Diplom-Arbeit

Untersuchung algorithmischer Methoden zur Partitionierung hybrider Systeme

Frank Eschmann

1. August 1999

Johann Wolfgang Goethe-Universität
Frankfurt am Main

Fachbereich Informatik
Lehrstuhl für Technische Informatik
Prof. Dr. K. Waldschmidt

Erklärung

Hiermit versichere ich, daß ich diese Diplomarbeit selbständig angefertigt und keine anderen, als die in der Arbeit angegebenen Hilfsmittel und Quellen verwendet habe.

Maintal, den 29. Juli 1999

Frank Eschmann

Inhaltsverzeichnis

Kapitel 1

Einleitung

Für den Entwurf digitaler Systeme sind im Laufe der Jahre viele Werkzeuge entstanden, die den Designer während des Entwurfsprozesses unterstützen und Teile des Entwurfs, wie z. B. die Synthese, sogar automatisieren. Auf diese Weise konnte der Entwurfsprozeß für digitale Systeme stark beschleunigt werden.

Wenn ein System jedoch mit seiner analogen Umwelt kommuniziert, wie es z. B. bei eingebetteten Systemen (embedded systems) der Fall ist, wird eine sogenannte *Einbettungshardware* notwendig, welche für die Signalvor- und -nachverarbeitung sorgt (siehe Abbildung 1.1).

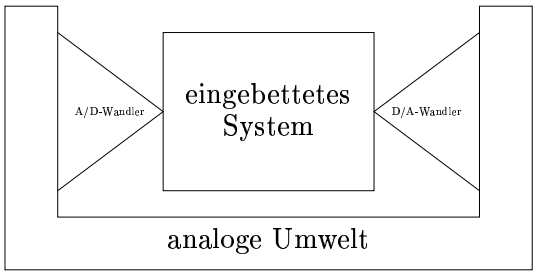
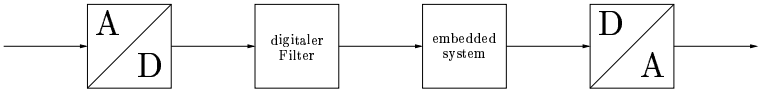


Abbildung 1.1: Eingebettetes System

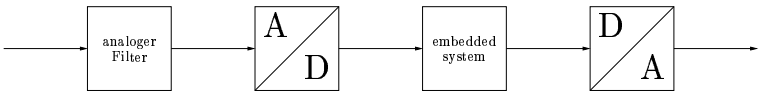
Da die Einbettungshardware nicht aus digitalen Komponenten besteht, muß sie separat von der eigentlichen Schaltung entworfen werden. Daher besteht die Einbettungshardware im allgemeinen lediglich aus A/D- und D/A-Wandlern.

Jedoch könnten viele digital realisierte Funktionen auch analog implementiert werden. Zum Beispiel kann ein Filter in Analogtechnik durch ein einfaches RC-Glied realisiert werden, während die digitale Implementierung eine aufwendige Logik, oder einen Mikrocontroller benötigt. Wenn der Filter also zur Signalvorverarbeitung benötigt wird, ist es unter Umständen sinnvoller, ihn in Analogtechnik vor dem A/D-Wandler zu implementieren

(Abbildung 1.2(b)), anstatt einen digitalen Filter hinter dem A/D-Wandler zu benutzen (Abbildung 1.2(a)). Dadurch können eventuell sogar geringere Anforderungen an den A/D-Wandler gestellt werden, weil er dann nur noch eine geringere Bandbreite zu verarbeiten hat.



(a) digitaler Filter hinter A/D-Wandler



(b) analoger Filter vor A/D-Wandler

Abbildung 1.2: Varianten zur Signalvorverarbeitung

Um solche Freiheitsgrade ausnutzen zu können, ist es sinnvoll, den Entwurf analog/digitaler Systeme (hybrider Systeme) innerhalb einer Methodik zusammenzufassen. Dies hat die Vorteile, daß eine Untersuchung des gesamten zur Verfügung stehenden Entwurfsraums möglich wird, und etwaige Nebenbedingungen (verfügbare Chipfläche, maximale Verzögerungszeit, ...) gemeinsam verwaltet werden können.

Eine solche Top-Down Entwurfsmethodik für hybride Systeme wird in [16] vorgeschlagen. Die Spezifikation wird hierbei auf einen hybriden Datenflußgraphen (HDFG) abgebildet und durch ein Blockschaltbild repräsentiert (siehe hierfür z. B. [17]). Aufgrund der speziellen Eigenschaften des HDFG ist es möglich, die jeweiligen Zugehörigkeiten zu einer Zeitmodelldomäne innerhalb der einzelnen Blöcke festzuhalten.

Um für die einzelnen Blöcke verschiedene Implementierungen in unterschiedlichen Zeitmodelldomänen zur Verfügung stellen zu können, wird in [19] das Modell des Systemgraphen vorgestellt. Mit diesem Modell wird es möglich, verschiedene Implementierungen bezüglich ihrer Eigenschaften (Flächenbedarf, Signallaufzeit, ...) zu bewerten und miteinander zu vergleichen.

Das Ziel dieser Diplom-Arbeit ist, zu untersuchen, ob und welche Optimierungsverfahren anwendbar sind, um das sich aus dem Systemgraphen-Modell ergebende Optimierungsproblem zu lösen. Da das Modell viele Aspekte der Realität aufgrund ihrer Komplexität nicht berücksichtigen kann, ist die Optimalität einer Implementierung natürlich auf das Modell zu beziehen und nicht auf die Realität.

Das bedeutet zwar, daß ein Optimierungsergebnis nicht zwangsläufig die optimale reale Implementierung ergibt, aber die systematische Untersuchung der Implementierungsalternativen wird im allgemeinen ein besseres Ergebnis erzielen als der intuitive manuelle Entwurf.

Die vorliegende Arbeit besteht aus einem theoretischen und einem praktischen Teil. In Kapitel 2 wird das in [19] vorgeschlagene Modell für hybride Systeme vorgestellt, erweitert und diskutiert.

Anschließend wird die Komplexität des Optimierungsproblems aufgezeigt, indem die $\mathcal{NP}$-Vollständigkeit (Kapitel 3) und die Nicht-Approximierbarkeit (Kapitel 4) bewiesen werden. Aufgrund dieser Ergebnisse steht fest, daß es weder einen effizienten (d. h. polynomiellen) Optimierungsalgorithmus für das Problem gibt, noch einen polynomiellen Approximationsalgorithmus mit konstanter Approximationsgarantie — sofern $\mathcal{P} \neq \mathcal{NP}$. Schließlich wird in Kapitel 5 gezeigt, daß es sich bei dem in [19] vorgeschlagenen Modell für hybride Systeme offensichtlich um ein völlig neues graphentheoretisches Problem handelt, für das bisher noch keine Lösungsverfahren untersucht worden sind.

Nach diesen theoretischen Ergebnissen werden im praktischen Teil verschiedene Optimierungsverfahren hinsichtlich ihrer Eignung betrachtet und an das vorliegende Problem angepaßt. Nach einem einleitenden Kapitel werden in den Kapiteln 7, 8 und 9 exakte Optimierungsverfahren (Lineares Programmieren, Branch & Bound und Dynamisches Programmieren) vorgestellt. Alle drei Verfahren benötigen zur Optimierung allgemeiner Systeme exponentielle Zeit. In Kapitel 9 wird jedoch gezeigt, daß sich spezielle Klassen von hybriden Systemen mittels dynamischer Programmierung in polynomieller Zeit exakt optimieren lassen.

Anschließend werden in den Kapiteln 10–13 die Meta-Heuristiken Simulated Annealing, Tabu Search und Genetische Algorithmen an das Problem angepaßt und Greedy-Algorithmen zum Finden einer „guten“ Implementierung vorgestellt.

Alle diese Verfahren sind Einzelziel-Optimierungsalgorithmen, sie vergleichen die möglichen Implementierungen also nur bezüglich eines Optimierungskriteriums (z. B. Flächenbedarf *oder* Verzögerungszeit). Häufig wird jedoch ein Kompromiß zwischen mehreren Optimierungszielen benötigt.

Zum Schluß dieser Arbeit werden deshalb Methoden vorgestellt, die einen geeigneten Kompromiß zwischen den verschiedenen Optimierungszielen suchen und entsprechende System-Implementierungen liefern.

Eine abschließende Zusammenfassung der Arbeit findet sich in Kapitel 15. Hier werden auch einige Hinweise zu vorhandener Optimierungs-Software gegeben.

Kapitel 2

Ein Modell für hybride Systeme

2.1 Grundlegende Begriffe

Der Begriff des *Systems* ist je nach Betrachtungsweise (z. B. Wirtschaft, Recht, Politik, Philosophie) mit sehr unterschiedlichen Bedeutungen belegt. Im Bereich der Elektrotechnik wird ein System folgendermaßen definiert:

Definition 2.1. Ein *System* ist ein an der Wirklichkeit orientiertes mathematisches Modell einer (meist technischen) komplexen Anordnung, das zur Beschreibung des Übertragungsverhaltens geeignet ist, also eine mathematisch eindeutige Zuordnung eines Ausgangssignales zu einem Eingangssignal gibt [28].

Das Modell dient auch zur Abstraktion der Wirklichkeit, indem man unwichtigere Eigenschaften der Realität nicht in das Modell mit einbindet, um die Komplexität möglichst klein zu halten.

Definition 2.2. Unter einer *Spezifikation* versteht man die Beschreibung der Funktionalität (d. h. das Verhalten) eines Systems [17].

Die Spezifikation beschreibt also, wie sich das System verhalten soll. Darunter gehören zum Beispiel

- Festlegung der Ein-/Ausgangs-Signale und deren zeitliche Relationen
- Leistungsdaten des Systems und der Ein-/Ausgangs-Signale.

