nicht nur dem Nachweis der Optimalität, es kann ggf. auch konstruktiv wirken, da es (wenn der Nachweis der Optimalität fehlschlägt) neue potentielle Varianten erzeugt.

Soll die neue Variante k mit negativem Schlupf in das Erlenkotter-Tableau aufgenommen werden, so ist um ein zulässiges Tableau zu erhalten wie folgt vorzugehen:[3]

1. Solange $s_k < 0$ führe aus:
   (a) Wähle ein $b \in B$ mit $k \in K_b^+ := \{k' \in K^+ | v_b > c_{b,k'}\}$ (ein solches existiert, sonst könnte s_k nicht negativ sein).
   (b) Erhöhe s_k um $v_b - \overline{c}_b$ und senke v_b auf den Wert $\overline{c}_b$ ab.[4] Existieren dabei noch weitere $i \in K$ mit $v_b - c_{b,i} > 0$ so sind die entsprechenden s_i ebenfalls um $v_b - \overline{c}_b$ anzuheben.

Übersicht über das Verfahren

Mögliche Vorgehensweisen sind somit:

1. Wähle $\kappa > 0$
2. Wende die Heuristiken an um erste Lösungen und eine Liste potentieller Varianten $L \subset K$ zu erhalten.
3. Berechne mit dem Verfahren von Erlenkotter (ggf. mit Branch & Bound) eine Lösung mit geschlossener Dualitätslücke bezüglich L (bzw. weise nach, dass die Dualitätslücke bezüglich L bereits geschlossen ist).
4. Prüfe, ob die gefundene Lösung Optimallösung bezüglich aller Varianten $k \in K$ ist, d.h. ob $s_k \geq 0$ für alle $k \in K$ gilt. Wenn ja, **stopp**.
5. Füge eine Variante k mit dem negativem Schlupf s_k zu L hinzu.
6. Führe einen oder mehrere der folgenden denkbaren Schritte aus:
   - (a) Erhöhe κ, vergrößere $[p_u, p_o]$ falls möglich.
     (b) Wende die Heuristik „Varianten hinzufügen“ mit der erweiterten Variantenliste L an.
     (c) Wende die „Kleben“-Heuristik an.
     (d) Führe optional noch andere Heuristiken („Kleben“, „Spalten“, „Varianten entfernen“), ggf. auch mehrfach, aus.
   - Füge s_k zu dem Tableau im Erlenkotter-Verfahren (wie oben beschrieben) hinzu.
7. Berechne mit dem Verfahren von Erlenkotter (ggf. mit Branch & Bound) eine Lösung mit geschlossener Dualitätslücke (bzw. weise nach, dass die Dualitätslücke bezüglich L bereits geschlossen ist).
8. Gehe zu 4.

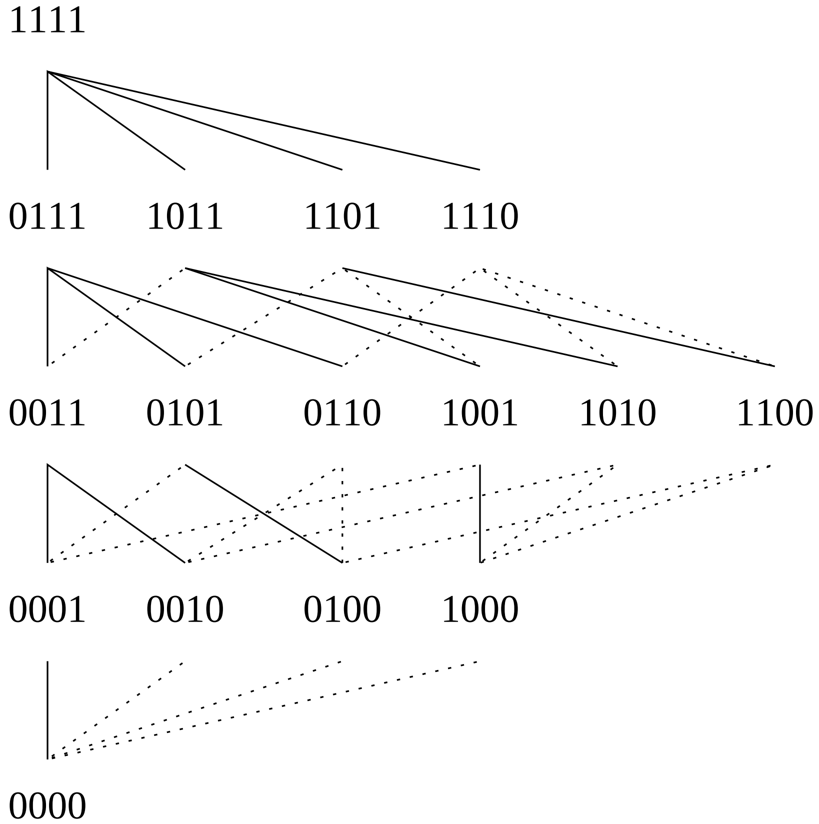
[3] vgl. dazu insbesondere Schritt 1a aus dem Algorithmus „Duale Anpassung“ aus Abschnitt 3.5.3

[4] Zur Definition von $\overline{c}_b$ vergleiche Gleichung (3.19) auf Seite 40.

4.2 Enumeration

Ausgangsposition soll die (nach Voraussetzung in der Liste der potentiellen Varianten vorhandene) Einhüllende E_B sein. Alle möglichen Varianten lassen sich aus dieser durch sukzessives Herausnehmen von Moduln erzeugen. Durch Beobachtung der Schlupfänderung beim Entfernen von Moduln ergeben sich dann Kriterien für die Auslotung.

Da bei der Betrachtung nur die in E_B vorhandenen Moduln eine Rolle spielen (alle anderen sind von keinem Kunden bestellt worden, sind also für das Problem nicht von Interesse), kann E_B o.B.d.A. als E_B = 1111-Variante angesehen werden, d.h. zu Beginn seien alle Moduln in der Variante enthalten. Für $m = 4$ (vier Moduln) ergibt sich dann folgender Graph von zu betrachtenden Modulkombinationen:



Die durchgezogenen Linien sollen kennzeichnen, wie die Enumeration gestaltet werden kann, damit jede Variante bei der Betrachtung der s_k-Werte nur genau einmal auftritt und dennoch neue Varianten nur durch Ersetzen von Einsen durch Nullen entstehen. Im Auslotungsfalle kann das mehrfache Anlaufen einer Variante (gekennzeichnet durch die gestrichelten Linien) nicht immer verhindert werden.

Der Baum wird dadurch gebildet, dass jeder Variante als Nachfolger genau die Varianten zugeordnet werden, die entstehen, wenn eine der Einsen aus einer ununterbrochenen Folge von Einsen am Ende der Modulliste „genullt“ wird.

Beispiel:

$$\begin{aligned} 1100110101000101111 &\Rightarrow 1100110101000101110 \\ &\Rightarrow 1100110101000101101 \\ &\Rightarrow 1100110101000101011 \\ &\Rightarrow 1100110101000100111 \end{aligned}$$

Auf diese Art und Weise kann eine Doppelbetrachtung von s_k-Werten von Varianten vermieden werden. Die gestrichelten Linien geben zusätzliche Auslotungsbeziehungen wieder. Ist sichergestellt, dass durch das Ersetzen von Einsen durch Nullen aus einer oberen Variante kein negativer Schlupf mehr entstehen kann, so gilt dies auch für alle Nachfolger.

Algorithmus

Das im nächsten Abschnitt angegebene Auslotungskriterium vorwegnehmend, kann die Enumeration im Pseudocode wie folgt beschrieben werden (ausgehend von einer Einhüllenden $E = (1, 1, \ldots, 1)$ mit m Moduln):

1. Führe Listen offener Knoten L_μ, $\mu = 0, \ldots, m$. Beginne mit $L_m = \{E\}$, $L_\mu = \emptyset$ für $\mu < m$.

   Führe Listen ausgeloteter Knoten A_μ, $\mu = 0, \ldots, m+1$. Beginne mit $A_\mu = \emptyset$ für alle μ.

2. Starte bei $\mu := m + 1$.

3. Verringere μ um eins. Gilt $\mu = 0$, dann **stopp**.

4. Für alle Varianten $k \in A_{\mu+1}$ entferne alle direkten Nachfolger[5] von k aus L_μ und füge diese Nachfolger zu A_μ hinzu.

5. Für alle $k \in L_\mu$ führe aus:

   (a) Betrachte die Variante k (d.h. füge sie der Variantenliste hinzu, und untersuche sie insbesondere auf Auslotung.)

   (b) Ist k nicht ausgelotet, dann:

      i. Setze $\nu = 1$. Solange $k_\nu = 1$ führe aus:[6]

         A. Füge die Variante k^-, gebildet gemäß

         $$k_i^- = \begin{cases} k_i & \text{falls } i \neq \nu \\ 0 & \text{falls } i = \nu \end{cases}$$

         der Liste $L_{\mu-1}$ hinzu.

         B. Erhöhe ν um 1.

[5] direkte Nachfolger sind hier alle Varianten, die durch Entfernen einer einzigen 1 aus der Variante k entstehen.

[6] k_1 ist dabei das ganz rechts stehende Modul. Das ν-te Modul entspricht also dem Wert 2^ν in der Binärdarstellung der Modulkombination.

(c) Ist k ausgelotet, füge k zu A_μ hinzu.

(d) Entferne k aus L_μ.

6. Gehe zu 3.

Bemerkung: Im Algorithmus müssen teilweise die Auslotungsinformationen für sehr viele Varianten gleichzeitig verfügbar sein. Allein $A_\mu \cup L_\mu$ hat $\binom{m}{\mu}$ Elemente.

Satz 4.3 *Der obige Algorithmus führt zur Bearbeitung aller möglichen Varianten (sofern keine Auslotung festgestellt wurde). Jede Variante wird dabei nur einmal betrachtet. Die Betrachtung geschieht so, dass eine Nachfolgevariante immer nur durch Entfernung eines Moduls aus der aktuell betrachteten Variante entsteht.*

Beweis: Per Induktion über die Anzahl der Einsen μ der jeweiligen Variante k.

Induktionsstart $\mu = m$: Der Algorithmus beginnt mit $L_m = E$, der einzigen Variante mit m Einsen. Alle Varianten aus L_m werden bearbeitet, denn μ ist zunächst $m + 1$ und wird dann vor der Schleife um 1 erniedrigt. Auslotung kann nicht bestehen, denn $A_{m+1} = A_m = \emptyset$.

Induktionsschritt Angenommen alle Varianten mit $(\mu + 1)$ Einsen werden, sofern keine Auslotung besteht, in der Schleife bearbeitet. Es ist zu zeigen, dass dann alle nicht-ausgeloteten Varianten mit μ-Einsen bearbeitet werden.

Angenommen, es existiert eine Variante k mit μ Einsen, die nicht ausgelotet ist, und die dennoch nicht bearbeitet wird. Da $\mu < m$ besteht k nicht ausschließlich aus Einsen. Also existiert ein minimals ν mit $k_\nu = 0$. Betrachte die Variante k^+ gegeben durch

$$k_i^+ = \begin{cases} k_i & \text{falls } i \neq \nu \\ 1 & \text{falls } i = \nu \end{cases}$$

k^+ hat dann $\mu + 1$-Einsen. Nach Induktionsvoraussetzung wurde aber k^+ in der Schleife bearbeitet. Da ν minmal gewählt wurde mit $k_\nu = 0$ ist dann aber k aus k^+ in der Schleife erzeugt und in L_μ eingefügt worden. Wenn aber k in L eingefügt wurde, so wurde k entweder bearbeitet oder aber ist aufgrund von Auslotung durch ein anderes $k^* \in A_{\mu+1}$ entfernt worden. Es besteht somit ein Widerspruch zur Annahme, k wäre nicht ausgelotet und nicht bearbeitet worden.

Dass jede Variante höchstens einmal bearbeitet wird, ergibt sich aufgrund der Eindeutigkeit mit der jeder Variante genau ein Vorgänger k_i^+ zugeordnet werden kann (Konstruktion wie im obigen Induktionsschritt).

$\square$

4.3 Änderung der UC-Matrix

Beim Entfernen eines Moduls μ aus einer Variante k sind drei Fälle für die Änderung jedes Eintrags $c_{b,k}$ in der UC-Matrix zu unterscheiden:

$c_{b,k}$ wird verkleinert

Deckt die neu enstandene Variante k' die Bestellung b immer noch ab, so sinkt $c_{b,k}$ um $l_b \cdot K_\mu$, denn die Kosten von k sind um K_μ gesunken.

$c_{b,k}$ wird ∞

Deckt die neue Variante k' die Bestellung b nicht mehr ab, so ist $c_{b,k'} = \infty$.

$c_{b,k}$ bleibt unverändert

Deckte bereits die Variante k die Bestellung b nicht ab, so war $c_{b,k}$ bereits unendlich.

Änderung von s_k

Jede Änderung der $c_{b,k}$ beeinflusst den Wert von s_k, denn es gilt:

$$s_k = a_k - \sum_{b \in B} \max\{0, v_b - c_{b,k}\} \tag{4.1}$$

Wird somit $c_{b,k}$ um $\triangle c_{b,k}$ kleiner und gilt danach $v_b > c_{b,k} - \triangle c_{b,k}$, so wird s_k auch kleiner, und zwar um

$$\min\{\triangle c_{b,k}, v_b - (c_{b,k} - \triangle c_{b,k})\}.$$

Also wird s_k um höchstens

$$\max\{0, \min\{\triangle c_{b,k}, v_b - (c_{b,k} - \triangle c_{b,k})\}\}$$

verkleinert.

Wird $c_{b,k}$ auf ∞ gesetzt, so wird s_k größer (wenn $v_b > c_{b,k}$ war).

4.4 Auslotung

Satz 4.4 *Auslotung wird erreicht, wenn*

$$\sum_{b \in B} \max\{0, \min\{c_{b,k}, v_b\}\} \leq s_k. \tag{4.2}$$

Beweis: Auslotung wird nach Satz 4.2 erreicht, wenn s_k für alle Nachfolger von k in der Enumeration nicht mehr negativ werden kann.
Aus Formel (4.1) wurde bereits gefolgert, dass s_k um

$$\min\{\triangle c_{b,k}, v_b - (c_{b,k} - \triangle c_{b,k})\}$$

kleiner wird, falls $v_b > c_{b,k} - \triangle c_{b,k}$.