Allerdings beschreibt die Spezifikation insbesondere nicht, wie das System aufgebaut ist [4].

Bei der Spezifikation von Systemen kann man je nach Problemstellung unterschiedliche Modelle benutzen. Zum Beispiel wird man die Übertragungsfunktion einer analogen Schaltung mit Differentialgleichungen beschreiben, während dieses Modell digitale Schaltungen nicht beschreiben kann, weil sie ein nicht-stetiges Übertragungsverhalten haben. Stattdessen wird man eher einen endlichen Automaten o. ä. als Modell zur Spezifikation benutzen.

Häufig ist ein System aber so komplex, daß es sich nicht mit Hilfe eines einzigen Modells spezifizieren läßt. Dann muß das System in Teilsysteme unterschiedlicher Modelle unterteilt werden. Diese werden anschließend separat spezifiziert, optimiert und implementiert. Solch ein System wird *heterogenes System* genannt.

Diese Methode liefert jedoch i. a. kein optimales Ergebnis, da man das System nicht als Ganzes betrachtet, sondern in Teilprobleme gleicher Modellklassen zerlegt. Wenn man die verschiedenen Modelle miteinander koppelt und Teilsysteme in verschiedenen Modellen spezifiziert, gemeinsam optimiert und implementiert, wird man mit besseren Ergebnissen rechnen können. Solche Systeme nennt man *hybride Systeme*.

2.2 Zeitmodelldomänen

Hybride Systeme lassen sich in drei *Zeitmodelldomänen* unterteilen [19]:

- *zeitkontinuierliche Domäne* (DESS)
- *zeitdiskrete Domäne* (DTSS)
- *ereignisdiskrete Domäne* (DEVS)

Die Spezifikation eines zeitkontinuierlichen Systemteils läßt sich durch Differentialgleichungen beschreiben (DESS, Differential Equation Specified System). Dies ist ein geeignetes Modell, um analoge Komponenten zu spezifizieren.

In einem zeitdiskreten Systemteil treten Signaländerungen an Ein- und Ausgängen ausschließlich zu diskreten, äquidistanten Zeitpunkten auf (DTSS, Discrete Time Specified System). Dies ist zum Beispiel bei SC- (Switched Capacitors) und Abtast-Systemen der Fall.

In einem ereignisdiskreten System treten Signaländerungen am Eingang zu beliebigen Zeitpunkten auf, werden jedoch nur zu diskreten, ereignisabhängigen Zeitpunkten (z. B. bei Überschreitung eines Schwellwertes am Eingang) verarbeitet (DEVS, Discrete Event Specified System). Die Ausgangssignale ändern sich entsprechend auch nur zu diesen Zeitpunkten. In diese Systemklasse fallen z. B. Mealy- oder Moore-Automaten.

Die einzelnen Modelle zur Spezifikation des Systems enthalten zwar zwangsläufig Strukturinformationen (und insbesondere die Zeitmodelldomäne des Modells), diese legen aber noch nicht die endgültige Zeitmodelldomäne der Teilsysteme fest. Diese initiale Unterteilung entspricht dem heterogenen System und wird i. a. suboptimal sein. Aufgrund der „überschüssigen" Informationen spricht man von einer *Überspezifikation*.

2.3 Partitionierung hybrider Systeme

Aufgabe des Schaltungsentwurfs ist es, eine Systemspezifikation in eine überprüfbare, korrekte Implementierung umzusetzen [11]. Dabei wird aus der Systemspezifikation ein Blockschaltbild der beteiligten Systemkomponenten erzeugt. Dies entspricht im Y-Diagramm nach Gajski (siehe hierfür z. B. [11, 4]) dem Übergang von der System-Ebene der Verhaltensdomäne zur Schaltungs-Ebene der Strukturdomäne (Abbildung 2.1, Übergänge ① und ②). Durch den Übergang von der Verhaltensdomäne zur Strukturdomäne ① wird für jeden Block auch die Zeitmodelldomäne festgesetzt.

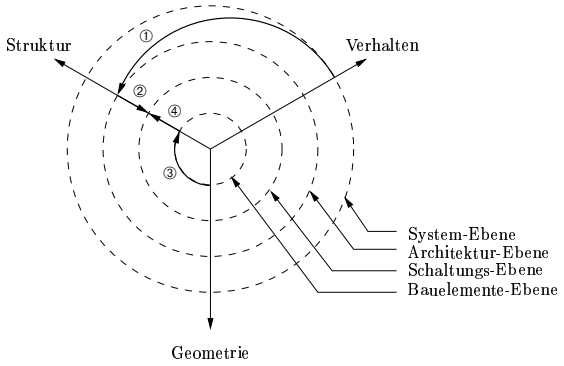


Abbildung 2.1: Einordnung des Entwurfs hybrider Systeme im Y-Diagramm

Ziel ist es nun, den einzelnen Systemkomponenten (auch Blöcke genannt) Implementierungen zuzuordnen, die mit entsprechenden Werkzeugen später weiterbearbeitet werden können. Diese zugeordneten Implementierungen der Blöcke sind jedoch keineswegs eindeutig festgelegt, sondern es wird im allgemeinen verschiedene Implementierungsmöglichkeiten, auch in unterschiedlichen Zeitmodelldomänen, geben.

Es ist aber andererseits auch nicht gesagt, daß alle möglichen Implementierungen genutzt werden dürfen. Dies kann z. B. firmenpolitische Gründe haben (Bauteil nicht lieferbar, keine Verträge mit dem Hersteller, zu teuer, ...).

Aus den verschiedenen Implementierungen der Blöcke lassen sich nun wiederum die entsprechenden *nichtfunktionalen Eigenschaften* extrahieren. Das

entspricht im Y-Diagramm dem Übergang von der Bauelemente-Ebene der Geometriedomäne zur Schaltungs-Ebene der Strukturdomäne (siehe Abbildung 2.1, Übergänge ③ und ④).

Nichtfunktionale Eigenschaften sind zum Beispiel:

- Flächenbedarf
- Leistungsverbrauch
- Geschwindigkeit / Verzögerungszeit
- Fertigungskosten
- Chipausbeute
- Genauigkeit

Die Auswahl der Implementierungen aller Systemkomponenten findet also sowohl in der Systemebene als auch in der Schaltungsebene statt. Und zwar wird in der Systemebene (während des Schrittes ①) die Zeitmodelldomäne der Systemkomponenten ausgewählt, während die Auswahl der geeigneten Implementierungen der Blöcke (für die gewählten Zeitmodelldomänen) in der Schaltungsebene (an dem Schnittpunkt der Schritte ② und ④) stattfindet. Um die größtmöglichen Freiheitsgrade bei der Wahl der Implementierung eines Blocks zu erreichen, werden beide Auswahlschritte kombiniert. Nur auf diese Weise ist es möglich, ein homogenes Modell zu erhalten.

Definition 2.3. Eine *Implementierungsmöglichkeit* ist die Kombination von Zugehörigkeit zu einer Zeitmodelldomäne und einer Implementierungsform innerhalb dieser Zeitmodelldomäne [19].

Unter einer Implementierungsmöglichkeit versteht man also eine mögliche, d. h. erlaubte Implementierung eines Blocks, unabhängig von der Zeitmodelldomäne.

Wenn man das System in unterschiedlichen Zeitmodelldomänen implementiert, benötigt man an den Schnittstellen noch zusätzliche Wandler, die das Ausgangssignal eines Blocks in das kompatible Eingangssignal des Nachfolge-Blocks transformieren. Diese Wandler besitzen natürlich ebenfalls nichtfunktionale Eigenschaften, vergrößern also Flächenbedarf, Laufzeit, etc. des Systems.

Um die Gesamt-Kosten (bzgl. einer Kostenfunktion aus den nichtfunktionalen Eigenschaften) des Systems möglichst niedrig zu halten, ist die Einteilung (d. h. die Festlegung) der Spezifikation in die verschiedenen Zeitmodelldomänen ein wichtiger Punkt im Systementwurf. Diese Einteilung bezeichnet man als *Partitionierung*.

Definition 2.4. Eine *Partition* ist ein maximaler, zusammenhängender Bereich des Systems, der in nur einer Zeitmodelldomäne liegt.

2.4 Darstellung als Graph

Ein hybrides System läßt sich formal durch einen gerichteten Graphen beschreiben.

Bemerkung 2.1. Sei S ein durch ein Blockschaltbild dargestelltes hybrides System. Dann existiert ein gerichteter Graph $G = (V, E)$, dessen Knoten $b \in V$ die jeweiligen Blöcke des hybriden Systems darstellen und die Kanten $(b, b') \in E$ den gerichteten Signalfluß zwischen den Blöcken b und b' repräsentieren.

Nun müssen die Knoten mit den möglichen Implementierungen verknüpft werden.

Definition 2.5. $D := \{\texttt{DESS}, \texttt{DTSS}, \texttt{DEVS}\}$ ist die Menge der Zeitdomänen.

Definition 2.6. L_k bezeichnet die Menge der möglichen Implementierungen von Block k.

Definition 2.7. Sei $l \in L_k$. Dann definiert der *allgemeine Block*

$$B_{k,l} := ((p_1^{k,l}, \ldots, p_n^{k,l}), d^{k,l}, d_{in}^{k,l}, d_{out}^{k,l}, t^{k,l})$$

die Implementierung l von Block k. Dabei sind

- $p_1^{k,l}, \ldots, p_n^{k,l}$ die nichtfunktionalen Eigenschaften der Implementierung (z. B. Chipfläche, Leistungsverbrauch, Verzögerungszeit, Genauigkeit, ...). Die Reihenfolge $1, \ldots, n$ dieser Eigenschaften ist systemweit festgelegt.
- $d^{k,l}, d_{in}^{k,l}, d_{out}^{k,l} \in D$ die Zeitmodelldomäne der Implementierung l von Block k ($d^{k,l}$), die benötigte Zeitmodelldomäne der Eingangssignale ($d_{in}^{k,l}$) und der Ausgangssignale ($d_{out}^{k,l}$).
- $t^{k,l}$ ein Zeiger auf ein (Text-)Objekt, das die Implementierung l des Blocks k eindeutig beschreibt.

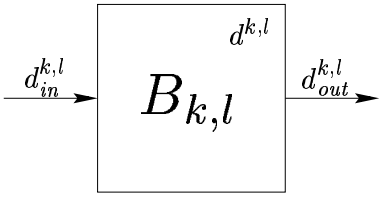


Abbildung 2.2: Allgemeiner Block k mit Implementierung l

Im allgemeinen wird zwar $d^{k,l} = d_{in}^{k,l} = d_{out}^{k,l}$ gelten, das ist aber nicht notwendig und für die Optimierungsalgorithmen auch unerheblich.

In $t^{k,l}$ befindet sich ein Verweis auf die exakte Schaltungsimplementierung der Implementierung l von Block k. Die Implementierung selbst wird bei der Optimierung nicht benötigt. Daher muß die Datenstruktur hier nicht näher betrachtet werden.

Für jeden Block k gibt es also $|L_k|$ verschiedene Implementierungen. Wenn man diese zu einer Menge B_k aller Implementierungsmöglichkeiten von Block k zusammenfaßt, liegt folgende Definition des Dimensionsbegriffs nahe:

Definition 2.8. $\dim B_k := |L_k|$

2.5 Ein-/Ausgabe-Blöcke

Jedes nicht-abgeschlossene System muß mit seiner Umgebung in Kontakt treten, also Eingabe-Signale empfangen und Ausgabe-Signale abgeben können. Würde man die Ein-/Ausgabe-Signale im Graphen nur durch Kanten realisieren, würden die Ein- und Ausgabe-Kanten an einem Ende keinen Knoten besitzen. Das Blockschaltbild wäre also nicht mehr durch einen Graphen darstellbar.

Daher werden Eingabe- bzw. Ausgabeblöcke in den Graphen eingefügt, die die Schnittstelle zur Umwelt symbolisieren.

Definition 2.9. Ein Ein-/Ausgabe-Block B_k ist ein Block, der

- keine Funktionalität,
- nur Ausgangskanten (gilt nur für Eingabe-Blöcke),
- genau eine Eingangskante (gilt nur für Ausgabe-Blöcke) und
- nur eine Implementierungsmöglichkeit

besitzt.

Die Einschränkung auf eine Implementierungsmöglichkeit garantiert, daß ein Ein-/Ausgabe-Block nur einer einzigen Zeitmodelldomäne angehören kann. Dadurch ergeben die Ein-/Ausgabe-Blöcke eine definierte Schnittstelle zur Umwelt des Systems.

Zur Unterscheidung der Ein-/Ausgabe-Blöcke von den allgemeinen Blöcken werden sie durch Kreise symbolisiert (siehe Abbildung 2.3).

Da die Ein-/Ausgabe-Blöcke einen Spezialfall der Blöcke darstellen, bezeichnet im folgenden, sofern nicht gesondert erwähnt, der Begriff Block ebenfalls die Ein-/Ausgabe-Blöcke.

(a) Eingabe-Block (b) Ausgabe-Block

Abbildung 2.3: Ein-/Ausgabe-Block

2.6 Wandlerblöcke

Der Signalfluß zwischen zwei Blöcken $i, j \in V$ wird durch die Kante $(i, j) \in E$ repräsentiert. Je nach Implementierung $l_i \in L_i$, $l_j \in L_j$ der beiden Blöcke kann der Signaltyp des Ausgangs vom Quellblock B_{i,l_i} inkompatibel zum Signaltyp des Eingangs vom Zielblock B_{j,l_j} sein. Formal

$$d_{out}^{i,l_i} \neq d_{in}^{j,l_j}.$$

Deshalb ist es notwendig, in solchen Fällen geeignete Wandler zwischen den beiden Blöcken i und j einzufügen. Da die Wandler ebenso wie normale Blöcke ein Subsystem darstellen, sind sie auch bewertbar bezüglich ihrer nichtfunktionalen Eigenschaften. Somit verändert die Existenz der Wandler die Eigenschaften des Gesamtsystems. Deshalb ist die Verschiebung der Partitionsgrenzen und die Minimierung der Wandler ein wichtiges Kriterium zur Optimierung.

Der Wandlerblock wird genauso wie der normale Block definiert.

Definition 2.10. Sei $(i, j) \in E$ und $l \in L_{(i,j)}$. Dann bezeichnet $B_{(i,j),l}$ den *allgemeinen Wandlerblock* zwischen den Blöcken i und j mit der Implementierung l.

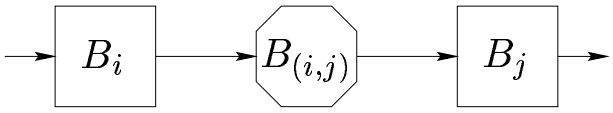


Abbildung 2.4: Wandler zwischen Block i und j

Da ein Wandler selbst keiner Zeitmodelldomäne zugeordnet werden kann, wird die Zeitmodelldomäne des Wandlers auf eine virtuelle Wandler-Domäne

$$d^{(i,j),l} := \texttt{CONV} \quad \forall l \in L_{(i,j)}$$

gesetzt.

$d_{in}^{(i,j),l}$ und $d_{out}^{(i,j),l}$ beschreiben weiterhin die Zeitmodelldomänen der Eingangs- bzw. Ausgangs-Signale und sind Elemente aus D.

Somit kann man die Wandlerblöcke als speziellen Block

$$B_{(i,j),l} := ((p_1^{(i,j),l}, \ldots, p_n^{(i,j),l}), \texttt{CONV}, d_{in}^{(i,j),l}, d_{out}^{(i,j),l}, t^{(i,j),l})$$

auffassen.

Da ein Wandlerblock nur einen Eingang und einen Ausgang besitzt, kann man ihn mit der Kante des Graphen identifizieren, deren beide Knoten (bzw. Blöcke) er verbindet.

Um nicht zwischen Kanten mit und ohne Wandlerblock unterscheiden zu müssen, wird generell zwischen zwei durch einen Signalfluß verbundenen Blöcken i und j ein Wandlerblock (i,j) eingefügt. Außerdem wird damit die Gesamtzahl der Blöcke konstant gehalten, was die Software-Implementierung des Modells vereinfacht. Das Einfügen der zusätzlichen Wandlerblöcke ist unkritisch, da man für die Wandler, deren Ein- und Ausgang der gleichen Zeitmodelldomäne angehören, „kostenneutrale“ Implementierungen zur Verfügung stellen kann.

$$B_{(i,j),\texttt{DTSS}} := ((\texttt{NULL}, \ldots, \texttt{NULL}), \texttt{CONV}, \texttt{DTSS}, \texttt{DTSS}, \texttt{NULL})$$
$$B_{(i,j),\texttt{DESS}} := ((\texttt{NULL}, \ldots, \texttt{NULL}), \texttt{CONV}, \texttt{DESS}, \texttt{DESS}, \texttt{NULL})$$
$$B_{(i,j),\texttt{DEVS}} := ((\texttt{NULL}, \ldots, \texttt{NULL}), \texttt{CONV}, \texttt{DEVS}, \texttt{DEVS}, \texttt{NULL})$$

Dabei stellt NULL den jeweils entsprechenden Wert dar, so daß sich der Wandlerblock bei der System-Bewertung neutral verhält. Im allgemeinen wird NULL auf 0 (bzw. bedeutungsäquivalent) gesetzt.

Um einen Block deutlicher von einem Wandlerblock zu unterscheiden, wird er häufig auch *Funktionsblock* genannt. Unter diesem Begriff werden auch die Ein-/Ausgabe-Blöcke verstanden, da diese lediglich ausgezeichnete Blökke sind.

Definition 2.11. Der Vektor aller Mengen B_k ($k \in V \cup E$) wird mit $\mathfrak{B}$ bezeichnet.

Definition 2.12. Das Tupel $\mathfrak{S} := (G, \mathfrak{B})$ wird als *Systemgraph* bezeichnet.

2.7 Bewertbare Eigenschaften

Wie in Kapitel 2.3 bereits erwähnt, kann man aus den verschiedenen Implementierungsmöglichkeiten die nichtfunktionalen Eigenschaften, wie z. B. Fläche oder Leistungsverbrauch, bestimmen. Die nichtfunktionalen Eigenschaften werden auch als *Kosten* bezeichnet.