Da $c_{b,k} \geq 0$ kann $c_{b,k}$ maximal um $c_{b,k}$ verkleinert werden, also $\triangle c_{b,k} \leq c_{b,k}$. Für ein bestimmtes $b \in B$ wird s_k somit um maximal

$$\begin{aligned} \triangle_{b,k} &= \min\{\underbrace{\triangle c_{b,k}}_{\leq c_{b,k}}, v_b - \underbrace{(c_{b,k} - \triangle c_{b,k})}_{\geq 0}\} \\ &\leq \min\{c_{b,k}, v_b\} \end{aligned}$$

abgesenkt.
Durch Summation der maximalen Absenkungspotentiale aller $b \in B$ für das untersuchte s_k folgt die postulierte Formel (4.2).

□

Wann Auslotung konkret erreicht wird, hängt stark von dem zu betrachteten Beispiel ab. Zumindest theoretisch kann so die Optimalität aber in jedem Fall bewiesen werden.

In der Praxis wird es von der Problemgröße (Anzahl der Bestellungen bzw. Moduln) abhängen, ob die Anzahl der notwendigen Schritte durch dieses Kriterium klein genug wird, um eine Berechnung in sinnvollen Zeiträumen zu ermöglichen.

Kapitel 5

Numerische Ergebnisse

In diesem Kapitel werden die mit dem im Anhang beschriebenen Programm erzielten numerischen Ergebnisse vorgestellt. Das Programm implementiert die im zweiten Kapitel vorgestellten Heuristiken unter Berücksichtigung zusätzlicher Restriktionen. Dabei sind zwei Typen von Restiktionen zu unterscheiden:

1. Ein Modul kann bei der Konstruktion zulässiger Varianten nicht frei gewählt werden. Ob das Modul in die Modulkombination eingebaut werden muss, hängt vollkommen von den anderen Moduln ab. Eine denkbare Beziehung wäre z.B. das Modul 4 genau dann vorkommt, wenn Modul 3 nicht vorkommt.

2. Ein Modul kann nur dann vorkommen, wenn gewisse andere Moduln auftreten bzw. nicht auftreten. Ein einfaches Beispiel für diesen Fall wäre, wenn Modul 1 nicht gleichzeitig mit Modul 2 auftreten darf.

Diese Abhängigkeiten berücksichtigt das Programm in allen vorgestellten Heuristiken durch Einschränkung der Kombinationsmöglichkeiten bzw. Berechnung der nicht frei wählbaren Moduln.

Das Verfahren von Erlenkotter und die in diesem Zusammenhang im vierten Kapitel vorgeschlagene Strategie zum Nachweis der Optimalität wurden nicht implementiert. Das Programm läuft interaktiv ab und ermöglicht dadurch eine benutzergesteuerte Reihenfolge bei der Anwendung der Heuristiken. Die Parameter κ und g können frei gewählt werden.

5.1 Die Beispiele

Betrachtet wurden im wesentlichen zwei Beispiele, die sich hauptsächlich in der Anzahl der verschiedenen Bestellungen unterscheiden.

5.1.1 Beispiel A

Es wurden 209 Bestellungen und 24 Moduln betrachtet. Dabei sind nur 78 Bestellungen verschieden, die meisten Bestellungen traten somit mehrfach (mit möglicherweise verschiedenen Stückzahlen) auf.

Von den 24 Moduln konnten nur 11 vom Kunden relativ frei gewählt werden. Alle anderen ergaben sich dann automatisch aufgrund von Abhängigkeiten zwischen den Moduln. Für im Programm neu gebildete Modulkombinationen wurden die 13 als „technische“ Moduln bezeichneten nicht frei wählbaren Moduln immer neu berechnet.[1]

Zwischen den „frei“ wählbaren Moduln bestanden folgende Abhängigkeiten, die die Wahlfreiheit einschränken:

- Die Moduln 2 und 3, 3 und 8 sowie 3 und 9 durften nicht gemeinsam auftreten.
- Von den Moduln 8, 9 und 10 durfte nur eines vorhanden sein.
- Modul 11 durfte im Produkt nicht vorkommen, wenn es nicht explizit bestellt wurde.

Aufgrund der Abhängigkeiten entstanden 6 Einhüllende (d.h. alle Bestellungen könnten mit nur 6 verschiedenen Varianten abgedeckt werden).

Die Moduln hatten eine Preisspanne von 0 bis 332290 Geldeinheiten (GE). Die Bestellvolumina lagen zwischen 1 und 1929 Stück. Das Gesamtvolumen des Auftrags belief sich auf ca. 16.300.000.000 GE.

5.1.2 Beispiel B

Hier wurden 844 verschiedene (!) Bestellungen von Kombinationen aus 44 Moduln betrachtet. Verschieden bedeutet dabei, das keine Kombination zweimal bei den Bestellungen auftrat.

Nur die ersten 20 Moduln waren „frei“ wählbar, die anderen 24 Moduln ergaben sich dann in eindeutiger Weise aus diesen 20 Moduln. Zwischen den 20 „freien“ Moduln bestanden folgende Abhängigkeiten:

- Modul 18 durfte nicht gemeinsam mit einem der Moduln 1, 5, 7, 9, 13, 15, 16 oder 19 auftreten.
- Die Moduln 13 und 17 dürfen ebenfalls nicht gemeinsam auftreten.
- Das Modul 14 durfte nicht vorkommen, wenn es nicht explizit bestellt wurde.

Die Preise für die Moduln lagen zwischen 100 und 28.000 GE. Die Bestellvolumina zwischen 1 und 32163 Stück. Das Gesamtvolumen betrug ca. 17.300.000.000 GE.

5.2 Rechenzeiten

Beide Beispiele wurden — trotz der großen Zahlen — exakt, also ohne jegliche Rundung, gerechnet.

Für die Diskussion der Heuristiken wurde eine spezielle Reihenfolge für die Heuristiken ausgewählt. Arbeiten alle Heuristiken (mit Ausnahme der Schnellklebeheuristik am Anfang) immer im Intervall $[10, 50]$, so ergeben sich für die beiden Beispiele für diese Reihenfolge folgende Laufzeiten (in Minuten):[2]

[1] Die 13 Moduln entsprechen somit den Verpackungen des Pralinenfabrikanten vom Vorwort.

[2] Die Zeiten für I/O wurden mitgerechnet. Es wurde nicht die real benötigte CPU-Zeit gemessen, sondern nur die Differenz der Uhrzeiten der Ergebnisdateien verwendet. Alle Zeitangaben geben somit nur die Größenordnung wieder!

Heuristik	Beispiel A ($\kappa = 10$)	Beispiel B ($\kappa = 5$)
Schnellkleben ($g = 30$)	0	2
Varianten entfernen	14	37
Varianten hinzufügen	32	919
Varianten entfernen	3	15
Varianten hinzufügen	4	83
Kleben & Spalten	4	*4500
Varianten entfernen	4	*67
Varianten hinzufügen	36	*1170
Kleben & Spalten	4	*4615
Varianten entfernen	5	*63
Varianten hinzufügen	35	*1167
Kleben & Spalten	6	*2347

Bis auf die mit $*$ gekennzeichneten Werte gelten die Laufzeiten für einen PII-350 mit 256 MB RAM. Die $*$-Werte gelten für eine UltraSPARC II-300 mit 4 MB Cache und 1 GB physikalischem Speicher.

Auf dem PII wurden ebenfalls Laufzeiten für verschiedene κ-Werte gemessen. Für $\kappa = 1$ wurden für Beispiel A (gleiche Reihenfolge und sonstige Parameter für die Heuristiken) 12 Minuten, für $\kappa = 5$ bereits 65 Minuten benötigt. Der Aufwand wächst, wie erwartet, ungefähr linear mit κ.

5.3 Entwicklung der Optimallösungen

Das Programm wurde so modifiziert, dass beim Auftreten einer neuen κ-besten Lösung[3] für ein p dies optisch dargestellt wurde. Das Auftreten einer neuen Optimallösung (für ein bestimmtes p) wurde ebenfalls gesondert ausgewiesen.

Dadurch konnte beobachtet werden, dass ausnahmslos alle Heuristiken immer mal wieder einen Beitrag zur Verbesserung der Lösung liefern. Auch wurde empirisch beobachtet, dass sich die κ-gepufferte Vorgehensweise als vorteilhaft erwies, denn aus $\kappa' \leq \kappa$-besten Lösungen entstanden für $p \pm 1$ in einigen Fällen neue beste Lösungen. Die Häufigkeit, mit der eine neue beste Lösung gefunden wird, nimmt jedoch mit wachsendem κ' extrem schnell ab.

Die Häufigkeit mit der bessere Lösungen von den einzelnen Heuristiken gefunden werden, hängt stark davon ab, welche anderen Heuristiken vorher abgelaufen sind, wie lange schon optimiert wurde und wie die Parameter gesetzt wurden. Die Idee, variantenerzeugende und variantenauswählende Verfahren abwechselnd einzusetzen, führte dabei zu guten Trefferquoten.

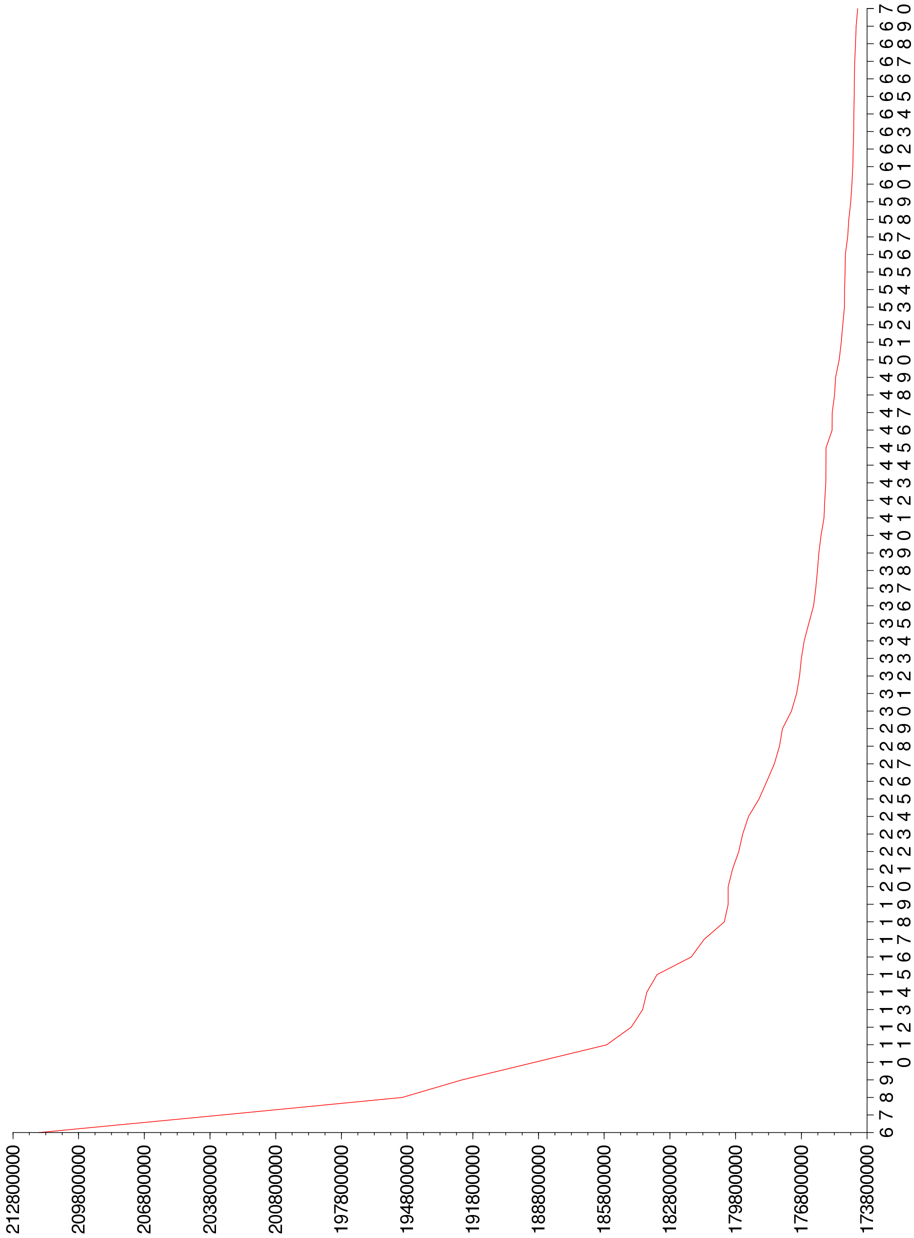
Wurden nur variantenerzeugende oder nur variantenauswählende Heuristiken eingesetzt, führte dies schnell zu einem Ende der Entwicklung, d.h. wiederholte Anwendung führte zu geringen oder gar keinen Verbesserungen mehr. Wurde hingegen zum dem anderen Typ von Heuristiken gewechselt, konnten — zumindest am Anfang — noch deutliche bessere Lösungen gefunden werden.

[3]Eine κ-beste Lösung ist eine Lösung die unter die κ besten bekannten Produktionspläne für eine gegebene Anzahl an Varianten fällt und die somit bei der begrenzten Enumeration weiterverwendet wird.

5.3.1 Abhängigkeit der Lösung von p

Die gefundenen Optimallösungen verhalten sich, in Abhängigkeit von p wie erwartet, d.h. bei steigendem p gibt es zunächst einen steilen Abfall, bei großen Werten von p verflacht dieser dann aber sehr schnell.