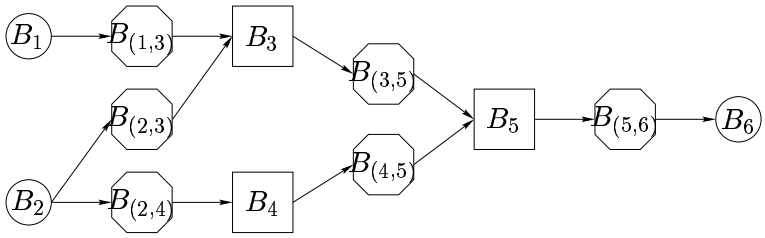


Abbildung 2.5: Beispiel für einen Systemgraphen

Die Eigenschaften, bzgl. denen optimiert werden soll, sind aus den Implementierungsmöglichkeiten aller Blöcke und Wandler zu extrahieren und dem entsprechenden Vektor $(p_1, \ldots, p_n)$ systemweit in eindeutiger Reihenfolge zuzuordnen. Daher kommen für die Bewertung nur Eigenschaften in Frage, die sich aus allen Implementierungsmöglichkeiten aller Blöcke extrahieren lassen. Dies sind zum Beispiel

p_1: benötigte Chipfläche

p_2: Verzögerungszeit

p_3: Leistungsverbrauch

p_4: Genauigkeit

p_5: Fertigungskosten

p_6: Chipausbeute

Ziel ist es nun, die Implementierungen für die einzelnen Funktions- und Wandler-Blöcke so festzulegen, daß das Gesamt-System bezüglich der gewünschten Kriterien optimal ist. Dazu gehört üblicherweise auch, daß das System bestimmte Grenzwerte (z. B. maximale zur Verfügung stehende Fläche, maximal erlaubter Leistungsverbrauch) nicht überschreiten darf.

Um die Darstellung übersichtlicher zu gestalten, beschreibe $\mathfrak{l}$ den Vektor der Implementierungen aller Funktions- und Wandler-Blöcke.

Definition 2.13.

$$\mathbb{I} := L_{b_1} \times \ldots \times L_{b_{|V|}} \times L_{e_1} \times \ldots \times L_{e_{|E|}}$$
$$\mathfrak{l} := (l_{b_1}, \ldots, l_{b_{|V|}}, l_{e_1}, \ldots, l_{e_{|E|}}) \in \mathbb{I}$$

Die Bewertung des Gesamtsystems aufgrund der einzelnen Blöcke ist je nach betrachteter nichtfunktionaler Eigenschaft recht unterschiedlich. Die folgenden Berechnungsfunktionen sind nur als einfache Beispiele zu verstehen. Eine genauere Berechnung der Systemeigenschaften führt im allgemeinen zu

komplexeren Funktionen, die aber eine Optimierung deutlich aufwendiger werden lassen.

Die Chipfläche verhält sich additiv, d. h. die Gesamtchipfläche berechnet sich als Summe der Fläche aller Blöcke. Das gleiche gilt auch für den Leistungsverbrauch und die Fertigungskosten.

$$P_1(\mathfrak{l}) := \sum_{k \in V} p_1^{k,l_k} + \sum_{(i,j) \in E} p_1^{(i,j),l_{(i,j)}}$$

$$P_3(\mathfrak{l}) := \sum_{k \in V} p_3^{k,l_k} + \sum_{(i,j) \in E} p_3^{(i,j),l_{(i,j)}}$$

$$P_5(\mathfrak{l}) := \sum_{k \in V} p_5^{k,l_k} + \sum_{(i,j) \in E} p_5^{(i,j),l_{(i,j)}}$$

Die Ausbeute verhält sich hingegen multiplikativ, wenn man davon ausgeht, daß die Fehlerwahrscheinlichkeiten unabhängig voneinander sind. Daher muß für die kostenneutralen Wandlerblöcke $p_6^{(i,j),l}=1$ gelten, damit sie sich in der Kostenberechnung tatsächlich neutral verhalten.

$$P_6(\mathfrak{l}) := \prod_{k \in V} p_6^{k,l_k} \cdot \prod_{(i,j) \in E} p_6^{(i,j),l_{(i,j)}}$$

Noch aufwendiger ist die Berechnung der Gesamt-Verzögerungszeit, da diese durch den kritischen Pfad durch die einzelnen Blöcke bestimmt ist, also kein lineares Verhalten aufweist, sondern neben den einzelnen Block-Verzögerungszeiten von der Graphen-Struktur abhängt.

$$P_2(\mathfrak{l}) := \max_{\lambda \subset E \text{ Pfad}} \sum_{\substack{k \in V \\ \exists j \in V : (k,j) \in \lambda \vee (j,k) \in \lambda}} p_2^{k,l_k} + \sum_{(i,j) \in \lambda} p_2^{(i,j),l_{(i,j)}}$$

Die Genauigkeit eines Systems hängt von der System-Struktur und von der Funktionalität der einzelnen Blöcke ab. Daher läßt sich für die Genauigkeit keine allgemeine Funktion angeben.

2.8 Beurteilung des Modells

Es stellt sich die Frage, wie genau das Modell die Realität abbildet.

Diese Fragestellung läßt sich in verschiedene Aspekte unterteilen:

- Wie genau sind die Daten, die man aus der Realität bzw. Simulation erhält?

- Wie gut lassen sich die Daten auf das Modell übertragen?
- Wie stark schränkt das Modell den Entwurfsraum ein?
- Wie weit ist die optimale reale Implementierung von der optimalen Implementierung im Modell entfernt?

2.8.1 Abbildung der Spezifikation

Als erster Schritt muß die Spezifikation in Blöcke unterteilt werden, die später separat synthetisiert werden. Da diese Block-Aufteilung verwendet wird, um den Graphen zu erzeugen, sollte man bereits vor der Synthese ein besonderes Augenmerk auf die Aufteilung der Blöcke haben. Dies ist kein einfaches Problem, da zu diesem Zeitpunkt, außer der Verhaltensbeschreibung, keine weiteren Informationen vorliegen.

Neben dem Problem der unglücklich gewählten Block-Grenzen ergibt sich aber noch ein weiteres Problem bei der Abbildung der Spezifikation in die Funktionsblöcke. Enthalten die Blöcke zuviel Funktionalität, d. h. sind sie zu groß, dann besitzt die Optimierung nicht genügend Freiheitsgrade, da vielleicht eine Partitionierung innerhalb eines Blocks kostengünstiger wäre. Sind die Blöcke andererseits zu klein, dann existieren zum einen extrem viele Kanten (also Interconnect-Leitungen), die in den Gesamtkosten nicht berücksichtigt werden (siehe auch weiter unten), zum anderen wird der Graph sehr groß, was den Optimierungsaufwand drastisch erhöht. Daher ist der Abbildungsschritt sehr sorgfältig auszuführen. Eventuell ist es auch sinnvoll, diesen mehrfach auszuführen und die Optimierungsergebnisse miteinander zu vergleichen.

2.8.2 Genauigkeit der nichtfunktionalen Eigenschaften

Die Eigenschaften der Blöcke müssen aus den verschiedenen Implementierungen bestimmt werden. Dies ist jedoch keine einfache Aufgabe, und sollte zudem noch möglichst genau erfolgen, da die Werte der nichtfunktionalen Eigenschaften die Grundlage der Optimierung darstellen. Sind diese Werte ungenau, liegen die Kosten der berechneten optimalen Implementierung zwar in der Nähe der Kosten der wahren optimalen Implementierung, aber das Ergebnis kann grundsätzlich nur so gut sein, wie die extrahierten Eigenschaftswerte. Andererseits ist es für die exakte Extraktion der Eigenschaften notwendig, die Blöcke bis zu einer sehr niedrigen Ebene zu implementieren. Dies ist jedoch extrem zeitaufwendig und daher nicht praktikabel. Deshalb ist es wichtig, einen geeigneten Kompromiß zwischen Genauigkeit und Zeitaufwand zu finden.

Zusätzlich zu den Implementierungen der einzelnen Blöcke benötigt man für die exakte Extraktion der Eigenschaften noch zusätzliche Informationen, die

sich erst durch die Betrachtung der fertigen Schaltung ergeben.

Zum Beispiel läßt sich die Verzögerungszeit des clock-Signals (durch Signallaufzeit auf der Leitung) erst abschätzen, wenn die Schaltung komplett geroutet ist. Die Differenz der clock-Signal-Ankunftszeit zwischen zwei benachbarten Blöcken (clock-skew) ist aber notwendig, um die Verzögerungszeit eines Blocks in DTSS besser abschätzen zu können. Außerdem benötigt die Taktleitung eine gewisse, von der Leitungslänge abhängige Fläche. Diese kann man aber auch nicht a priori berechnen, da

1. die Schaltung noch nicht geroutet ist, und
2. Teile der Leitung von mehreren Blöcken verwendet werden könnten, also nicht jedem Block komplett aufgeschlagen werden dürfen.

Genau wie bei den Funktionsblöcken, können auch bei den Wandlerblöcken die Leitungslaufzeiten, sowie die Leitungsfläche nicht richtig abgeschätzt werden, solange die Schaltung nicht geroutet ist.

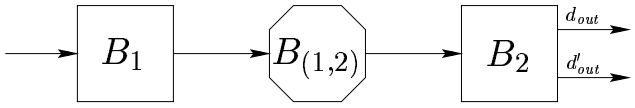
Mögliche Toleranzen der nichtfunktionalen Eigenschaften werden nicht berücksichtigt, um das System nicht zusätzlich zu verkomplizieren. Außerdem ließen sich die Toleranzen in der Kostenfunktion u. U. nicht geeignet handhaben. Stattdessen sollte man Toleranzen als ungenaue Extraktion der nichtfunktionalen Eigenschaften auffassen, denn im allgemeinen wird die Verfälschung durch die Toleranzen deutlich geringer ausfallen als durch die Extraktion.

2.8.3 Einschränkung des Entwurfsraums durch das Modell

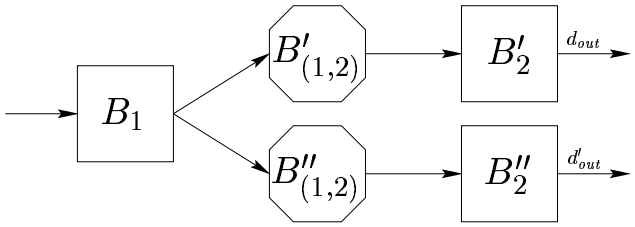
Das Modell selbst schränkt den Entwurfsraum ein. Und zwar müssen alle Eingangs- bzw. Ausgangssignale eines Blocks die gleiche Zeitmodelldomäne besitzen. Ein Block mit Ausgängen in zwei verschiedenen Zeitmodelldomänen kann nachgebildet werden, indem man ihn dupliziert — die beiden Blöcke besitzen dann jeweils nur Ausgänge in einer Zeitmodelldomäne (siehe auch Abbildung 2.6). Allerdings werden die Kosten auf diese Weise im schlechtesten Fall verdoppelt.

Blöcke mit Eingängen unterschiedlicher Zeitmodelldomänen lassen sich nicht so einfach ersetzen. Sie müssen schon beim Entwurf entsprechend spezifiziert werden, d. h. in die Implementierung des Blocks müssen bereits Wandler eingebaut werden (Abbildung 2.7). Auch dadurch erhöhen sich die Kosten unnötigerweise.

Eine weitere Einschränkung ergibt sich durch die Modellierung als Graph: Angenommen mit dem Ausgang von Block b seien zwei Blöcke b' und b'' verbunden (Abbildung 2.8(a)) , und die Blöcke b', b'' besitzen in der optimalen Implementierung gleiche Zeitmodelldomänen, aber unterschiedlich zu der von b. Somit würde ein Wandler von b zu b' und b'' benötigt werden.



(a) ursprüngliche Spezifikation



(b) modifizierte Spezifikation

Abbildung 2.6: Modellierung von Blöcken mit verschiedenen Ausgangs-Zeitmodelldomänen

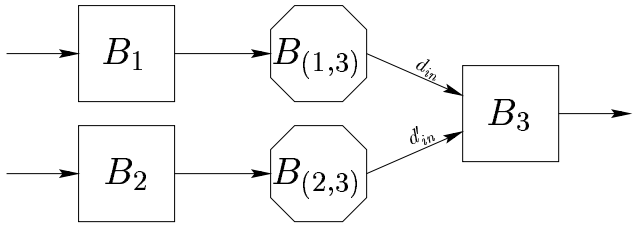
Da die Wandler in dem Modell jedoch als Kanten interpretiert werden, werden also zwei Wandlerblöcke benötigt (Abbildung 2.8(b)). Dies vergrößert natürlich wiederum die Kosten des Optimierungsergebnisses.

Auch hier ist es möglich, eine bessere Lösung zu konstruieren, indem man zwischen b und b' bzw. b'' einen Dummy-Block d ohne Kosten einfügt. Somit wird in diesem Fall ein optimales Ergebnis erzielt, indem man den Wandler zwischen b und d einfügt, anstatt zwischen d und b', sowie d und b'' (Abbildung 2.8(c)). Dieser Trick ist allerdings als Standardmethode nicht sinnvoll, da die Komplexität des Graphen dadurch deutlich erhöht wird.

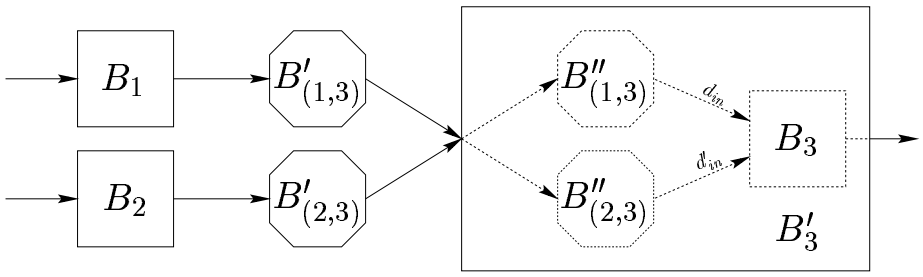
2.8.4 Gesamtbeurteilung

Das Hauptproblem ist die optimale Abbildung der Spezifikation auf die Blöcke, die aber erst auf der Schaltungsebene optimal durchführbar ist. Es ist allerdings gar nicht gewollt, auf eine so niedrige Ebene herunterzugehen — vielmehr ist es das Ziel, auf möglichst hohem Abstraktionsniveau zu arbeiten. Nur dadurch kann man die Komplexität des Optimierungsproblems und des gesamten gemischt analog/digitalen Schaltungsentwurfs in realistischen Grenzen halten.

Die Kosten für die Interconnect-Leitungen zwischen den Blöcken und Wandlern werden in dem Modell nicht berücksichtigt. Dies ist aber durchaus üblich: In den heutzutage verfügbaren Optimierungs-Werkzeugen werden ebenfalls nur die Kosten für die aktiven Bauteile optimiert und die Leitungskosten nicht berücksichtigt. Dadurch lassen sich zwar die Gesamt-



(a) ursprüngliche Spezifikation



(b) modifizierte Spezifikation

Abbildung 2.7: Modellierung von Blöcken mit verschiedenen Eingangs-Zeitmodelldomänen

kosten nicht exakt berechnen, jedoch kann man davon ausgehen, daß eine optimale Implementierung im Modell auch einer optimalen Implementierung unter Berücksichtigung der Signalleitungs-Kosten entspricht.

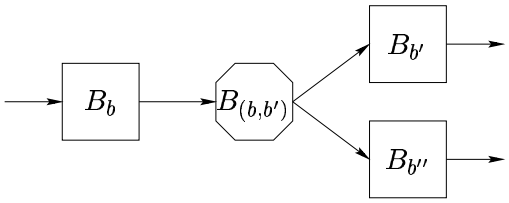
Durch die in Kapitel 2.8.3 beschriebenen Einschränkungen des Entwurfsraums wird das Optimierungsergebnis im Vergleich zum tatsächlichen Optimum nochmals verschlechtert.

Auf diese Weise kann man zwar beliebig schlechte Ergebnisse erzielen, in praktischen Anwendungen wird man im allgemeinen aber relativ realistische Ergebnisse erreichen. Dies müßte jedoch mit repräsentativen Benchmarks überprüft werden.

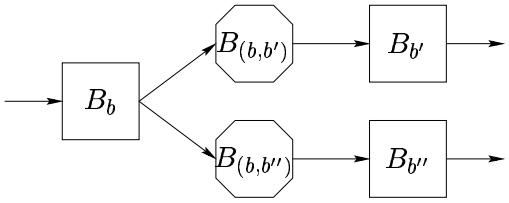
2.9 Formalisierung des Modells

Um die Komplexität des Modells einschätzen zu können, muß das Problem zuerst entsprechend formalisiert werden.

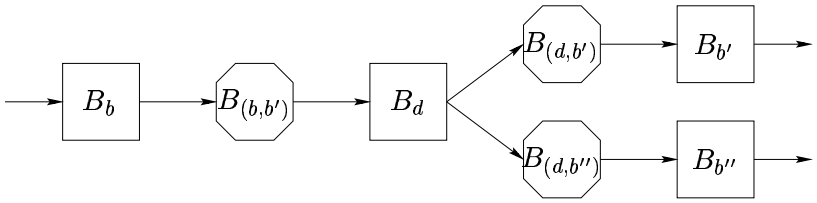
Das eben erzeugte Modell zur möglichst kostengünstigen Partitionierung hybrider Systeme, läßt sich in zwei Teilschritten folgendermaßen formalisieren:



(a) ursprüngliche Spezifikation



(b) modellierbare Spezifikation



(c) evtl. optimal implementierbare Spezifikation (d ist neutraler Funktionsblock, die Wandler (d, b') und (d, b'') sind Kopien der Wandler (b, b'), bzw. (b, b''))

Abbildung 2.8: Spezifikation mehrfach nutzbarer Wandlerblöcke

Definition 2.14.

$$PART := \Big\{ \mathfrak{S} = (G, \mathfrak{B}) \Big| \ \exists \mathfrak{l} \in \mathbb{l} : d_{out}^{v,l_v} = d_{in}^{(v,w),l_{(v,w)}} \wedge d_{out}^{(v,w),l_{(v,w)}} = d_{in}^{w,l_w} \ \forall (v,w) \in E \Big\}$$

$PART$ ist also die Menge aller Systemgraphen für die eine Implementierung existiert. Das Problem der Entscheidung, ob ein Systemgraph $\mathfrak{S}$ in $PART$ enthalten ist ($\mathfrak{S} \in PART$) oder nicht ($\mathfrak{S} \notin PART$) wird *Sprachenproblem* genannt. Mit polynomial vielen Abfragen des Sprachenproblems $PART$ läßt sich für einen Systemgraphen $\mathfrak{S} \in PART$ eine gültige Implementierung berechnen, indem man Schritt für Schritt die Implementierungsmöglichkeiten

der Blöcke einschränkt und überprüft, ob dieser neue Systemgraph noch immer zu der Sprache $PART$ gehört.

Es ist aber mit dem Sprachenproblem nicht möglich, eine nach bestimmten Gesichtspunkten optimale Implementierung für einen Systemgraphen zu erhalten. Erst das *Optimierungsproblem* gibt Auskunft, welche der möglichen Implementierungen die günstigste bezüglich einer Kostenfunktion ist.

Definition 2.15. Sei ein Systemgraph $\mathfrak{S} \in PART$ gegeben, sowie ein in Polynomialzeit berechenbares Maß

$$m : \mathbb{S} \times \mathbb{I} \to \mathbb{R}$$

($\mathbb{S}$ bezeichne die Menge aller Systemgraphen $\mathfrak{S} \in PART$ und $\mathbb{I}$ die Menge aller möglichen Implementierungen (Def. 2.13). Dann bezeichnet

$$PART_m^{\text{opt}}(\mathfrak{S}) := \min\Big\{ m(\mathfrak{S}, \mathfrak{I}) \;\Big|\; \mathfrak{I} \in \mathbb{I},\, d_{out}^{v,l_v} = d_{in}^{(v,w),l_{(v,w)}} \wedge d_{out}^{(v,w),l_{(v,w)}} = d_{in}^{w,l_w}\ \forall (v,w) \in E \Big\}$$

das *Optimierungsproblem* von $PART$.