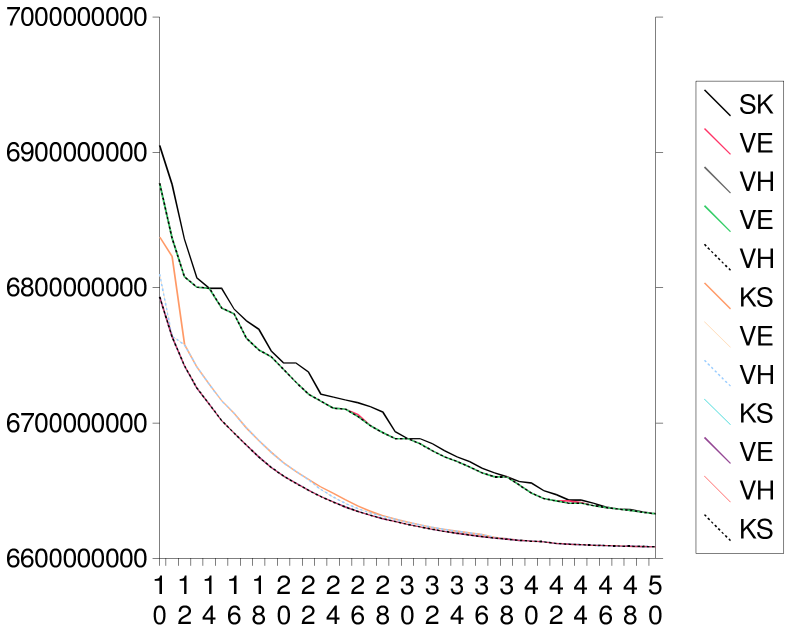
Für das Beispiel B sieht dies wie folgt aus:



5.3.2 Testergebnisse für die beschriebene Reihenfolge

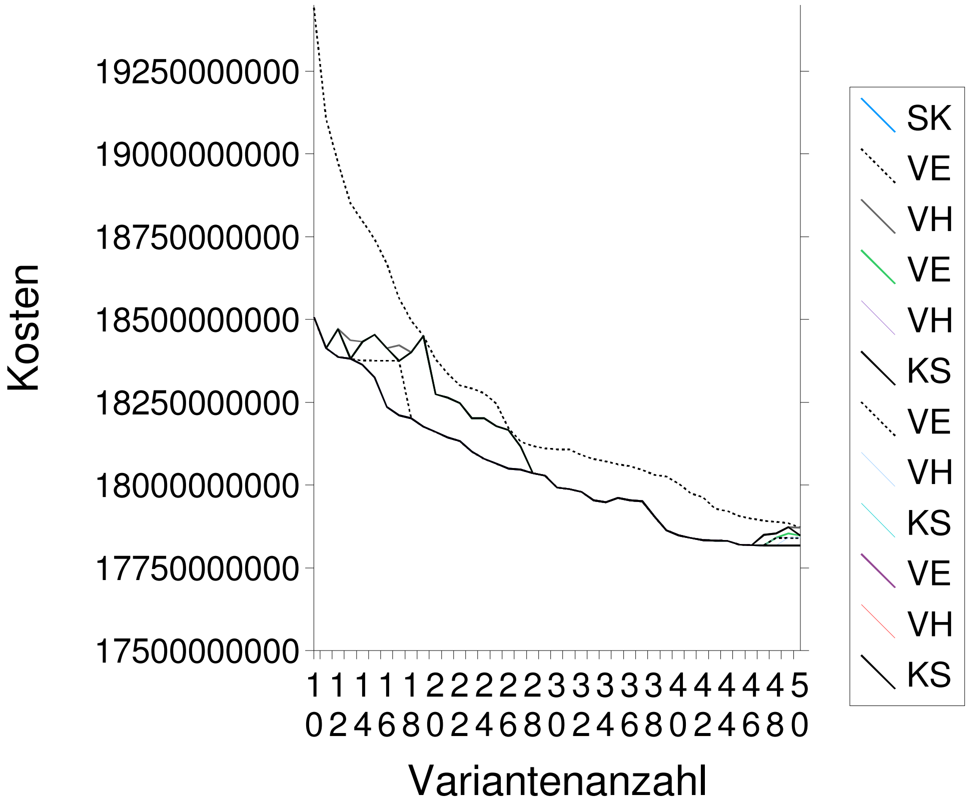
Für Testzwecke wurde die oben bereits angegebene Reihenfolge verwendet. Dabei wird die „Kleben-Spalten-Heuristik“ immer so lange angewendet, bis sich durch Kleben und Spalten nichts mehr ändert. Dies ist in der Regel nach ein oder zwei Aufwärts-Abwärts-Iterationen der Fall.

Für das Beispiel A mit $k = 5$ sieht die Entwicklung der Optimallösungen im Intervall $[10, 50]$ wie folgt aus:



Die Kurve und die zugehörigen Tabelle befinden sich auf der beiliegenden CD-ROM im Verzeichnis `documentation/rechnungen/`. *Unter* `documentation/` *befindet sich auch die obige Grafik in Farbe. Die Ausgangsdaten sind unter* `data/` *zu finden.*

Für das Beispiel B mit $k = 5$ sieht die Entwicklung der Optimallösungen im Intervall $[10, 50]$ wie folgt aus:

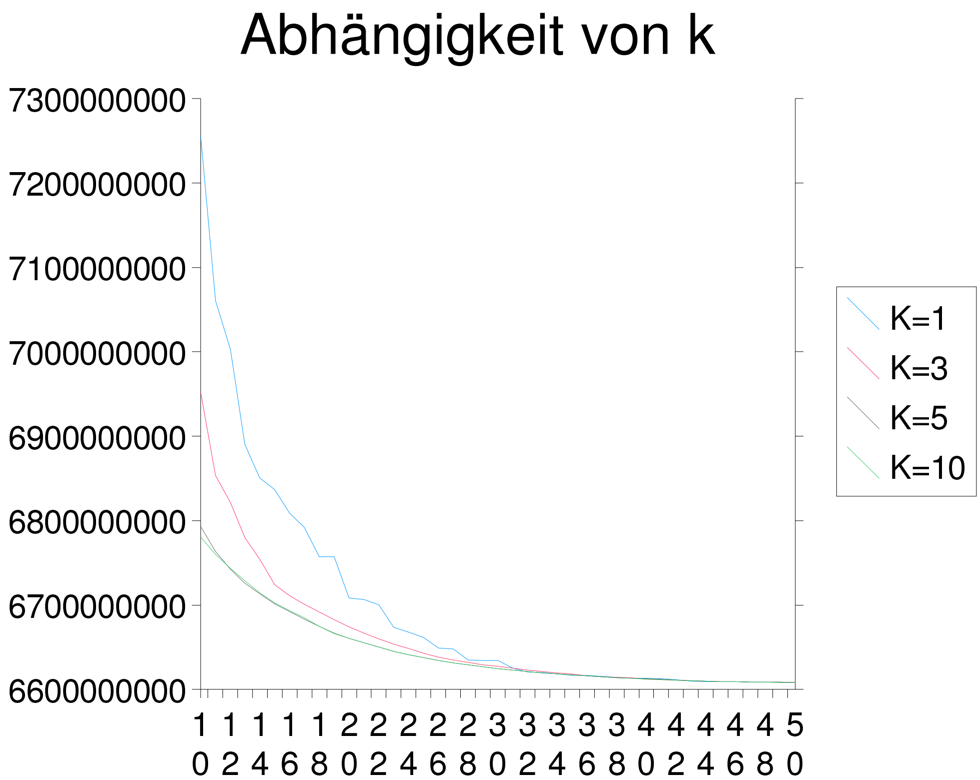


Die Kurve und die zugehörigen Tabelle befinden sich auf der beiliegenden CD-ROM im Verzeichnis `documentation/rechnungen/`. *Unter* `documentation/` *befindet sich auch die obige Grafik in Farbe. Die Ausgangsdaten sind unter* `data/` *zu finden.*

Aus den Kurven wird ersichtlich, dass, um Fortschrite zu erzielen, es sinnvoll ist, zwischen den verschiedenen Heuristikentypen zu wechseln.

5.3.3 Abhängigkeit der Lösung von k

Das Beispiel A wurde für verschiedene k-Werte (mit der oben angegebenen Test-Reihenfolge) untersucht. Die Endergebnisse für $k = 1, 3, 5$ und 10 sehen im Vergleich wie folgt aus:



5.4 Grenzen des Verfahrens

Die Laufzeit aller Heuristiken hängt von der Anzahl der Moduln nur linear ab, da alle Arbeitsschritte (wie Preis der Modulkombination bestimmen, Modulkombinationen „verkleben", usw.) linear zur Anzahl der Moduln sind. Daher sind zusätzliche Moduln für die Laufzeit des Programms praktisch nicht von Bedeutung.

Bei der Schnellklebe-Heuristik geht die Anzahl der Bestellungen n (wenn man vom Sortiervorgang absieht) ebenfalls nur linear ein, da ausgehend von der Bestellliste B nach maximal n Klebeschritten (mit Aufwand $O(m \cdot g^2)$) die Heuristik beendet ist.

Damit steht ein Werkzeug zur Verfügung, um Startlösungen in einem vorgegebenem Intervall zu erhalten. Anhand der durch Schnellkleben erlangten Startlösungen kann auch ein sinnvolles Intervall $[p_u, p_o]$ (ausgehend von den a_k-Werten) festgelegt werden.[4] Auf dieses Intervall begrenzt (und nicht mehr auf $[1, n]$) ist es dann möglich, mit den anderen Heuristiken ebenfalls relativ schnell zu Ergebnissen zu kommen.

Das einzige Problem, dass auftritt, ist die Geschwindigkeit mit der eine ausreichend gute Optimallösung gefunden wird. Bei vielen Moduln und vielen Bestellungen kann die Anzahl der notwendigen Verbesserungsschritte groß werden. Auch muss ggf. κ größer gewählt werden, damit die für weitere Verbesserungen notwendigen Varianten einbezogen werden. Abgesehen davon kann das Programm im Prinzip mit jeder Problemgröße umgehen (d.h. theoretisch auch mit hunderten von Moduln und zehntausenden von Bestellungen). Zu beachten ist dann allerdings der Speicherbedarf.

Abhängigkeiten zwischen den Moduln können, ohne einen relevanten Einfluss auf die Laufzeiten zu haben, aufgrund der gewählten Struktur ebenfalls fast beliebig komplex ausfallen. Zu den Möglichkeiten der Definition von Abhängigkeiten vergleiche die Dokumentation im Anhang.

Durch die Verwendung der `gmp`-Bibliothek ist die verwendete Arithmetik im Prinzip unbegrenzt, d.h. es kann mit beliebig großen Zahlen exakt gerechnet werden. Zur Zeit bestehen allerdings noch bei der Ein- und Ausgabe Beschränkungen auf 100 Zeichen pro Zahl.

[4] Vgl. z.B. die symmetrische Wahl des Intervalls in den Methoden `create-final-set-max-ap` und `create-final-set-max-uc` (im Abschnitt A.6.6 beschrieben).

Anhang A

Dokumentation zum Programm

In diesem Kapitel wird das erstellte Programm vorgestellt. Dabei wird zuerst ein Überblick über den Ablauf gegeben und dann tiefer in die Details gegangen. Am Ende werden weitere numerischen Ergebnisse (unter Berücksichtigung des vom „Pralinenfabrikanten“ gewünschten Datenschutzes) kurz vorgestellt.

A.1 Einführung

A.1.1 Systemvoraussetzungen

Das Programm wurde unter verschiedenen Linux-Installationen, so wie unter Solaris (V5.6 und V7) getestet. Aber auch andere UNIX-Distributionen sollten kein generelles Problem darstellen.[1]

Folgende Hilfsprogramme werden vom Programm benötigt:[2]

- (gnu-)make, (gnu) C++-Compiler und Linker
- C++-Standard Template Library (g++-2.95)
- gnu mp Library (2.0.2)
- sh, tr, gnu-tail, gnu-head, egrep, gnu-sed (!)
- nur für die Dokumentation: TeX, LaTeX2e, amsTeX, dvips (tetex)

Auf `Pentium II`-Klasse-Systemen[3] beträgt die Rechenzeit je nach Problemgröße Minuten bis Tage. Die Dauer hängt auch von den gewählten Parametern (welche Algorithmen, welcher κ-Wert, wieviele Iterationen) ab. Exemplarische Rechenzeiten für verschiedene Problemgrößen werden in Abschnitt A.9.3 angegeben.

Der Server sollte über 92 MB Arbeitsspeicher verfügen, wobei das Programm je nach Problemgröße durchaus deutlich mehr Speicher benötigt. Hier genügt dann jedoch virtueller Speicher, da das Programm sehr stark lokal arbeitet.

[1] Unter Windows NT müsste diverse Zusatzsoftware installiert werden. Dem „Pralinenfabrikanten“ ist eine Portierung gelungen.

[2] in Klammern: empfohlene Variante

[3] Im Prinzip gehen alle Prozessoren, auch nicht-Intel-kompatible, wenn ein entsprechender C++-Compiler verfügbar ist.

A.1.2 Systematik der Dateien und Verzeichnisse

Das Programm sollte als eine Datei mit dem Namen *variantenoptimierer.tar* auf Diskette vorliegen. Die Datei wird durch Eingabe von

```
tar xvf variantenoptimierer.tar
```

entpackt.[4]

Dabei wird ein Verzeichnis `variantenoptimierer` mit den Programmdaten angelegt. Diese sind wie folgt gegliedert:

data/ Daten, d.h. Eingangsdaten, Zwischenergebnisse und Endergebnisse werden hier abgelegt bzw. von hier eingelesen.

bin/ Hier werden die Programme nach der Compilation abgelegt. Es ist nach dem Entpacken zunächst leer.

documentation/ (diese) Dokumentation im TeX-Quellcode

src/ Quellcodes des Hauptprogramms

src/filters/ Quellcodes der Eingabe- und Ausgabefilter

src/tools/ lange Integer-Arithmetik für die Shell-Skripte

AUTHORS Liste der Autoren

CHANGES Liste der Änderungen

COPYING Copyright (GNU GPL)

INSTALL Kurzanleitung zur Installation (Englisch)

README Kurzanleitung (Deutsch)

Makefile Makefile zum Übersetzen und Aufrufen des Optimierers

Makefile.cfg globale Konfigurationsdatei

A.1.3 Compilieren und Aufrufen des Optimierers

Zunächst müssen in der Datei *Makefile.cfg* Optionen für den Compiler eingestellt werden. Für die meisten Plattformen sollten die Voreinstellungen zwar funktionieren, sie werden aber nicht zwingend den schnellsten Code produzieren[5]

Durch Eingabe von `make KOMMANDO` können dann verschiedene Vorgänge gestartet werden. Hier die wichtigsten:

make install Compiliert das Programm (und die Filter) und kopiert die Binärdateien nach `bin/`.

make daten Übersetzt die Eingangsdaten in ein für das Hauptprogramm lesbares Format.

[4] Liegt ein `.tgz`- oder `.tar.gz`-Archiv vor, ist dieses vorher mit `gunzip variantenoptimierer.tgz` zu dekomprimieren.

[5] Z.B. sollte bei Intel-Pentium-II-Prozessoren mit `--DCPU=686` für ein Pentium-II optimierter Code (statt 80386-Code) erzeugt werden.

make Ruft das Hauptprogramm mit den Eingangsdaten auf.

make docs Erzeugt die Dokumentation (PostScript-Datei)

make clean Löscht alle erzeugten Daten, d.h. die Binär- und Objektdateien, die Programmausgaben und alle Zwischenergebnisse.

Ansonsten ist noch zu bemerken, dass `make` so intelligent ist zu merken, wenn z.B. für `make` (ohne Paramter) zuerst das Hauptprogramm übersetzt werden muss, und dies dann automatisch ausführt. Weiterhin werden (bis auf die Optimierung und die Ausgabe in HTML) immer nur die unbedingt notwendigen Aktionen ausgeführt, d.h. wurden die Eingangsdaten einmal umgewandelt, wird dieser Schritt immer übersprungen, es sei denn, die Daten wurden geändert.

An dieser Stelle sei direkt noch darauf hingewiesen, dass das Hauptprogramm jedesmal an die jeweiligen Daten angepasst wird, d.h. neue Daten können auch dazu führen, dass das Programm neu übersetzt werden muss.