$PART_m^{\text{opt}}(\mathfrak{S})$ ergibt also die Kosten der bezüglich der Kostenfunktion m optimalen Implementierung. Da $|\mathbb{I}|$ endlich ist, existiert das zu berechnende Minimum, sofern $\mathbb{I} \neq \emptyset$ (da dann inf = min). Falls die Implementierungskosten maximiert werden sollen, ist dies durch Invertierung von m möglich (wg. $-\max x = \min -x$). Aufgrund dieser Tatsache wird in den folgenden Kapiteln angenommen, daß die Kosten zu minimieren sind.

Wenn eine Aussage unabhängig von der Kostenfunktion m gilt, bzw. das Optimierungsproblem ohne eine spezielle Kostenfunktion bezeichnet werden soll, entfällt im folgenden das m in der Bezeichnung, und das Optimierungsproblem wird im allgemeinen nur mit $PART^{\text{opt}}$ bezeichnet.

Analog zum Sprachenproblem läßt sich auch für das Optimierungsproblem eine optimale *Implementierung* mit Hilfe von polynomiell vielen Abfragen des Optimierungsproblems errechnen, indem die Implementierungsmöglichkeiten der Blöcke systematisch eingeschränkt werden, und man überprüft, ob die minimalen Kosten für den eingeschränkten Systemgraphen zu denen des ursprünglichen Systemgraphen gleich bleiben oder sich vergrößern. Sollten die minimalen Kosten größer werden, dann kann diese Block-Implementierung nicht in der optimalen System-Implementierung enthalten sein.

Kapitel 3

$\mathcal{NP}$-Vollständigkeit

In diesem Kapitel wird gezeigt, daß *PART* $\mathcal{NP}$-vollständig ist. Das bedeutet, daß es keinen Polynomialzeit-Algorithmus gibt, der entscheidet, ob ein Systemgraph $\mathfrak{S}$ implementierbar ist oder nicht (sofern $\mathcal{P} \neq \mathcal{NP}$). Dies erscheint irrelevant, da aufgrund der Überspezifikation bekannt ist, daß eine Implementierung für $\mathfrak{S}$ existiert. Außerdem lautet die eigentliche Frage nicht, ob überhaupt eine Implementierung existiert, sondern welches die optimale Implementierung ist.

Die beiden Probleme sind aber durchaus miteinander verwandt. So ist es nicht möglich, daß $PART^{\text{opt}}$ ein leichteres Problem darstellt als *PART*, denn falls $PART^{\text{opt}}$ die Kosten einer optimalen Implementierung liefert, dann ist $\mathfrak{S}$ zwangsläufig implementierbar, also in *PART* enthalten. Wenn andererseits $PART^{\text{opt}}$ keine optimale Implementierung findet, heißt dies automatisch, daß $\mathfrak{S}$ nicht implementierbar ist, also nicht in *PART* enthalten ist.

Das bedeutet somit, daß $PART^{\text{opt}}$ ein $\mathcal{NP}$-vollständiges Optimierungsproblem sein muß, wenn *PART* $\mathcal{NP}$-vollständig ist, da es nicht leichter lösbar sein kann als *PART* (siehe dazu auch [15]).

Außerdem ist der „Umweg“ über *PART* notwendig, weil der Begriff der $\mathcal{NP}$-Vollständigkeit nur für Sprachenprobleme definiert ist. Wenn man also von einem $\mathcal{NP}$-vollständigen Optimierungsproblem spricht, bedeutet das, daß die Sprachenversion des Optimierungsproblems $\mathcal{NP}$-vollständig ist. Die Sprachenversion eines Optimierungsproblems charakterisiert aber, wie schon gesagt, die Komplexität des Optimierungsproblems.

3.1 Theoretische Grundlagen

Es gibt verschiedene Wege, die $\mathcal{NP}$-Vollständigkeit zu definieren (z. B. mittels Existenz-Quantoren, nichtdeterministischer Turing-Maschine oder Turingmaschine mit Orakel-Band). Diese spannen aber alle die gleiche Pro-

blemklasse auf, so daß man für das hier gegebene Problem die geeignetste Definition nutzen kann.

Definition 3.1. Eine Sprache L ist in $\mathcal{NP}$ enthalten ($L \in \mathcal{NP}$), wenn eine nicht-deterministische Turingmaschine in polynomieller Zeit (in Abhängigkeit zur Eingabelänge) entscheidet, ob eine Eingabe w in L enthalten ist, oder nicht.

Es sind natürlich auch in Polynomialzeit lösbare Probleme (d. h. $L \in \mathcal{P}$) in $\mathcal{NP}$ enthalten, da sich eine deterministische Turingmaschine durch eine nicht-deterministische ersetzen läßt, die den Nicht-Determinismus nicht ausnutzt. Es konnte jedoch bisher noch nicht bewiesen werden, daß $\mathcal{P} \neq \mathcal{NP}$ ist. Zumindest gilt aber $\mathcal{P} \subseteq \mathcal{NP}$.

Jedoch gibt es Sprachen in $\mathcal{NP}$, die als Repräsentanten von $\mathcal{NP}$ angesehen werden können. Dafür wird aber zuerst der Begriff der *polynomiellen Reduzierbarkeit* benötigt.

Definition 3.2. Seien L_1 und L_2 Sprachen über den Eingabe-Alphabeten Σ_1 und Σ_2. Dann heißt L_1 *polynomiell auf* L_2 *reduzierbar* ($L_1 \leq_p L_2$), wenn es eine von einer deterministischen Turingmaschine in Polynomialzeit berechenbare Funktion $f : \Sigma_1^* \to \Sigma_2^*$ gibt, so daß für alle Eingaben $w \in \Sigma_1^*$ gilt:

$$w \in L_1 \Longleftrightarrow f(w) \in L_2$$

Falls $L_1 \leq_p L_2$ gilt, kann also L_1 höchstens um einen polynomiellen Faktor schwieriger sein als L_2 (weil L_1 mit Hilfe von L_2 und einer (relativ) einfachen, turing-berechenbaren Funktion f nachgeahmt werden kann). Da die Anti-Symmetrie nicht erfüllt ist, ist die „$\leq$“-Schreibweise nur symbolisch zu verstehen und darf nicht als Ordnungsrelation aufgefaßt werden.

Definition 3.3. Eine Sprache L heißt $\mathcal{NP}$-*hart*, wenn für alle $L' \in \mathcal{NP}$ gilt: $L' \leq_p L$.

Definition 3.4. Eine Sprache L heißt $\mathcal{NP}$-*vollständig*, wenn $L \in \mathcal{NP}$ und L $\mathcal{NP}$-hart ist.

Es lassen sich also alle Sprachen aus $\mathcal{NP}$ auf $\mathcal{NP}$-harte bzw. $\mathcal{NP}$-vollständige Sprache reduzieren. Und dies gilt insbesondere auch für $\mathcal{NP}$-vollständige Sprachen.

Daher gilt folgender Satz:

Satz 3.1. *Sei L_1 $\mathcal{NP}$-vollständig und $L_2 \in \mathcal{NP}$. Dann gilt*

$$L_1 \leq_p L_2 \Longrightarrow L_2 \ \mathcal{NP}\text{-vollständig}$$

Bisher wurde noch keine Aussage getroffen, ob es überhaupt $\mathcal{NP}$-vollständige Probleme gibt. Den Existenzbeweis lieferte Cook, indem er die nichtdeterministische Turingmaschine auf eine konjunktive Normalform abbildete. Für den Beweis sei z. B. auf [15] verwiesen.

Definition 3.5.

$$\textit{KNF-SAT} := \{f : \ f \text{ ist Boolesche Formel in konjunktiver Normalform } \wedge \ f \text{ ist erfüllbar}\}$$

Satz 3.2 (Cook, 1971). *KNF-SAT ist $\mathcal{NP}$-vollständig*

Das Ziel ist es, ein $\mathcal{NP}$-vollständiges Problem auf *PART* zu reduzieren. Dies wird *3-KNF-SAT* sein. Dafür muß aber erst einmal gezeigt werden, daß *3-KNF-SAT* $\mathcal{NP}$-vollständig ist.

Definition 3.6.

$$\textit{3-KNF-SAT} := \{f : \ f \in \textit{KNF-SAT} \ \wedge \text{ jede Klausel in } f \text{ besitzt genau 3 Literale}\}$$

Satz 3.3. *3-KNF-SAT ist $\mathcal{NP}$-vollständig*

Beweis. Der Beweis erfolgt durch polynomielle Reduktion von *KNF-SAT* auf *3-KNF-SAT* (*KNF-SAT* $\leq_p$ *3-KNF-SAT*) und kann in [37] nachgelesen werden. □

3.2 $\mathcal{NP}$-Vollständigkeit von *PART*

Satz 3.4. *PART ist $\mathcal{NP}$-vollständig.*

Beweis. Wir zeigen, daß *3-KNF-SAT* $\leq_p$ *PART* — konstruieren also eine Funktion, so daß sich *3-KNF-SAT* durch *PART* ausdrücken läßt.

Die Struktur einer Booleschen Formel in *3-KNF* entspricht

$$\begin{aligned} &f = C_1 \wedge \ldots \wedge C_m \\ &C_i = c_{i_1} \vee c_{i_2} \vee c_{i_3} \quad \forall i = 1, \ldots, m \\ &c_{i_j} \in \{x_1, \neg x_1, \ldots, x_n, \neg x_n\} \quad \forall i = 1, \ldots, m \ , \ j = 1, 2, 3 \end{aligned}$$

wobei die Variablen $x_1, \ldots, x_n$ die logischen Werte 0 und 1 annehmen dürfen.