A.1.4 Übersicht über den Datenfluss

Eingangsdaten

Nach der Installation sind zunächst im Unterverzeichnis *data/* sieben Dateien mit Eingangsdaten anzulegen. Diese enthalten die Modulnamen und -preise, die Modulausschlüsse, die Bestellliste sowie Informationen zu speziellen Abhängigkeiten. Die siebte Datei enthält in der ersten Zeile nur den Index des „COMMUN“-Moduls, d.h. des Moduls, das in allen Bestellungen vorkommt. Ein solches Modul muss existieren, ggf. mit Kosten 0.

Eingangsfilter

Durch Eingangsfilter werden die ersten drei Dateien vorbehandelt. Dabei wird zunächst umformatiert. Unnötige Informationen werden entfernt und das Format für C++-Programme einfacher lesbar gemacht.

Danach werden die Eingangsdaten analysiert und das Problem um die durch K in der Abhängigkeitsmatrix gekennzeichneten Fälle reduziert. Auch das „COMMUN“-Modul wird entfernt.

Abschließend werden die Eingangsdateien zu einer zusammengefasst ausgegeben. Mit der enstehenden Datei *data/combined-list* startet dann das Hauptprogramm.

Hauptprogramm

Bereits bei der Übersetzung des Hauptprogramms werden aus den Eingangdateien *f?-implicit.txt* („?“ $= 4 - 6$) Include-Dateien (*.cppi*) erzeugt, die direkt in das Programm eincompiliert werden. Dies ist aus Geschwindigkeitsgründen vorteilhaft.

Einmal gestartet analysiert das Hauptprogramm die Daten und wendet verschiedene Heuristiken an. Dazu wurde ein einfacher Dialog mit dem Anwender programmiert.

Um die Ergebnisse in einem lesbaren Format zu erhalten, muss das gewünschte (Zwischen-)Ergebnis in die Datei *data/results* ausgegeben werden.

HTMLisierung

make result erzeugt anschließend aus *data/results* lesbare Ergebnisse. Dazu wird zunächst ein Ausgabefilter aufgerufen. Dieser analysiert die Datei und erzeugt pro Variantenanzahl drei temporäre Dateien. Zwei Hilfsprogramme wandeln diese in HTML um, wobei die vom Eingangsfilter entfernten K-Moduln wieder hinzugefügt werden.

Indizierung

Abschließend erzeugt ein Programm die Datei *data/index.html* in der alle Ergebnisse in Kurzform aufgelistet und durch Links mit der kompletten Kombinationsliste verbunden werden.

data/index.html kann danach mit einem Browser betrachtet werden.

A.2 Datenfluss

A.2.1 Format der Ausgangsdaten

data/f1-modulnamen.txt

In dieser Datei werden die Modulnamen und Modulpreise definiert.

Berücksichtigt werden nur die 1. und die 3. Zeile der Datei. Die erste Zeile muss mit drei Semikolons (;;;) beginnen. Danach kommen die Modulnamen, eingefasst in Anführungszeichen (") und durch jeweils genau ein Semikolon voneinander getrennt. Am Ende der Zeile steht ebenfalls ein Semikolon.

Die dritte Zeile beginnt mit *;;"Price of option (FF)";*. Danach folgen die Modulpreise (gleiche Reihenfolge wie bei den Moduln!), wieder durch Semikolons getrennt, aber ohne Anführungszeichen.

Dieses „kuriose" Format entstand historisch durch die aus einer Excel-Tabelle übernommenen Orginaldaten. Unter StarOffice wurde dazu einfach die Bestelliste als *Text - txt - csv (StarCalc)* gespeichert, wobei Semikolons bzw. Anführungszeichen als Trenner benutzt wurden.

data/f2-abhaengigkeiten.txt

Diese Datei enthält die Abhängigkeitsmatrix. Im folgenden wird beschrieben, in welcher Form die Matrix in der Datei anzugeben ist und was die einzelnen Einträge aussagen.

Bei dieser Datei werden die ersten drei Zeilen ignoriert. Die restlichen Zeilen werden gezählt (die Anzahl **muss** gleich der Modulanzahl n sein) und dann von jeder Zeile die letzten n Einträge zwischen den Semikolons betrachtet. Dabei werden Anführungszeichen sowie „B"'s ignoriert. Die verbleibende Matrix sollte „O"'s auf der Diagonalen und ansonsten nur „O"'s, „K"'s und „E"'s enthalten.

Dabei steht ein „O" dafür, dass die zu der entsprechenden Spalte bzw. Zeile gehörenden Moduln nicht kombiniert werden dürfen. „O"s sollten daher zur Diagonalen symmetrisch auftreten.[6]

„E"'s bedeuten, dass das Zeilenmodul das Modul in der Spalte beinhaltet, d.h. das Modul aus der Zeile hat alle Funktionen des Moduls aus der Spalte. Das

[6]Treten die „O"s nicht symmetrisch auf, wird vom Programm automatisch ergänzt.

Zeilenmodul sollte teurer sein als das Spaltenmodul. „E“s sollten asymmetrisch auftreten.

„K“'s bedeuten, dass das Spaltenmodul das Zeilenmodul beinhaltet **und** das das Zeilenmodul immer genau dann vorhanden sein muss, wenn das Spaltenmodul nicht eingebaut wird. Dies bedeutet insbesondere, dass die Funktionen des Zeilenmoduls **immer**, d.h. wirklich in jeder Variante vorhanden sein müssen.[7] Das Spaltenmodul **muss** teurer sein, als das Zeilenmodul.

Auch dieses Format entstand historisch durch die aus einer Excel-Tabelle übernommenen Orginaldaten. Unter StarOffice wurde dazu die Abhängigkeitsmatrix als *Text - txt - csv (StarCalc)* gespeichert, wobei wieder Semikolons bzw. Anführungszeichen als Trenner benutzt wurden.

data/f3-bestellungen.txt

Diese Datei ist — wenn aus den Orginaldaten erzeugt — exakt die gleiche wie *f1-modulnamen.txt*. Nur diesmal werden vom Programm nicht die ersten drei Zeilen sondern der Rest interpretiert. Fehlt daher *f3-bestellungen.txt* wird *f1-modulnamen.txt* von *make* nach *f3-bestellungen.txt* kopiert.

f3-bestellungen.txt sollte von der vierten bis zur vorletzten Zeile jeweils Informationen über eine Bestellung enthalten. Der Inhalt jeder der Zeilen vor dem ersten Semikolon wird ignoriert. Zwischen dem ersten und dem zweiten Semikolon muss die Bestellanzahl für die im Rest der Zeile spezifizierte Bestellung angegeben sein. Zwischen dem zweiten und dritten Semikolon steht der Name der Bestellung. Auch dieser wird vom Programm ignoriert.

Danach folgt, wie immer durch Semikolons getrennt, die Liste der bestellten Moduln. 0 steht für nicht bestellt, 1 für bestellt. Es wird nicht überprüft ob beim COMMUN-Modul immer eine 1 steht. Der letzte Eintrag in der Zeile (nach dem letzten Semikolon) wird ignoriert. Hier steht üblicherweise der Preis der entsprechenden Bestellkombination.

Die letzte Zeile der Datei wird ignoriert.

f4-implicit.txt

In dieser Datei werden Regeln angegeben, wie sich Moduln aus anderen rechnerisch zwingend ergeben. Diese Moduln werden auch als technische Moduln bezeichnet.[8] Im Beispiel unseres Pralinenfabrikanten aus dem Vorwort wären dies also die Verpackungsparameter.

Die Datei könnte z.B. wie folgt aussehen:

```
[12]=[4]&&![8]
[13]=[8]||[11]
[14]=[8]&&[11]
[15]=![11]&&([8]||[7])
[16]=[11]&&([8]||[7])
```

Das bedeutet, dass die Moduln 12 bis 15 sich nach bestimmten Regeln aus den vorangegangnenen Moduln ergeben. Die Syntax ist dabei nicht etwa nur an C++ angelehnt, sondern es handelt sich unmittelbar um C++-Code. Die

[7] Im Algorithmus werden daher für die Berechnung diese Moduln dem **COMMUN**-Modul zugeschlagen.

[8] Im Gegensatz zu Funktionsmoduln die vom Kunden „frei“ wählbar sind.

eckigen Klammern werden lediglich auf die entsprechenden *bool*-Werte für die Moduln gesetzt. Dadurch sind beliebig komplizierte Anweisungen möglich, z.B. bedeutet

```
[12]=([11]+[1]+[3]-[4])>1,
```

dass Modul 12 vorkommt, wenn von 1, 3 oder 11 mindestens zwei vorhanden sind, oder 3 wenn auch 4 vorhanden ist.

Zu beachten ist noch, dass die Zählung der Moduln bei 0 beginnt, dass das COMMUN-Modul NICHT mitgezählt werden darf, und auch die K-Moduln (vgl. *f2-abhaengigkeiten.txt*) hier nicht mitgezählt werden.

f5-implicit.txt

In dieser Datei können zusätzliche Ausschlüsse formuliert werden. Die Funktionalität entspricht somit im Prinzip der Angabe von „O“ in *f2-abhaengigkeiten.txt*, *f5-implicit.txt* erlaubt aber mehr.

Der durch diese Datei erzeugte Code wird aufgerufen mit der Aufforderung zu prüfen, ob die durch {} angesprochene erste Kombination mit der durch [] erreichbaren zweiten vereinbar ist. Die Beziehung ist natürlich symmetrisch, darauf wird jedoch automatisch geachtet, d.h. die Abhängigkeiten brauchen nur einmal formuliert zu werden.

Der Ausdruck in jeder Zeile wird ausgewertet, und wenn einer „falsch“ zurückgibt, wird angenommen, dass die Kombinationen **nicht** vereinbar sind. Wie bei *f4-implicit.txt* werden COMMUN- und K-Moduln nicht mitgezählt.

Der Inhalt der Datei könnte z.B. wie folgt aussehen:

```
!(({13} && {11}) && [12])
```

Dies bedeutet, dass wenn die eine Kombination die Moduln 11 und 13 enthält, die andere nicht 12 enthalten darf.

f6-implicit.txt

In dieser Datei wird angegeben, welche Moduln andere abdecken. Die Funktionalität entspricht somit im Prinzip der Angabe von „U“ in *f2-abhaengigkeiten.txt*, *f6-implicit.txt* erlaubt aber noch viel mehr.

Die Datei wurde primär eingeführt um zu ermöglichen, dass die über *f4-implicit.txt* definierten technischen Moduln nicht in der Abdeckungsrechnung berücksichtigt werden (da hier durchaus eine Kombination ein technisches Modul vorschreiben kann, aber dennoch von einer Kombination abgedeckt wird, die dieses Modul nicht beinhaltet).

Der durch diese Datei erzeugte Code wird aufgerufen mit der Aufforderung zu prüfen, ob das Modul i von der Kombination die durch [] erreichbar ist, abgedeckt wird. Der Ausdruck in jeder Zeile wird ausgewertet, und wenn einer „wahr“ zurückgibt, wird angenommen, dass die i abgedeckt wird. Wie bei *f4-implicit.txt* und *f5-implicit.txt* werden COMMUN- und K-Moduln nicht mitgezählt.

Der Inhalt der Datei könnte z.B. wie folgt aussehen:

```
(i>11)
(i==4) && ([10]&&![9])
```

Das bedeutet, dass Moduln über 11 technische Moduln sind und für die Abdeckungsrechnung nicht berücksichtigt werden. Weiterhin wird das Modul 4 abgedeckt, wenn das Modul 10 vorhanden ist, und 9 **nicht** vorhanden ist. (natürlich wird das Modul 4 auch abgedeckt, wenn es selber vorhanden ist, also eine Zeile wie $(i == 4)\&\&[4]$ oder kürzer $[i]$) muss somit nie auftauchen, da entsprechender Code bereits vorher ausgeführt wurde. Alle Zeilen werden mit ODER verknüpft, d.h. das Ergebnis ist TRUE wenn nur eine einzige Zeile TRUE zurückgibt.

data/commun.ascii

Diese Datei enthält in der ersten Zeile den Index des Basismoduls (COMMUN-Modul). Dabei ist von **0** an zu zählen. Der Rest der Datei wird ignoriert.

A.2.2 getrennte Vorbereitung

Die drei ersten der vier oben beschriebenen Dateien mit den Ausgangsdaten werden durch Shell-Skripte (*f?-preconvert*) vorbehandelt. Dabei werden überflüssige Informationen entfernt. Durch die Einführung von Zeilenumbrüchen werden die Dateien weiterhin für C++-Programme lesbarer gemacht.

Die eigentlichen Daten werden weder analysiert noch sonst verändert. Es entstehen die Dateien *f1-modulnamen.ascii*, *f2-abhaengigkeiten.ascii* und *f3-bestellungen.ascii*.

A.2.3 Zusammenfassung und erste Analyse

Das Programm *input-filter* liest die drei *.ascii*-Dateien ein und führt eine erste Analyse durch. Enthält die Abhängigkeitsmatrix K-Einträge, so bedeutet dies, dass es Moduln gibt, die immer eingebaut werden. Der Preis dieser Moduln wird dann dem COMMUN-Modul zugeschlagen und vom übergeordneten Modul abgezogen, damit der Preis, wenn die erweiterte Funktionalität bestellt wurde, nicht doppelt gezählt wird.

Danach werden diese K-Moduln aus der Modulliste, allen Bestelllisten und der Abhängigkeitsmatrix entfernt. Auch das COMMUN-Modul wird aus der Liste und der Matrix entfernt, da es für die Berechnung unwichtig ist. Der (um K-Modulpreise erhöhte) Preis des COMMUN-Moduls wird mit der Summe der Anzahl der bestellten Varianten aller Bestellungen multipliziert und in eine über Parameter spezifizierte Datei geschrieben. Dieser Preis wird später von den Ausgabefiltern addiert, um wieder auf den Gesamtpreis zu kommen.

Abschließend gibt *input-filter* die Liste der Moduln (Anzahl jeweils vorangestellt), die Liste der (verbleibenden) Abhängigkeiten und die Bestelliste aus. Üblicherweise sorgt *make* für die Speicherung der Ausgaben in der Datei *combined-list*.

A.2.4 Initialisierung des Optimierers

Das Hauptprogramm *optimierer* (entsteht aus der Datei *main.cpp*) liest die so entstandene *combined-list* wenn die zentralen Programmobjekte, die Modulliste (mit Abhängigkeitsmatrix) und die Bestelliste initialisiert werden.

Anschließend wird eine Instanz von **TProduktionsplan** mit der Bestellliste initialisiert und als „Urplan“ einer Instanz von **TOptimierung** übergeben. Danach wird der Optimierer interaktiv aufgefordert, verschiedene Heuristiken anzuwenden.