Ziel ist es nun, die Variablen $x_1, \ldots, x_n$ und die Klauseln $C_1, \ldots, C_m$ so auf Funktionsblöcke eines Systemgraphen abzubilden, daß der Systemgraph genau dann implementierbar ist, wenn eine erfüllbare Belegung für die Formel

f existiert. Daher wird für jede Variable x_i und für jede Klausel C_j ein Funktionsblock erzeugt.

Wenn eine Implementierung für den Systemgraphen existiert, sollen die Nummern der gewählten Implementierungen der Variablen-Blöcke direkt der Belegungen der Variablen entsprechen, die die Formel erfüllen. Deshalb dürfen die Variablenblöcke auch nur zwei Domänen für die Ausgänge besitzen[1]:

$$\left.\begin{aligned} B_{x_j,0} &:= (-,-,-,0,-) \\ B_{x_j,1} &:= (-,-,-,1,-) \end{aligned}\right\} \quad \forall j = 1,\ldots,n$$

Die 3 Variablen einer Klausel lassen sich auf $2^3 = 8$ verschiedene Arten belegen, aber nur $2^3 - 1 = 7$ dieser Belegungen erfüllen die Klausel. Da die Erfüllbarkeit einer Klausel als gültige Implementierung verschiedener Blöcke dargestellt werden soll, werden für jeden Klauselblock 7 verschiedene Implementierungen benötigt, die die einzelnen erfüllenden Belegungsmöglichkeiten der Klausel eindeutig nachahmen.

$$\left.\begin{aligned} B_{C_i,1} &:= (-,-,1,-,-) \\ B_{C_i,2} &:= (-,-,2,-,-) \\ B_{C_i,3} &:= (-,-,3,-,-) \\ B_{C_i,4} &:= (-,-,4,-,-) \\ B_{C_i,5} &:= (-,-,5,-,-) \\ B_{C_i,6} &:= (-,-,6,-,-) \\ B_{C_i,7} &:= (-,-,7,-,-) \end{aligned}\right\} \quad \forall i = 1,\ldots,m$$

Die achte Implementierungsmöglichkeit entspräche derjenigen Belegung, die die Klausel nicht erfüllt, und darf deshalb nicht implementierbar sein.

Nun müssen die Klausel-Blöcke mit den zu der Klausel gehörenden Variablen-Blöcken verknüpft werden. Dies geschieht mit Wandlern, die nur die Implementierungen der Variablen-Blöcke (also Belegungen der Variablen) zulassen, für die auch die entsprechenden Klauseln erfüllbar sind.

Zur besseren Anschauung sei folgendes Beispiel gegeben.

Beispiel 3.1. Sei $C = x_1 \vee \neg x_2 \vee x_3$. Dann ergeben sich für C folgende gültige Belegungen:

[1] Die nichtfunktionalen Eigenschaften und die Implementierung des Blockes sind für den Beweis unerheblich und werden deshalb mit $-$ gekennzeichnet.

x_1	x_2	x_3
1	1	0
0	0	0
0	1	1
1	0	0
1	1	1
0	0	1
1	0	1

Die Belegung $x_1 = 0, x_2 = 1, x_3 = 0$ ist die Belegung, für die $C = 0$ ist, also C nicht erfüllt.

Nun lassen sich den einzelnen gültigen Belegungen die möglichen Implementierungen der Klausel so zuordnen, daß die sich ergebende Abbildung bijektiv ist. Zum Beispiel:

x_1	x_2	x_3	Implementierung
1	1	0	1
0	0	0	2
0	1	1	3
1	0	0	4
1	1	1	5
0	0	1	6
1	0	1	7

Wenn nun für Block C Implementierung 1 gewählt wird, dann müssen zwangsläufig $x_1 = 1$, $x_2 = 1$ und $x_3 = 0$ sein; wenn Implementierung 2 gewählt wird, dann müssen $x_1 = 0$, $x_2 = 0$ und $x_3 = 0$ sein, usw. Dadurch gibt es keine Möglichkeit, die Implementierung des Klauselblocks und der zugehörigen Wandler-Blöcke so zu wählen, daß eine nicht-erfüllende Belegung von den Variablenblöcken implementiert werden kann.

Die Implementierungsmöglichkeiten der Wandler-Blöcke müssen dann der

obigen Tabelle entsprechend lauten:

$$B_{(x_1,C),1} := (-,-,1,1,-) \qquad B_{(x_2,C),1} := (-,-,1,1,-)$$
$$B_{(x_1,C),2} := (-,-,0,2,-) \qquad B_{(x_2,C),2} := (-,-,0,2,-)$$
$$B_{(x_1,C),3} := (-,-,0,3,-) \qquad B_{(x_2,C),3} := (-,-,1,3,-)$$
$$B_{(x_1,C),4} := (-,-,1,4,-) \qquad B_{(x_2,C),4} := (-,-,0,4,-)$$
$$B_{(x_1,C),5} := (-,-,1,5,-) \qquad B_{(x_2,C),5} := (-,-,1,5,-)$$
$$B_{(x_1,C),6} := (-,-,0,6,-) \qquad B_{(x_2,C),6} := (-,-,0,6,-)$$
$$B_{(x_1,C),7} := (-,-,1,7,-) \qquad B_{(x_2,C),7} := (-,-,0,7,-)$$

$$B_{(x_3,C),1} := (-,-,0,1,-)$$
$$B_{(x_3,C),2} := (-,-,0,2,-)$$
$$B_{(x_3,C),3} := (-,-,1,3,-)$$
$$B_{(x_3,C),4} := (-,-,0,4,-)$$
$$B_{(x_3,C),5} := (-,-,1,5,-)$$
$$B_{(x_3,C),6} := (-,-,1,6,-)$$
$$B_{(x_3,C),7} := (-,-,1,7,-)$$

Eine bildliche Darstellung findet sich in Abbildung 3.1. Zur besseren Übersichtlichkeit werden nur die Zeitmodell-Domänen der einzelnen Implementierungen angegeben. Weitere Informationen, wie z. B. nichtfunktionale Eigenschaften, sind für den Beweis irrelevant.

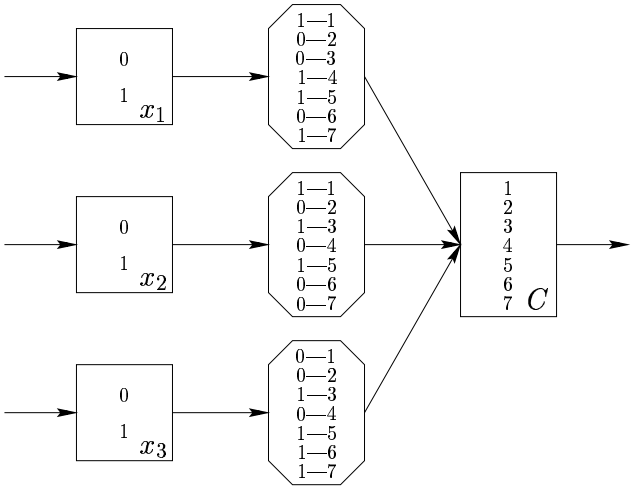


Abbildung 3.1: Systemgraph der Klausel $C = x_1 \vee \neg x_2 \vee x_3$

Um zum allgemeinen Fall zurückzukehren, müssen die Wandler-Blöcke so definiert werden, daß alle Belegungen der Variablen, die die Klauseln erfüllen, implementiert werden können. Andererseits dürfen diejenigen Variablenbelegungen, die mindestens eine Klausel nicht erfüllen, nicht implementierbar sein.

$$B_{(x_j,C_i),l} := (-,-,d_{in}^{(x_j,C_i),l},l,-) \quad \forall i = 1,\ldots,m\,, \quad \forall x_j \in \{c_{i_1}, \neg c_{i_1}, c_{i_2}, \neg c_{i_2}, c_{i_3}, \neg c_{i_3}\}$$

Zur einfacheren Definition der $d_{in}^{(x_j,C_i),l}$ wird nach x_j unterschieden.

1. $x_j = c_{i_1}$: (d. h. x_j ist 1. Literal in Klausel C_i)

$$d_{in}^{(x_j,C_i),l} := \begin{cases} 0 & \text{falls } l \in \{2,3,6\} \\ 1 & \text{falls } l \in \{1,4,5,7\} \end{cases}$$

2. $x_j = \neg c_{i_1}$: (d. h. $\neg x_j$ ist 1. Literal in Klausel C_i)

$$d_{in}^{(x_j,C_i),l} := \begin{cases} 0 & \text{falls } l \in \{1,4,5,7\} \\ 1 & \text{falls } l \in \{2,3,6\} \end{cases}$$

3. $x_j = c_{i_2}$: (d. h. x_j ist 2. Literal in Klausel C_i)

$$d_{in}^{(x_j,C_i),l} := \begin{cases} 0 & \text{falls } l \in \{1,3,5\} \\ 1 & \text{falls } l \in \{2,4,6,7\} \end{cases}$$

4. $x_j = \neg c_{i_2}$: (d. h. $\neg x_j$ ist 2. Literal in Klausel C_i)

$$d_{in}^{(x_j,C_i),l} := \begin{cases} 0 & \text{falls } l \in \{2,4,6,7\} \\ 1 & \text{falls } l \in \{1,3,5\} \end{cases}$$

5. $x_j = c_{i_3}$: (d. h. x_j ist 3. Literal in Klausel C_i)

$$d_{in}^{(x_j,C_i),l} := \begin{cases} 0 & \text{falls } l \in \{1,2,4\} \\ 1 & \text{falls } l \in \{3,5,6,7\} \end{cases}$$