A.2.5 Ausgaben der Hauptprogramme

Auf Aufforderung gibt das Hauptprogramm Zwischenergebnisse in eine Datei aus. Dabei steht in der ersten Zeile immer der Preis der Bestellliste, d.h. der Produktionspreis ohne Unused Content. Danach wird die erzeugte Instanz von **TOptimierer** ausgegeben.

Sind nur die Ergebnisse für ein Intervall gewünscht, so werden, um eine mit der Komplettausgabe kompatible Ausgabe zu erhalten neben dem Preis der Bestellliste auch ein KSC-Wert (wird von der Datenaufbereitung ignoriert), die Anzahl der ausgebenen Pläne und die Anzahl der Moduln mit denen gerechnet wurde, den Daten des Produktionsplans vorangestellt. Außerdem wird jedem Produktionsplan eine „1“ vorangestellt, da bei der Intervallausgabe nur ein Produktionsplan pro Variantenanzahl ausgegeben wird.

Dadurch wird erreicht, dass die Ausgabefilter sowohl die Komplettausgabe (die sehr viel mehr Informationen enthalten), als auch das auf ein kleines Intervall beschränkte Endergebnis ohne Fallunterscheidung behandeln können.

Das Gleiche gilt auch für den Warmstart des Optimierers. Dieser kann aus einer Komplettausgabe oder auch nur aus dem Endergebnis heraus erfolgen.

A.2.6 Extraktion der Produktionspläne

Die Ausgabe des Hauptprogramms wird zunächst von dem Shell-Skript *globlist-postconvert* in die einzelnen Produktionspläne aufgespalten. Dabei werden die Zusatzinformationen (Preis der Modulliste, Anzahl der Moduln) ausgewertet und durchgereicht.

A.2.7 Ausgabe nach HTML

Das Shell-Skript *htmlizer* übernimmt dann die Ausgabe des jeweiligen Produktionsplanes als HTML-Tabelle. Dabei hilft das C++-Programm *extend-prod.cpp*, welches die von den Eingabefiltern entfernten K-Moduln ergänzt.

A.2.8 Erstellung einer Übersicht

Das Shell-Skript *createhtmlindex* erzeugt schließlich eine Schnellübersicht, aus der sich die minimalen Preise für die jeweilige Anzahl an Varianten ablesen lassen. Der erzeugte Index enthält Links zu den entsprechenden Produktionsplänen.

A.3 Datenvorbereitung

A.3.1 Das „Urformat“

Das Format der Eingangsdaten wurde bereits unter A.2.1 auf Seite 68 erläutert.

Hier sei noch einmal daran erinnert, dass die Eingangsdaten aus sieben Dateien bestehen:

f1-modulnamen.txt Modulnamen und Preise

f2-abhaengigkeiten.txt Abhängigkeitsmatrix

f3-bestellungen.txt Bestellliste

f4-implicit.txt technische Moduln

f5-implicit.txt unvereinbare Modulkombinationen

f6-implicit.txt Abdeckung von Moduln

commun.ascii Index des „COMMUN"-Moduls

A.3.2 Modulnamenvorverarbeitung

Das Shell-Skript *f1-preconvert* bekommt als Parameter den Dateinamen *f1-modulnamen* übergeben. Aus dieser Datei werden die Anzahl der Moduln sowie die Modulnamen und -preise bestimmt.

Zunächst gibt das Skript die Anzahl der Moduln aus. Danach werden erst alle Modulnamen und dann alle Modulpreise zeilenweise ausgegeben. Bei den Modulpreisen wird das Komma entfernt, so dass die Angabe in Cents erfolgt.

Da der Quellcode extrem kurz ist, sei er hier zitiert:

```
#!/bin/sh
# Anzahl der Moduln ausgeben
head -n 1 $1 | sed -e "s/;;;//" -e "s/;$//" | tr ";" '\012' \
      | sed -e "s/\"//g" | wc -l | sed -e "s/ //g"
# Modulnamen ausgeben (einer pro Zeile)
head -n 1 $1 | sed -e "s/;;;//" -e "s/;$//" | tr ";" '\012' \
      | sed -e "s/\"//g"
# Modulpreise ausgeben (einer pro Zeile)
head -n 3 $1 | tail -n 1 | \
      sed -e "s/;;\"Price of option (FF)\";//" -e "s/,//g" \
      | tr ";" '\012'
```

A.3.3 Verarbeitung der Abhängigkeitsmatrix

Die Abhängigkeitsmatrix wird durch das Shell-Skript *f2-preconvert* vorverarbeitet. Zunächst wird die Anzahl der Zeilen bestimmt. Da die ersten drei Zeilen ignoriert werden, ist die Anzahl der Moduln gleich der Anzahl der Zeilen minus 3.

```
#!/bin/sh
lines=`cat $1 | wc -l`
lines=`bexpr $lines - 3`
```

Nun wird zeilenweise die $i + 3$-te Zeile aus der Abhängigkeitsmatrix isoliert und verarbeitet. Für die Verarbeitung werden zunächst alle Semikolons in Zeilenumbrüche („012") umgewandelt, und durch Auswahl der letzten *lines*-Zeilen die Einträge vor dem ersten relevanten Semikolon abgeschnitten. Danach werden die Anführungszeichen, „B“'s und Leerzeichen entfernt und die Zeilenumbrüche wieder durch Semikolons ersetzt. Ein abschließendes echo sorgt dafür, dass die Ausgabe wieder Zeilenweise erscheint.

```
i=0;
while (test $i -ne $lines)
do
im1=i;
lmi=‘bexpr $lines - $i‘
i=‘bexpr $i + 1‘
tail -n $lines $1 | head -n $i | tail -n 1 \
 | tr ";" '\012' | tail -n $lines  \
 | sed -e "s/\"//g" -e "s/B//g" -e "s/ //g" \
 | tr '\012' ";"
echo
done
```

A.3.4 Verarbeitung der Bestellliste

Die Bestellliste wird durch das Shell-Skript *f3-preconvert* vorverarbeitet. Dabei werden die ersten drei und die letzte Zeile ignoriert. Die Anzahl der restlichen Zeilen (also alle Zeilen minus 4) entspricht der Anzahl der Bestellungen. Jede der übrigen Zeilen (also vierte bis einschließlich der vorletzten) enthält die Angaben zu genau einer Bestellung.[9]

Im Skript wird jede dieser Zeilen einzeln abgearbeitet. Zunächst wird der Inhalt der gesamten Zeile in der Variablen *spalte* gespeichert. In *rspalten* wird dann die (tatsächliche) Anzahl der Moduln der Bestellung gespeichert. Anschließend wird erst die bestellte Stückzahl und dann die einzelnen Moduln mittels echo ausgegeben. Dabei wird nur das Semikolon zwischen den Moduln entfernt, d.h. in der Ausgabe steht die gesamte 0-1-Folge in einer Zeile.

```
spalte=‘tail -n $lines $1 | head -n $i | tail -n 1‘
spalten=‘echo $spalte | tr ";" '\012' | wc -l‘
spalten=‘bexpr $spalten - 3‘
rspalten=‘bexpr $spalten - 1‘
echo $spalte | tr ";" '\012' | head -n 2 | tail -n 1 \
 | tr '\012' ";"  | sed -e "s/;//g"
echo
echo -n $spalte | tr ";" '\012' | tail -n $spalten \
 | head -n $rspalten | tr '\012' ";" | sed -e "s/;//g"
echo
```

A.3.5 Zusammenfassung der Daten und erste Analyse

Das Programm *input-filter.cpp* liest die von den drei oben erläuterten Shell-Skripten produzierten Dateien ein und produziert daraus eine für das Hauptprogramm lesbare Ausgabedatei.

Dabei werden die K-Abhängigkeiten und das COMMUN-Modul eleminiert, da sie für das eigentliche Problem irrelevant sind: das COMMUN-Modul kommt

[9]Mehr zum Format der Eingabedatei steht im Abschnitt A.2.1 auf Seite 69.

in jeder Bestellung vor, daher ist hier in keinem Fall zu entscheiden, ob es in einer Produktionsvariante vorkommt oder nicht.

Bei den K-Abhängigkeiten verhält es sich ähnlich: nach der Definition der K-Abhängigkeiten enthält das Spaltenmodul alle Funktionen des Zeilenmoduls. Weiterhin muss das Zeilenmodul dann eingebaut werden, wenn das Spaltenmodul nicht eingebaut wurde.

Zieht man also den (kleineren) Preis des Zeilenmoduls vom Preis des Spaltenmoduls ab und fügt ihn dafür dem COMMUN-Modulpreis hinzu, ergeben sich für jede Bestellvariante und für jede Produktionsvariante die gleichen Preise und Abdeckungsverhältnisse. Da dann aber das Zeilenmodul den Preis 0 hat, kann es auf das Ergebnis der Optimierung keinen Einfluß mehr haben.

Natürlich ist es für die Fabrikation am Ende praktisch, Informationen über das Zeilenmodul im Ergebnis zu haben. Da es dann jedoch exakt komplementär zum Spaltenmodul auftaucht, kann die Zeilenmodul-Information durch den Ausgabefilter ergänzt werden.[10]

A.3.6 Übergabe an den Optimierer

Die Ausgabe der Daten erfolgt durch den Eingabefilter, wobei alle Einträge jeweils in einer Zeile stehen, in der Reihenfolge:

1. Anzahl der Moduln
2. Modulliste mit Modulnamen und Preis
3. Anzahl der Abhängigkeiten (ohne K-Fälle)
4. Für jede Abhängigkeit: Typ (gekennzeichnet durch den jeweiligen Buchstaben) und ein Paar von Moduln (indiziert, beginnend bei 0)
5. Anzahl der Bestellungen
6. Für jede Bestellung: Stückzahl und danach die Modulkombination, 0 oder 1, auch hier: nur ein Wert (0 oder 1) pro Zeile!

Die so erzeugten Daten können vom Hauptprogramm unmittelbar eingelesen werden. Gemäß der Voreinstellung werden sie in der Datei *data/combined-list* zwischengespeichert.

A.4 Datenstrukturen

A.4.1 Das Modul-Objekt: TModuln

Die Definition dieses Objektes erfolgt in den Dateien *module-info.cpp* bzw. *module-info.h*. Jede der Instanzen repräsentiert ein Modul. Gespeichert werden der Modulname und der Modulpreis. Das Objekt kann mit den Operatoren $>>$ bzw. $<<$ in einen Stream gespeichert bzw. aus einem Stream gelesen werden.

Das Objekt erwartet in der ersten Zeile der Modulname und in der zweiten Zeile der Modulpreis. Der Operator $==$ dient dem Vergleich von Instanzen vom Typ TModuln. Da lediglich der Name der Moduln verglichen wird, sollten alle Moduln unterschiedliche Namen haben.

[10] Dieser ergänzt auch das COMMUN-Modul.

Im aktuellen Programm erfolgt auf die Modulnamen jedoch kein Zugriff, da eine Inidzierung über die Modul-Liste erfolgt.

A.4.2 Die Modul-Liste

Das Objekt **TModulnList** wird in den Dateien *module-list.cpp* bzw. *module-list.h* definiert. Diese zentrale Datenstruktur erlaubt den Zugriff auf die Moduln über einen Index. Da das „Löschen“ von Moduln nicht erlaubt ist, sind die Inidizes für die Laufzeit des Programms eindeutig.

Neben den Moduln (mit Preisen und Namen) speichert die Modulliste auch die Abhängigkeiten zwischen den Moduln.

A.4.3 Die Modul-Kombination

Das in der Datei *module-comb.h* definierte Objekt **TModulnCombination** dient der Speicherung einer Modulkombination. Dabei wird für jedes Modul der Modulliste gespeichert, ob es in der Kombination enthalten ist („1“) oder nicht („0“). Für die Speicherung wird ein Vektor vom Typ `bool` verwendet.

Neben dieser Information speichert das Objekt noch den Preis der Modulkombination. Ddieser muss dadurch nicht jedesmal neu berechnet werden.

Das Objekt enthält Methoden um abzufragen, welche Moduln in der Kombination enthalten sind, wie teuer die Kombination ist, ob eine andere Kombination abgedeckt wird und ob zwei Kombinationen kombiniert werden dürfen. Dazu wird der aus den Eingangsdateien *f?-implicit.txt* erzeugte C++-Code eingebunden.

Zu einer Kombination können einzelne Moduln oder andere Kombinationen (sofern erlaubt) hinzugefügt werden. Das Objekt ist auch in der Lage, sich in einem Stream zu speichern bzw. daraus ausgelesen zu werden. Weiterhin sind Methoden vorhanden, die den Vergleich von Kombinationen (enthalten, gleich, ungleich) erlauben.

TModulnCombination wird von den Objekten **TProduktionsPlan** und **TOrderList** intensiv verwendet.

A.4.4 Die Bestellliste

In diesem Objekt werden die bestellten Modulkombinationen mit den dazugehörigen Stückzahlen gespeichert. Für die bestellten Kombinationen wird dazu ein Vektor vom Typ **TModulnCombination** verwendet.

Neben dem Zugriff auf die Bestellkombinationen und deren Stückzahlen ermittelt **TOrderList** auch den Gesamtpreis der Bestellungen, also den Preis ohne Unused Content. Auch **TOrderList** kann aus einem Stream gelesen bzw. in einen Stream gespeichert werden.

A.4.5 Die Unused-Content Matrix

Die UC-Matrix wurde im Quellcode als **TUsefulCombinationList** bezeichnet. In diesem Objekt wird gespeichert, wieviel es kostet, die Bestellung $b \in B$ durch eine Kombination $k \in K$ zu ersetzen. Gespeichert wird der Preis des Unused-Content multipliziert mit der Stückzahl der Bestellung. Deckt die Kombination

k die Bestellung b nicht ab, wird ein sehr großer Wert (100.000-facher Wert des Auftrags) — symbolisch für ∞ — gespeichert.

Die UC-Matrix verwaltet eine Liste *my-list* der Kombinationen K, für die die Abhängigkeiten gespeichert wurden. Weiterhin wird die Matrix *my-dependencies* mit den jeweiligen Preisdifferenzen gespeichert.

Da der Aufwand die UC-Matrix zu durchsuchen schnell sehr groß wird, empfiehlt es sich, diese von Zeit zu Zeit zu reduzieren. Das Hauptprogramm bietet dazu zur Zeit nur die Möglichkeit auf $K = B$ zurückzufallen.

Das Objekt **TUsefulCombinationList** stellt jedoch Methoden zum Hinzufügen und Entfernen von einzelnen Kombinationen zur Verfügung. Die UC-Matrix wird von den Methoden `melt` und `grow` des Objektes **TProduktionsplan** genutzt.

A.4.6 Der Produktionsplan

Das Objekt **TProduktionsPlan** ist das zentrale Objekt des gesamten Programms, da ein (möglichst) optimaler Produktionsplan gefunden werden soll.

Folgende Daten werden vom Produktionsplan gespeichert:

my-plan Vektor, der die zu produzierenden Modulkombinationen, also eine Liste von Objekten vom Typ **TModulnCombination**, enthält.

my-numbers Vektor, der die zu denen in *my-plan* beschriebenen Kombinationen zugehörigen Stückzahlen enthält.

my-assignment Vektor, der jeder Bestellung der Bestellliste einen Index auf ein Element von *my-plan* zuordnet, und somit angibt, welche Modulkombination diese Bestellung abdeckt. *my-assignment* enthält immer genau so viele Einträge wie die Bestellliste.

my-costs Enthält die Gesamtkosten des Produktionsplans.

my-sorting Enthält eine Permutation der Indizes von *my-plan* bzw. *my-numbers*, die bei indirekter Addressierung dieser Variablen zu einer Sortierung nach Stückzahl führt.

is-sorted Gibt an, ob *my-sorting* korrekt berechnet wurde. Da *my-sorting* auf Grund des hohen Rechenaufwandes nicht immer aktualisiert wird, kann hiermit überprüft werden, ob *my-sorting* aktuell ist. Wird durch eine Operation die Sortierung möglicherweise zerstört, ist *is-sorted* auf `false` zu setzen.

Neben grundlegenden Methoden zur Initialisierung, zum Zugriff auf die gespeicherten Daten oder zur Eingabe und Ausgabe stellt das Objekt **TProduktionsPlan** zwei zentrale Gruppen von Methoden zur Verfügung.

Die erste Gruppe stellt rein technische Realisierungen elementarer Operationen wie des Zusammenfügens von zwei Kombinationen zu einer einzigen oder dem Spalten einer Kombination (die bisher mehrere Bestellkombinationen abdeckte) in zwei neue bereit. Insbesondere werden dabei Änderungen am Gesamtpreis (*my-costs*), den Stückzahlen (*my-numbers*) und der Zuordnung zu den Bestellungen (*my-assignments*) berücksichtigt. Im Quellcode sind diese Methoden unter `Mid-Level-Routines` zu finden.

Auf diesen Methoden baut die zweite Gruppe von Methoden, die zentralen Algorithmen, auf. Hier wird der Produktionsplan analysiert. Auf der Grundlage verschiedenster bereits diskutierter Strategien erzeugen diese Methoden neue Produktionspläne. Folgende Methoden gehören zu dieser Gruppe:[11]

`combine` Entspricht dem Algorithmus „Kleben“

`combine-limited` Entspricht dem Algorithmus „Schnellkleben“

`split` Entspricht dem Algorithmus „0-1-Spalten“

`split-mm` Entspricht dem Algorithmus „Spalten mit maximalen Modulkombinationen“

`grow` Entspricht dem Algorithmus „Varianten hinzufügen“

`melt` Entspricht dem Algorithmus „Varianten entfernen“

Alle Algorithmen geben eine Liste der $\boldsymbol{k}$-besten gefundenen neuen Produktionspläne zurück. Der übergebene Parameter $\boldsymbol{k}$ stellt dabei den Maximalwert da. Der `return`-Wert der Methoden gibt die exakte Anzahl der zurückgegebenen Produktionspläne an. 0 bedeutet also insbesondere, dass der Algorithmus keinen einzigen Produktionsplan finden konnte.

Allen Algorithmen wird weiterhin ein Limit übergeben, welches entweder angibt wie stark der Preis maximal ansteigen (minimal fallen) darf bzw. welcher Preis unterboten werden muss. Durch Angabe guter Schranken kann dann der Algorithmus schneller ablaufen. Zur genauen Spezifikation aller Parameter sei hier auf die Header-Datei *module-prod.h* bzw. den Quellcode *module-prod.cpp* verwiesen.

A.4.7 Der Optimierer

Das Objekt **TOptimierer** verwaltet alle im Moment interessanten Produktionspläne. Entsprechend der Idee vom κ-gepufferten Kleben bzw. Spalten werden für jede Variantenanzahl κ Pläne gespeichert. Wie groß κ ist, speichert **TOptimierer** in der internen Variable *my-ksc*. Dieser Wert kann jederzeit mit der Methode `change-ksc` geändert werden.

TOptimierer kann (mit allen Produktionsplänen) in einem Stream gespeichert und wieder daraus gelesen werden. Neben technischen Methoden zur Verwaltung der Datenstrukturen stellt **TOptimierer** folgende Heuristiken zur Verfügung:

`varianten-entfernen-heuristik` Verwendet die Methode `melt` des jeweils besten Produktionsplanes um Produktionspläne mit einer Variante weniger zu erhalten.

`varianten-hinzufuegen-heuristik` Verwendet die Methode `grow` des jeweils besten Produktionsplanes um Produktionspläne mit einer Variante mehr zu erhalten.

[11] Vorsicht: einige der Namen wurden polymorph verwendet, daher im Quellcode am besten unter `High-Level-Routines` nachsehen!

`schnellkleben-heuristik` Verwendet die Methode `combine-limited` um aus dem besten Produktionsplan Produktionspläne mit einer Variante weniger zu erhalten.

`kleben-heuristik` Verwendt die Methode `combine` um aus allen κ-besten Produktionsplänen Produktionspläne mit einer Variante weniger zu erhalten.

`iks-heuristik` Verwendet die Methoden `combine`, `split` und `split-mm` um aus allen κ-besten Produktionsplänen im übergebenen Intervall neue Produktionspläne zu erhalten. Dabei wird im Intervall solange iteriert, bis sich die Daten nicht mehr ändern oder eine vorgegebene Anzahl an Iterationen überschritten wird.

`create-final-set-max-uc` Unter Verwendung der Methode `iks-heuristik` wird der Produktionsplan mit der geringsten Variantenanzahl gesucht, der unter ***max-uc***% unused-content enthält.

`create-final-set-max-ap` Unter Verwendung der Methode `iks-heuristik` wird der kostengünstigste Produktionsplan gesucht, wobei pro Variante ***steigung***% Mehrkosten anfallen.

Alle Heuristiken beachten dabei den κ-Wert und fordern beim Produktionsplan nur entsprechend viele neue Pläne an. Die Ergebnisse werden anschließend mit den bereits bekannten Plänen verglichen und nur die κ besten bleiben bestehen.

Sofern nicht anders beschrieben, wird jeweils bei einem Extrem begonnen (maximale bzw. minimale Anzahl an Produktionsvarianten) und dann mit der angegebenen Methode bis zum anderen Extrem der Algorithmus aufgerufen.

A.5 Basisalgorithmen

Alle in diesem Abschnitt vorgestellten Algorithmen werden vom Objekt **TProduktionsplan** aus der Datei *module-prod.cpp* bereitgestellt.

A.5.1 Combine

Die Methode `Combine` dient dazu bestimmte Kombinationen des Produktionsplans zusammenzufassen. Sie bestimmt nicht, welche Kombinationen zusammengefasst werden sollen. Auch wird vorausgesetzt, dass die anhand der Aufrufparameter spezifierte Zusammenführung erlaubt ist.

`Combine` werden die Indizes der beiden zusammenzuführenden Varianten übergeben. Die Indizes beziehen sich dabei auf die Variantenliste des jeweiligen Produktionsplanes. Sind die Indizes größer als die Variantenliste wird ein Ausnahmefehler („Combination-Not-Included“) ausgelöst.

`Combine` führt folgende Operationen durch:

- Zusammenfassen der beiden Kombinationen (die Methode `add-combination` des **TModulnCombination**-Objektes berücksichtigt dabei eventuell zu beachtende Abdeckungen.
- Neuberechnung der Produktionskosten
- Aktualisierung der Abdeckungsinformationen

- Löschen der Sortierung nach Stückzahl (wird danach nur bei Bedarf wiederhergestellt). Die Sortierung wird nicht aufrechterhalten, da nur ein kleiner Teil der erzeugten Kombinationen überhaupt weiter verwendet wird. Und bei einem noch kleineren Anteil wird dann wieder eine Sortierung vorausgesetzt.

`Combine` wird von den Algorithmen `Schnellkleben`, und `Kleben` verwendet.

A.5.2 Split

Das Gegenstück zu `Combine` ist `Split`. Diese spaltet eine Produktionsvariante in zwei. Dazu werden ihr zwei Listen von Bestellungen (Indizes) übergeben. Die Bestellungen müssen vorher von einer einzigen Produktionsvariante abgedeckt worden sein. Keine der beiden Listen darf leer sein.

Analog zu `Combine` führt `Split` folgende Operationen durch:

- Erzeugen und Einfügen der beiden neuen Kombinationen durch Kombination der Bestellvarianten. Die auch hier benutzte Methode `add-combination` des **TCombination**-Objektes berücksichtigt dabei eventuell zu beachtende Abdeckungen.
- Neuberechnung der Produktionskosten
- Aktualisierung der Abdeckungsinformationen
- Löschen der Sortierung nach Stückzahl.

`Split` wird von den Algorithmen `0-1-Spalten` und `Spalten nach Maximalen Modulkombinationen` verwendet.

A.5.3 Maximale Modulkombinationen

Die Methode `find-max-modcombinations` soll die maximalen Modulkombinationen in einer Menge von Bestellvarianten finden. Maximal sind jene Modulkombinationen, die keine gegenseitigen Inklusionsbeziehungen mehr erfüllen.

Im Objekt **TProduktionsplan** wird der Methodenname `find-max-modcombinations` zweimal verwendet. Grob kann die erste Definition als Hilfsfunktion bezeichnet werden. Diese erste sucht tatsächlich die maximalen Modulkombinationen. Die zweite dient dazu, ein sinnvolles Ergebnis für eine Spaltung auch dann herzustellen, wenn es nur eine maximale Modulkombination gibt. Auch ist für das reine Ziel eine Spaltung herzustellen, das Interface günstiger gewählt.

`find-max-modcombinations`: Hilfsfunktion

Der Hilfsfunktion wird im Parameter ***info*** eine Liste von Bestellvarianten[12] übergeben. Unter diesen sind die maximalen Varianten zu finden und im Zeiger ***result*** zurückzugeben.

Das Programm entspricht der Beschreibung aus 2.1.1 von Seite 11, wobei durch die Benutzung der Methode `module-combination-included` des Objektes **TModulnCombination** auch die Abhängigkeitsmatrix berücksichtigt wird.

[12]Die Indizes beziehen sich auf globale Bestellliste.

`find-max-modcombinations`: Vorbereitung für Spalten

Dieser Methode wird ebenfalls der Paramter ***info*** mit einer Liste der zu untersuchenden Bestellvarianten übergeben. Als Ergebnis wird jedoch eine (kürzere) Liste von Bestellvarianten zurückgegeben die eine Hälfte der entstehenden Spaltung enthält.

Diese Liste entsteht, indem zunächst die Hilfsfunktion aufgerufen wird. Existiert mehr als eine maximale Modulkombination, so wird die erste Maximale Modulkombination als Ergebnis in die Liste eingetragen und die Methode beendet.

Existiert nur genau eine maximale Modulkombination, so wird diese aus der übergebenen Liste entfernt und erneut die Hilfsfunktion aufgerufen. Da natürlich vorausgesetzt wird, dass ***info*** mindestens zwei Kombinationen enthält, gibt die Hilfsfunktion eine auf die kleinere Menge bezogene maximale Modulkombination zurück. Diese wird dann als Ergebnis (über ***list[0]***) zurückgegeben.

Abschließend erfolgt in ***list*** die Eintragung jener Kombinationen, die von der gefundenen maximalen Modulkombination abgedeckt werden.

A.5.4 Schnellkleben

Die Methode `combine-limited` sortiert die Liste der produzierten Varianten nach deren Stückzahl und versucht dann die ***goal*** Kombinationen mit der geringsten Stückzahl jeweils paarweise zusammenzufassen. Die ***k*** besten so gefundenen Produktionspläne werden dann in ***result*** zurückgegeben.[13]

Über den optionalen Parameter ***limit*** kann dafür gesorgt werden, dass Kombinationen die einen höheren **Preisanstieg** als ***limit*** haben, nicht berücksichtigt werden, d.h. wenn die Abschätzung

$$\text{Stückzahl}_1 \cdot (\text{Preis}_2 - \text{Preis}_1) > \boldsymbol{limit}$$
$$\text{oder}$$
$$\text{Stückzahl}_2 \cdot (\text{Preis}_1 - \text{Preis}_2) > \boldsymbol{limit}$$

erfüllt ist, wird die eigentliche Kombination nicht weiter betrachtet. Da die genaue Überprüfung der Kombinierbarkeit, so wie die Berechnung des Preises der Kombination relativ aufwendig ist, können gute Abschätzungen an dieser Stelle den Algorithmus stark beschleunigen. Diese Abschätzung ist erlaubt, da für die Zusammenfassung zweier Kombinationen gilt:

$$\text{Gesamtpreis} \cdot \text{Gesamtstückzahl} \geq (\text{Stückzahl}_1 + \text{Stückzahl}_2) \cdot \max\{\text{Preis}_1, \text{Preis}_2\}$$

Der Algorithmus verbessert anhand des Kriteriums „die ***k***-besten“ eigenständig die vorgegebene Schranke.

A.5.5 Kleben

Diese Heuristik wird von der Methode `kleben-heuristik` bereitgestellt. Nur aus dem besten bekannten Plan für jede Variantenanzahl werden dabei die κ besten Produktionspläne mit einer weiteren Bestellvariante mit Hilfe der Methode `grow` (beschrieben unter A.5.9) bestimmt und im Optimierer gespeichert.

[13] Klar: ***result*** kann zwischen 0 und ***k*** Einträge am Ende enthalten.

Beim Kleben wird Schnellkleben über alle Kombinationen durchgeführt, d.h. es wird Schnellkleben mit der Anzahl der Varianten als ***goal*** aufgerufen. Im Programm wird der Heuristik „Kleben“ eine Sortierung „vorgetäuscht“, damit diese nicht unnötigerweise sortiert.

A.5.6 0-1-Spalten

Der Algorithmus `split` bearbeitet all jene Produktionsvarianten, denen mehr als eine Bestellung zugeordnet wurde. Diese werden dann wie weiter unten beschrieben gespalten. Die κ besten so gefundenen Produktionspläne werden zurückgegeben. Die Anzahl der Varianten steigt dabei genau um 1.

Wurden einer Produktionsvariante genau zwei Bestellungen zugeordnet, wird diese Produktionsvariante aus dem Plan entfernt und die beiden Bestellungen werden einzeln aufgenommen.

Bei genau drei Bestellungen werden alle drei Möglichkeiten, jeweils zwei Bestellvarianten zusammenzufassen ausprobiert. Dieser Aufwand ist immer noch viel geringer, als das ansonsten verwendete Verfahren:

Wurden einer Produktionsvariante mehr als drei Bestellungen zugeordnet, so wird für jedes Modul die Liste dieser Bestellungen in zwei Gruppen unterteilt,[14] in die Gruppe der Bestellungen, die dieses Modul enthalten, und die, die es nicht enthalten. In diese beiden Gruppen wird, sofern keine der Gruppen leer ist, die aktuelle Produktionsvariante unterteilt.

A.5.7 Spalten nach Maximalen Modulkombinationen

Der Algorithmus `split-mm` sucht zunächst eine Produktionsvariante, die mehr als eine Bestellung abdeckt. Bei zwei bzw. drei abgedeckten Bestellungen wird nichts ausgeführt, da `split` bereits alle Möglichkeiten ausprobiert.

Bei mehr als drei abgedeckten Bestellungen wird mit dem Algorithmus „maximale Modulkombinationen“ eine maximale Modulkombination *lists[0]*, die gleichzeitig nicht alle Varianten abdeckt, gesucht. Die von dieser Kombination abgedeckten Bestellungen bilden die eine, die nicht abgedeckten Bestellungen die andere Hälfte der Unterteilung.

A.5.8 Varianten entfernen

Dieser Algorithmus überprüft alle Varianten darauf, wieviel es kosten würde, sie aus dem Plan zu entfernen. Die zum Algorithmus gehörige Methode ist `melt`.

Vor der eigentlichen Berechnung werden alle Kombinationen des Produktionsplans in die (globale) UC-Matrix eingetragen. Dadurch kann später im Algorithmus für jede Kombination das Abdeckungsverhalten bei der globalen Matrix abgefragt werden.

Mit Hilfe der UC-Matrix werden dann die Kosten berechnet, die bei Entfernung jeder Variante anfallen würden. Jene κ Produktionspläne, die durch den Wegfall der κ günstigsten Kombinationen entstehen, werden zurückgegeben.

[14] Dabei werden nur jene Moduln betrachtet, die in der bisherigen abdeckenden (!) Produktionsvariante auch vorkommen. Alle anderen Moduln würden ja die Gruppe der Bestellungen in eine leere Menge (Modul kommt vor) und „alles“ (Modul kommt nicht vor) unterteilen.

A.5.9 Variante hinzufügen

Hier wird für jede der Varianten aus der UC-Matrix geprüft, wie groß die Absenkung der Produktionskosten wäre, wenn die Variante in den Plan aufgenommen wird. Die κ besten Ergebnisse gibt die Methode `grow` dann zurück.

Auch `grow` trägt zu Beginn alle aktuellen Varianten des Produktionsplans in die globale UC-Matrix ein.

A.6 Heuristiken

In diesem Abschnitt werden die verwendeten Heuristiken beschrieben. Bisher laufen alle Heuristiken nach dem gleichen Schema ab. Durch Anwendung einer der Algorithmen des Objektes **TProduktionsplan** wird die Anzahl der Varianten vergrößert bzw. verkleinert.

Die jeweils κ besten so gefundenen Varianten (die bereits vorher bekannten mitgezählt) werden weiterverwendet. Für jede Variante speichert der Optimierer in *big-mod-tag*, welche Heuristiken bereits auf sie angewendet wurden. Dadurch können (nutzlose) Mehrfachanwendungen der gleichen Heuristik auf die gleiche Variante vermieden werden.

Ändern sich jedoch die Rahmenbedingungen der Heuristik (z.B. der ***goal***-Parameter beim Schnellkleben oder die UC-Matrix beim Varianten entfernen bzw. hinzufügen) muss mit `set-modifications` die entsprechende Heuristik wieder „freigeschaltet“ werden.

A.6.1 Varianten entfernen

Diese Heuristik entspricht der Methode `varianten-entfernen-heuristik`. Mit Hilfe von `melt` werden die κ besten Produktionspläne um eine Variante erleichtert. Die Heuristik speichert die so gefundenen κ besten Produktionspläne im Optimierer.

A.6.2 Varianten hinzufügen

Diese Heuristik wird von der Methode `varianten-hinzufuegen-heuristik` bereitgestellt. `grow` entfernt aus den κ besten Produktionsplänen eine Variante. Die so gefundenen κ besten Produktionspläne werden anschließend im Optimierer gespeichert.

A.6.3 Schnellkleben Heuristik

Die Schnellklebe-Heuristik verwendet den Algorithmus Schnellkleben um aus dem jeweils besten bekanntem Produktionsplan mit Variantenzahl p neue κ beste Produktionspläne mit Variantenzahl $p-1$ zu gewinnen. Der übergebene Parameter ***goal*** wird an „Schnellkleben“ durchgereicht und spezifiziert wieviele Kombinationen höchstens getestet werden.

A.6.4 κ-gepuffertes Kleben

Beginnend mit den vorliegenden Produktionsplänen im Optimierer werden alle bekannten κ besten Produktionspläne jeder Variantenanzahl mit dem Algorithmus aus A.5.5 um jeweils eine Variante verkleinert. Die κ besten dann gefundenen Varianten werden weiterverklebt bis es nicht weiter geht.

A.6.5 Kleben und Spalten

Im mittels der Paramter definierten Intervall werden die κ besten bekannten Produktionspläne für diese Variantenzahlen immer wieder mit den Algorithmen 0-1-Spalten (A.5.6), Spalten nach maximalen Modulkombinationen (A.5.7) bzw. Kleben (A.5.5) bearbeitet, bis sich die Varianten nicht mehr ändern oder bis die Anzahl der Aufwärts-Abwärts-Iterationen (per Parameter übergeben) überschritten wird.

A.6.6 Abschlussheuristiken

Die Heuristiken `create-final-set-*` versuchen ausgehend von einer bekannten Kostenfunktion $A(p)$ das tatsächliche Optimum zu finden. Insbesondere wird dabei das p mitbestimmt. Die Heuristiken erlauben es, entweder einen maximalen Wert für den Unused Content[15] oder Mehrkosten pro Variante anzugeben.

Ausgehend von den bereits bestehenden Lösungen suchen diese Heuristiken das aktuelle Minimum p_0 und beginnen anschließend, in einem symmetrischen Intervall um dieses p_0 mit der `iks-heuristik` zu verbessern. Dabei wird p_0 nach jedem Durchlauf an die (möglicherweise) veränderten Lösungen angepasst.

Am Ende wird p_0 zurückgegeben.

A.7 Das Hauptprogramm

Das Hauptprogramm beginnt damit den Anwender nach einer Datei zu fragen, in der die Moduln und die Bestellliste bereitgestellt werden. Die Eingabefilter stellen eine solche Datei unter *data/combined-list* bereit. Danach folgt ein einfacher textbasierter Dialog in dem der Anwender den Optimierer steuern kann. Nicht dialogbezogene Ausgaben werden vom Optimierer auf den Standartfehlerkanal ausgegeben.[16]

Im folgenden werden die einzelnen Kommandos beschrieben.

Gesamtausgabe in DATEI (für Warmstart)

Dieses Kommando gibt alle zur Zeit bekannten Produktionspläne in eine Datei aus. Sinnvoll für die Weiterverwendung der Daten für einen Warmstart.

Ausgabe der besten Ergebnisse über ein INTERVALL

Dieses Kommando gibt nur die besten zur Zeit bekannten Pläne in einem einzugebenden Intervall in eine Datei aus. Sinnvoll, falls eine Betrachtung des aktuellen Ergebnisses gewünscht ist.

[15] in Prozent, bezogen auf das Gesamtvolumen ohne COMMUN-Modul und K-Moduln

[16] Dieser erscheint beim Aufruf des Optimierers mit *make* in einem gesonderten *xterm*-Fenster.

Schnellkleben durchführen

Dieses Kommando ruft die Schnellkleben-Heuristik auf. Der Parameter ***goal*** muss noch spezifiziert werden. Wurde ***goal*** im Vergleich zum letzten Aufruf vergrößert, wird automatisch das Schnellklebe- Modifikationsgedächtnis gelöscht.

Varianten entfernen im INTERVALL

Dieses Kommando ruft die Heuristik „Varianten entfernen" für ein zu spezifizierendes Intervall auf.

Varianten hinzufügen im INTERVALL

Dieses Kommando ruft die Heuristik „Varianten hinzufügen" für ein zu spezifizierendes Intervall auf.

Kleben im INTERVALL

Dieses Kommando ruft die Heuristik „Kleben" für ein zu spezifizierendes Intervall auf.

Kleben und Spalten im INTERVALL

Dieses Kommando ruft die iks-Heuristik (Intervall-Kleben-und-Spalten) für ein zu spezifizierendes Intervall und zu spezifizierende Iterationsanzahl auf.

Kleben und Spalten mit ZIELWERT Unused Content

Dieses Kommando ruft die Abschluss-Heuristik für bekannten maximalen Unused Content auf. Das optimale p wird ausgegeben.

Kleben und Spalten mit ZIELWERT Mehrkosten pro Variante

Dieses Kommando ruft die Abschluss-Heuristik für bekannte Mehrkosten pro Variante auf. Das optimale p wird ausgegeben.

Modifikationsgedächtnis löschen

Dieses Kommando löscht im Optimierer die Vermerke, dass eine bestimmte Heuristik bereits ausgeführt wurde. Notwendig ist dies, wenn die durch Kleben bzw. Spalten entstandenen neuen Varianten beim Varianten-Hinzufügen berücksichtigt werden sollen, Varianten-Hinzufügen jedoch vorher bereits durchgeführt wurde.

k-Wert ändern

Mit dieser Funktion kann der κ-Wert des Optimierers geändert werden.

Adjustment-Vorgaben ändern

Steuert ob beim Kleben bzw. Spalten ein „Adjustment“ durchgeführt werden soll. Beim „Adjustment“ wird geprüft, ob durch die neu dem Produktionsplan hinzugefügte Variante Bestellungen besser abgedeckt werden, als dies durch die alte, bisher der Bestellung zugeordnete Variante der Fall ist. Wieso dieses Verfahren die Lösung verbessern kann, wird im Abschnitt „Anpassung“ auf Seite 14 beschrieben.

Status anzeigen

Mit diesem Kommando werden diverse Informationen über den Status des Optimierers ausgegeben. Der Aufruf dient insbesondere zur Diagnose von Fehlern.

Variantenauswahl zurücksetzen

Die UC-Matrix wächst im Laufe des Verfahrens. Wird sie zu groß kann sie mit diesem Kommando wieder auf „Variantenliste gleich Bestellliste“ reduziert werden.

Warmstart: Optimierer aus DATEI initialisieren

Dieses Kommando initialisiert den Optimierer aus einer früher durch eine Ausgabe erzeugten Datei. Dieses macht insbesondere am Anfang Sinn, wenn auf Ergebnisse eines früheren Durchlaufes zurückgegriffen werden soll.

Optimierer beenden

Dieser Befehl beendet das Programm.

A.8 Datenaufbereitung

A.8.1 Extraktion der relevanten Teile

Das Skript *globlist-postconvert* steuert die Erzeugung der Ergebnisse. Da hier sehr viele Daten zusammengeführt werden, ist die Parameterliste relativ lang:

1. Pfad der Datei mit den Ergebnissen, in der Regel *data/results*.
2. Pfad zur Datei, die dem Preis des (um k-Fälle erweiterten) COMMUN-Moduls, multipliziert mit der Anzahl aller zu produzierenden Stückzahlen, enthält.
3. Pfad zur Datei mit der Abhängigkeitsmatrix, in der Regel *data/f2-abhaengigkeiten.ascii*.
4. Pfad zur Datei mit dem Index des COMMUN-Moduls, in der Regel *common.ascii*.

globlist-postconvert liest als erstes den Preis (in CENTS) für die maximale Bestellvariantenanzahl (also den Preis ohne jeglichen Unused Content) aus *data/results*. Der Preis des COMMUN-Moduls wird addiert und das Ergebnis in der Variablen *minimalprice* (in EURO bzw. FF) gespeichert. Die Anzahl der Moduln (mit denen der Optimierer gerechnet hat) und die Anzahl der Ergebnisse mit unterschiedlicher Variantenanzahl werden ebenfalls aus *data/results* ausgelesen. Durch Analyse der Datei *data/f2-abhaengigkeiten.ascii* wird die richtige Modulanzahl bestimmt und unter *realmodulecount* gespeichert.

```
#!/bin/sh
minimalprice=`head -n1 $1 | tail -n1`
totalgroups=`head -n3 $1 | tail -n1`
modulecount=`head -n4 $1 | tail -n1`
commonprice=`head -n2 $2`
commonprice=`bexpr $commonprice / 100`
minimalprice=`bexpr $minimalprice / 100`
minimalprice=`bexpr $minimalprice + $commonprice`
realmodulecount=`egrep K $3 | wc -l`
realmodulecount=`bexpr $realmodulecount + $modulecount + 1`
currentline=5
```

In *currentline* wird die aktuelle Zeile gespeichert, bis zu der *data/results* abgearbeitet wurde. In der anschließenden Schleife werden für alle Variantenanzahlen und alle κ-optimalen Lösungen[17] die Produktionskosten (*prodcost*) und die Anzahl der Varianten (*groupsize*) aus *data/results* ausgelesen und dann die einzelnen Produktionsvarianten (hier noch um die K-Moduln verkürzt) in eine nach der Variantenanzahl benannte Datei *htmlizer-preprod** geschrieben. Die Stückzahlen für die jeweilige Variante werden in der Datei *htmlizer-numb** gespeichert, die Zuordnung zu den Bestellungen werden in *htmlizer-assi** abgelegt.

Anschließend wird das Hilfsprogramm *htmlizer* aufgerufen, welches aus diesen Daten eine HTML-Datei erstellt:[18]

```
  prodcost=`head -n$currentline $1 | tail -n1`
  prodcost=`bexpr $prodcost + $commonprice`
  currentline=`bexpr $currentline + 1`
  groupsize=`head -n$currentline $1 | tail -n1`
  currentline=`bexpr $currentline + 1`
  filename="$groupsize $currentgroupitem.html"
  filename=`echo $filename | tr " " _`
  groupline=`bexpr $modulecount \* $groupsize`
  currentline=`bexpr $currentline + $groupline - 1`
  head -n $currentline $1 | tail -n $groupline \
      > htmlizer_preprod$filename
  currentline=`bexpr $currentline + $groupsize + 1`
  head -n $currentline $1 | tail -n $groupsize \
      > htmlizer_numb$filename
  currentline=`bexpr $currentline + 1`
  assignmentcount=`head -n $currentline $1 | tail -n1`
  currentline=`bexpr $currentline + $assignmentcount`
  head -n $currentline $1 | tail -n $assignmentcount \
      > htmlizer_assi$filename
  currentline=`bexpr $currentline + 1`
```

[17]Bei der Ausgabe von Produktionsläufen gilt $\kappa = 1$

[18]Das Shell-Skript ist hier nicht vollständig wiedergegeben.

```
htmlizer $realmodulecount $groupsize $prodcost $filename \
    $minimalprice $3 $4 > $filename
```

htmlizer ruft zuerst das Programm *extend-prod* auf. Dieses fügt die vom Eingabefilter entfernten Moduln wieder hinzu.[19]

A.8.2 Hinzufügen der impliziten Moduln

Das Hinzufügen der impliziten (K)-Moduln erfolgt durch ein $C++$-Programm. Dieses liest zunächst die Abhängigkeitsmatrix ein und sucht K-Abhängigkeiten. Anhand dieser Abhängigkeiten wird bestimmt, wie die Modulpositionen mit denen der Optimierer gerechnet hat auf die realen Modulpositionen umzuschreiben sind, und wie die weiteren Positionen zu berechnen sind.

Danach wird die Variantenproduktionsliste eingelesen und zeilenweise die ergänzte Modulkombination ausgegeben. Dabei wird bereits HTML-Code verwendet. Die ausgegebenen Zeilen entsprechen jeweils dem Mittelteil einer Zeile im Endergebnis: Nummerierung, Stückzahl, Zuordnung zu den Bestellungen, sowie die `<tr>`-Tags fehlen noch.

A.8.3 Ausgabe nach HTML

Das Shell-Skript *htmlizer* generiert aus den von *module-prod* bzw. *globlist-postconvert* erzeugten Ausgaben eine HTML-Datei. Dabei wird *egrep* verwendet um die Zuordnungen zur jeweiligen Bestellung zu erhalten:

```
egrep -n "^$varnumber\$" htmlizer_assi$4 \
| sed -e "s/:$varnumber/,/g" -e "s/,\$//"
```

egrep liefert hier die Zeilennummern von Bestellungen, die der gerade betrachteten Produktionsvariantennummer *varnumber* zugeordnet sind. Mit *sed* werden überflüssige Ausgaben von *egrep* entfernt.

A.8.4 Erstellung einer Zusammenfassung

Das Shell-Skript *createhtmlindex* erstellt einen Index für die durch Parameter übergebenen HTML-Dateien. Dabei wird der Inhalt zwischen den TITLE-Tags als Linktext verwendet. Der HTML-Code wird auf die Standardausgabe ausgegeben.

Zunächst wird ein HTML-Kopf generiert,

```
#!/bin/sh
echo "<HTML><HEAD><TITLE>Result Overview</TITLE></HEAD><BODY>"
echo "<H1 ALGIN=CENTER>Result Overview</H1><p>"
echo "<UL>"
```

danach wird für jeden übergebenen Dateinamen ein Link erzeugt

[19] Diese Unterteilung der Shell-Skripte wurde vorgenommen, da so auf Multiprozessorrechnern leicht die Auswertung parallelisiert werden kann: ein & hinter der letzten zitierten Zeile aus *globlist-postconvert* genügt.

```
for n
# in *.html
do
 echo -n "<LI><A href="$n">"
 grep TITLE $n | sed -e "s/<HTML><HEAD><TITLE>//" \
                    -e "s/<\/TITLE><\/HEAD><BODY>//"
 echo ": $n</A></li>" | sed -e "s/.html//g"
done
```

und abschließend die HTML-Datei abgeschlossen:

```
echo "</UL>"
echo "</TABLE>"
echo "</BODY></HTML>"
```

A.9 Laufzeitverhalten

In diesem Abschnitt wird beschrieben, wieviel Zeit das Programm für welche Arbeitsschritte benötigt. Auch soll anhand zweier Beispiele die Effektivität der Algorithmen diskutiert werden. Da es sich hierbei um reine numerische Experimente handelt, sind die Ergebnisse nicht zu verallgemeinern.

A.9.1 Eingabefilter

Die Eingabefilter benötigen für die Bearbeitung von ein paar hundert Bestellungen nur ein paar Minuten. Die meiste Zeit wird dabei insgesamt für die Konvertierung der Bestellliste benötigt. Ursächlich ist die im Shell-Skript verwendete Methode zur Isolierung der einzelnen Zeilen, bei der der Aufwand bei $O(n^2)$ (n ist die Anzahl der Zeilen) liegt.

A.9.2 Ausgabefilter

Bei den Ausgabefiltern steigt der Aufwand im ersten Algorithmus quadratisch mit der Anzahl der ausgegebenen Produktionspläne. Da diese Anzahl jedoch in der Praxis gering (10) sein dürfte, und die eigentliche HTMLisierung einem Spezialprogramm übergeben wurde, liegt die Verarbeitungszeit auch hier im Bereich von Sekunden bis zu wenigen Minuten.

A.9.3 Hauptprogramm

Das Hauptprogramm kann, wie bereits erwähnt, mit sehr unterschiedlichen Parametern aufgerufen werden. Für alle Algorithmen gilt, dass die Laufzeit mit κ nahezu linear ansteigt. Auch die Anzahl der Moduln geht linear in den Aufwand ein.

Die Anzahl der Bestellungen geht in die Laufzeit der verschiedenen Algorithmen von linear bis quadratisch ein.

Die Laufzeit für die Abschlussoptimierung ist nicht ganz linear zur Anzahl der Iterationen i, da für jede weitere Iteration weniger sinnvolle Operationen noch möglich sind (insbesondere bei $i < k$).

Für die Abschlussoptimierung erscheint eine Intervallbreite von i sinnvoll. Die beiden Algorithmen mit automatischer Wahl der Modulanzahl berücksichtigen dies. Die Intervallbreite geht ebenfalls linear in die Laufzeit der Abschlussoptimierung ein.

Angesichts der Laufzeiten der Heuristiken kann die Zeit für Eingabe der Daten und Ausgabe der Ergebnisse vernachlässigt werden.

Für ca. 200 Bestellungen (nur ca. 80 verschiedene Bestellungen) und ca. 20 Moduln ergaben sich sich folgende Laufzeiten auf einem PII 350:[20]

Heuristik	Laufzeit	sonstige Parameter
Schnellkleben	1 min	$\kappa = 10$, $goal = 50$
Varianten entfernen	1 min	$\kappa = 10$, $[10, 30]$
Varianten hinzufügen	13 min	$\kappa = 10$, $[10, 30]$
κ-gepuffertes Kleben	1 min	$\kappa = 10$, $[10, 30]$
Aufwärts-Abwärts-Optimierung	2 min	$\kappa = 10$, $[10, 30]$

Dabei ist zu beachten, dass nachfolgende Algorithmen Laufzeitvorteile haben, da die Vorgänger erste Schranken geliefert haben. Interessant ist weiterhin, welche Algorithmen welchen Beitrag zur Annäherung an die „Optimallösung“ liefern. Dies hängt im Detail auch von den gewählten Paramtern ab. Ist z.B. *goal* kleiner oder κ größer, so ist es wahrscheinlicher, dass κ-gepuffertes Kleben noch einen Beitrag leisten kann. Hier die nicht repräsentativen Ergebnisse:[21]

Heuristik	bestes Ergebnis ($p = 20$)
Schnellkleben	6700014216
Varianten entfernen	6695978722
Varianten hinzufügen	6695978722
κ-gepuffertes Kleben	6695978722
Aufwärts-Abwärts-Optimierung	6674724546
Kosten ohne unused-content	6606670570

Später aufgerufene Heuristiken können keinen so großen Beitrag mehr liefern, wie die Startheuristiken. Falls die zweite und dritte Heuristik (Varianten hinzufügen bzw. entfernen) später ausgeführt werden, so ergibt sich folgendes Bild:[22]

Heuristik	bestes Ergebnis ($p = 20$)
Schnellkleben	6700014216
κ-gepuffertes Kleben	6700014216
Aufwärts-Abwärts-Optimierung	6673258076
Varianten entfernen	6663627142
Varianten hinzufügen	6663627142
Kosten ohne unused-content	6606670570

Als nächstes wurden die Rechenzeiten für ein viel größeres Problem betrachtet: 850 Bestellungen, 45 Moduln. Es ergaben sich folgende Rechenzeiten (Achtung: anderer κ-Wert!):

[20] Abhängigkeiten zwischen den Moduln spielen für die Rechenzeiten keine große Rolle. Alle angegebenen Rechenzeiten sind mit Fehlern behaftet und sollen nur die Größenordnung wiederspiegeln!

[21] Die gewählten p-Werte sind dabei willkürlich.

[22] Die Laufzeiten ändern sich durch Permutationen, solange Schnellkleben zuerst ausgeführt wurde, praktisch nicht.

Heuristik	Laufzeit	sonstige Parameter
Schnellklebe Heuristik	2 min	$\kappa = 3$, $goal = 50$
κ-gepuffertes Kleben	0 min	$\kappa = 3$, $[48, 52]$
Varianten entfernen	3 min	$\kappa = 3$, $[48, 52]$
Varianten hinzufügen	50 min	$\kappa = 3$, $[48, 52]$
Aufwärts-Abwärts-Optimierung	300 min	$\kappa = 3$, $[48, 52]$

Die gefundenen „Optimallösungen“ ändern sich wie folgt:

Heuristik	bestes Ergebnis $p = 50$
Schnellklebe Heuristik	17889742790
κ-gepuffertes Kleben	17889742790
Varianten entfernen	17876805148
Varianten hinzufügen	17871880303
Aufwärts-Abwärts-Optimierung	17871880303
Kosten ohne unused-content	17285264372

Für dieses Beispiel wurden ca. 600 MB Arbeitsspeicher benötigt.

A.10 Ausblick

A.10.1 Geschwindigkeitsoptimierungen

Die Aufteilung der Bestellliste anhand von Ausschlüssen dürfte wohl als einziges noch nennenswerte Vorteile bieten. Allerdings ist dies eine Aufgabe die vor der Optimierung steht. Falls der Pralinenfabrikant aus dem Vorwort auch noch Backwaren produziert, sollte er die Bestelllisten von Anfang an trennen, und nicht beide gleichzeitig ausrechnen lassen.

A.10.2 Sonderfälle

Die Aufnahme weiterer Sonderfälle in das Programm ist durch die objektorientierte und modularisierte Programmierung relativ leicht möglich. Die eingebauten Konfigurationsmöglichkeiten sind weitreichend. Leider sind widersprüchliche Eingangsdaten durch den Benutzer dabei nicht immer offensichtlich. Im Programm sind zwar Mechanismen eingebaut, um Widersprüche zu erkennen, diese benötigen jedoch viel Rechenzeit und sind standardmäßig **abgeschaltet**.

Mit dem Schalter ***CG-DEBUG*** in der Datei *global-include-file.h* können entsprechende Mechanismen aktiviert werden.

Bei Sonderfällen, die mit den vorhandenen Methoden nicht behandelt werden können, sind die Objekte **TModulnList** und **TModulnCombination** (und hier insbesondere die Methoden `combinable` und `combine`) die ersten Kandidaten für Änderungen. In der Regel dürfte es selten nötig sein, die eigentlichen Algorithmen bzw. Heuristiken anzupassen.

A.10.3 Datenbankanbindung

Eine Anbindung des Programms an eine Datenbank ist möglich. Ein an die jeweilige Datenbank angepasster Eingabe-Filter müsste die Daten in einem der

Zwischenformate speichern. Soll das Ergebnis auch in die Datenbank eingetragen werden, so muss ebenfalls ein passender Ausgabefilter geschrieben werden. Da die Schnittstellen einfach gehalten (und dokumentiert) sind, dürfte dies kein Problem darstellen.

Eine Steuerung des Programms über das WWW könnte z.B. durch Verwendung von Apache, CGI und PHP für die TCP/IP-Kommunikation und ggf. durch Anbindung an eine mySQL/postgres-Datenbank kostengünstig realisiert werden. Auch stehen bei einer solchen Lösung diverse Authentikationsmechanismen zur Herstellung der Datensicherheit zur Verfügung. Unter Linux wäre somit, ohne den Kauf lizenzpflichtiger Software, ein über WWW-Browser von beliebigen Clients gesteuerter, datenbankbasierter Compute-Server kostengünstig realisierbar.

A.10.4 Ausbaumöglichkeiten

Es ist zur Zeit nicht möglich, während der Optimierung neue Bestellungen in den Produktionsplan aufzunehmen und alte zu entfernen. Dies ist ein rein technisches Problem (Datenstrukturen anpassen). Es fehlen lediglich Hilfsprogramme bzw. zusätzliche Methoden um dies zu ermöglichen.

Ist eines der Ziele, dass sich an der Optimallösung durch diese Änderung der Daten möglichst wenig ändert (Abwägung: Umstellungskosten bei der Produktion gegen Gewinne durch anderen Produktionsplan), so wäre numerisch zu testen, ob die bestellvariantenorientierten Heuristiken nicht bereits diese gewünschte „Lokalität“ beinhalten. Ansonsten wären ggf. neue Varianten mit zusätzlichen Kosten zu belegen.

Eine automatische Überprüfung der Eingangsdaten auf Konsistenz wäre ebenfalls denkbar. Da das Format der Eingangsdaten aber für den Bediener nicht handlich ist, wird in der Praxis die Eingabe über andere Programme laufen. Daher ist es vermutlich besser dieser Oberfläche auch die Konsistenzprüfung zu überlassen.

Literaturverzeichnis

[1] Beisel, Mendel: *Optimierungsmethoden des Operations Research*, Band 1, Braunschweig 1987

[2] D. Erlenkotter: *A Dual-Based Procedure for uncapacitated Facility Location*, Operations Research, 26(6): S. 992-1009, 1978

[3] Hanjoul, Peeters: *A Comparison of Two Dual-Based Procedures for Solving the p-Median Problem.* European Journal of Operational Research, 20:387-396, 1985

[4] Christofides, Nicos: *Combinatorial optimization*, The Pitman Press, Bath 1977

[5] W. Domschke: *Logistik*, 2. Auflage, München 1985

[6] K. Neumann: *Operations Research Verfahren*, Band 2, Wien 1977

[7] Gessner, Wacker: *Dynamische Optimierung*, München 1972

[8] Cornuéjols, Nemhauser, Wolsey: *The uncapacitated facility location problem.* Aus P. Mirchandani, R. Francis: *Discrete Location Theory*, Seiten 119-171. New York 1997

[9] Guha, Khuller: *Greedy strikes back: Improved facility location algorithms.* Aus *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, Seiten 649-657. San Francisco 1998

[10] Fabiàn A. Chudak: *Improved Approximation Algorithms for Uncapaciated Facility Location.* Aus R. E. Bixby, E. A. Boyd, R. Z. Ríos-Mercado: *Integer Programming and Combinatorial Optimization*, volume 1412 of *Lecture Notes in Computer Science*, Seiten 180-194. Berlin 1998

[11] Mirchandani, Fancis: *Discrete location theory*, New York 1990

Index