6. $x_j = \neg c_{i_3}$: (d. h. $\neg x_j$ ist 3. Literal in Klausel C_i)

$$d_{in}^{(x_j,C_i),l} := \begin{cases} 0 & \text{falls } l \in \{3,5,6,7\} \\ 1 & \text{falls } l \in \{1,2,4\} \end{cases}$$

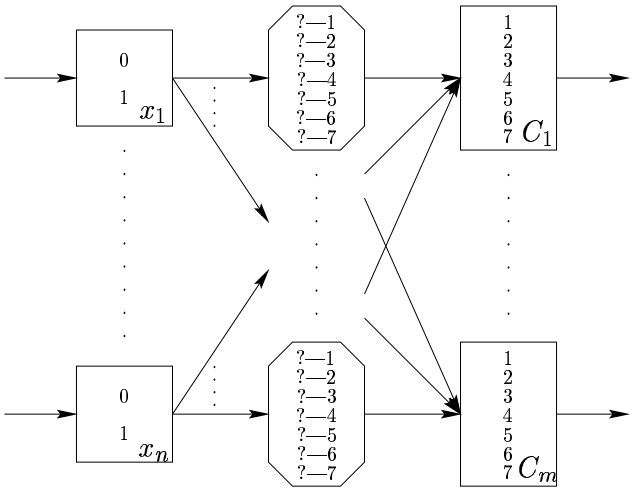


Abbildung 3.2: Systemgraph $\mathfrak{S}$ der Funktion $f = C_1 \wedge \ldots \wedge C_m$

Damit ergibt sich durch die Wandler folgende Abbildung von den jeweiligen Variablen-Blöcken zu dem entsprechenden Klausel-Block. Da in den Klauseln natürlich auch negierte Variablen vorkommen können, werden hier nur die Literale angegeben[2]):

c_{i_1}	c_{i_2}	c_{i_3}	Implementierung
0	0	0	nicht möglich
1	0	0	1
0	1	0	2
0	0	1	3
1	1	0	4
1	0	1	5
0	1	1	6
1	1	1	7

Somit haben wir aus der *3-KNF* einen Systemgraphen $\mathfrak{S}$ (Abbildung 3.2) generiert, der auf Zugehörigkeit zu *PART* überprüft werden kann. Wie man leicht erkennt, sind der Graph und die Blöcke in Polynomialzeit (zur Eingabelänge) berechenbar.

Jetzt ist noch zu zeigen, daß

$$f \in \textit{3-KNF-SAT} \Longleftrightarrow \mathfrak{S} \in \textit{PART}.$$

Sei f erfüllbar mit der Belegung $x_1 := z_1, \ldots, x_n := z_n$ $(z_i \in \{0,1\})$. Dann erhält man eine gültige Implementierung der Blöcke, indem die Variablen-Blöcke entsprechend der Variablenbelegung $z_1, \ldots, z_n$ implementiert wer-

[2]Das entspricht einer Klausel, die nur nicht-negierte Variablen enthält.

den. Also:

$$l_{x_i} := z_i \quad \forall i = 1, \ldots, n.$$

Da bei einer erfüllenden Variablenbelegung alle Klauseln erfüllt werden, existiert — aufgrund der Konstruktion — auch für jeden Klausel-Block C_i eine Implementierung $l_{C_i} \in \{1, \ldots, 7\}$, so daß die Wandler implementierbar sind. Damit ist also $\mathfrak{S} \in PART$.

Andererseits sei $\wr$ eine gültige Partitionierung des Graphen. Die Implementierung l_{C_i} eines einzelnen Klausel-Blocks C_i zwingt aufgrund der Implementierungsmöglichkeiten der Wandler die zugehörigen Variablen-Blöcke zu Implementierungen, die die Klausel C_i erfüllen. Da dies für alle Klausel-Blöcke gilt, ergeben sich die Implementierungen der Variablen-Blöcke zum Schnitt aller möglichen Erfüllungen der Klauseln. Somit ergibt sich die erfüllende Belegung von f direkt aus den Implementierungen der Variablen-Blöcke:

$$z_i := l_{x_i} \quad \forall i = 1, \ldots, n$$

Damit ist die polynomielle Reduzierbarkeit von *3-KNF-SAT* zu *PART* bewiesen.

Zum Beweis der $\mathcal{NP}$-Vollständigkeit von *PART* fehlt noch der Nachweis, daß $PART \in \mathcal{NP}$. Es sei $\mathfrak{S} \in PART$ zu entscheiden. Eine nichtdeterministische Turingmaschine „errate“ (in Polynomialzeit) eine gültige Implementierung für $\mathfrak{S}$. Ob diese Implementierung gültig ist, ist in polynomieller Zeit mittels eines Traversierungs-Algorithmus nachprüfbar. Also ist $PART \in \mathcal{NP}$ und damit auch $\mathcal{NP}$-vollständig. □

Für den Beweis wurden Blöcke mit 7 verschiedenen Zeitmodelldomänen verwendet. Man könnte annehmen, daß darin die $\mathcal{NP}$-Vollständigkeit begründet ist. Da die Partitionierung hybrider Systeme aber nur drei Zeitmodelldomänen besitzt, könnte das praxisrelevante Problem eventuell doch in $\mathcal{P}$ liegen. Dem ist aber nicht so:

Bemerkung 3.1. *PART* ist $\mathcal{NP}$-vollständig, auch wenn $|D| = 3$ festgelegt wird.

Beweis. Es genügt zu zeigen, daß man eine Klausel mit 3 Literalen (mit polynomiellem Aufwand) so darstellen kann, daß die sieben erfüllenden Belegungen implementierbar sind, aber die nicht-erfüllende Belegung nicht. Denn dann kann man dies auf alle im Beweis von Satz 3.4 erzeugten Klauseln anwenden und diesen Beweis nochmals führen mit der Einschränkung, daß $|D| = 3$ gilt.

O. B. d. A. sei

$$C = x_1 \vee x_2 \vee x_3$$

die zu wandelnde Klausel. Klauseln mit ein oder mehreren negierten Variablen lassen sich analog behandeln, ebenso unterschiedliche Variablenreihenfolgen.

Um die Klausel im Graphen nachzubilden, werden drei Klauselblöcke C^1, C^2, C^3 benötigt, für die es je drei verschiedene Implementierungsmöglichkeiten gibt.

$$\left.\begin{array}{l} B_{C^k,1} := (-,-,0,-,-) \\ B_{C^k,2} := (-,-,1,-,-) \\ B_{C^k,3} := (-,-,2,-,-) \end{array}\right\} \quad \forall k = 1,2,3$$

Die Variablen-Blöcke sind genauso definiert, wie im Beweis zu Satz 3.4:

$$\left.\begin{array}{l} B_{x_j,0} := (-,-,-,0,-) \\ B_{x_j,1} := (-,-,-,1,-) \end{array}\right\} \quad \forall j = 1,2,3$$

Nun müssen die einzelnen erfüllenden Variablenbelegungen in den Klauselblöcken so codiert werden, daß die nichterfüllende Belegung nicht implementiert werden kann, die erfüllenden Variablenbelegungen aber implementierbar bleiben.

x_1	x_2	x_3	C^1	C^2	C^3
1	0	0	2	1	0
0	1	0	1	2	1
0	0	1	0	0	2
1	1	0	2	2	1
1	0	1	2	1	2
0	1	1	1	2	2
1	1	1	2	2	2

Damit ergeben sich folgende Implementierungsmöglichkeiten für die Wandlerblöcke:

$$B_{(x_1,C^1),1} := (-,-,0,0,-)$$
$$B_{(x_1,C^1),2} := (-,-,0,1,-)$$
$$B_{(x_1,C^1),3} := (-,-,1,2,-)$$

$$B_{(x_2,C^1),1} := (-,-,0,0,-)$$
$$B_{(x_2,C^1),2} := (-,-,1,1,-)$$
$$B_{(x_2,C^1),3} := (-,-,0,2,-)$$
$$B_{(x_2,C^1),4} := (-,-,1,2,-)$$

$$B_{(x_3,C^1),1} := (-,-,0,1,-)$$
$$B_{(x_3,C^1),2} := (-,-,0,2,-)$$
$$B_{(x_3,C^1),3} := (-,-,1,0,-)$$
$$B_{(x_3,C^1),4} := (-,-,1,1,-)$$
$$B_{(x_3,C^1),5} := (-,-,1,2,-)$$

Die Implementierungsmöglichkeiten der restlichen Wandlerblöcke lassen sich aufgrund der geschickten Codierung auf die bereits Definierten zurückführen.

$$B_{(x_1,C^2)} := B_{(x_2,C^1)}$$
$$B_{(x_2,C^2)} := B_{(x_1,C^1)}$$
$$B_{(x_3,C^2)} := B_{(x_3,C^1)}$$

$$B_{(x_1,C^3)} := B_{(x_3,C^1)}$$
$$B_{(x_2,C^3)} := B_{(x_2,C^1)}$$
$$B_{(x_3,C^3)} := B_{(x_1,C^1)}$$

Dadurch entsteht der in Abbildung 3.3 dargestellte Systemgraph.

Um zu zeigen, daß die sich auf diese Weise ergebende Abbildung die notwendigen Bedingungen (also Implementierbarkeit genau aller erfüllenden Belegungen der Klausel C) erfüllt, berechnet man die gültigen Variablen-Block-Implementierungen zu allen Implementierungs-Kombinationen der Klausel-Blöcke C^1, C^2 und C^3. Die Wertetabelle der Umkehrabbildung ist in Tabelle 3.1 angegeben. Daraus erkennt man auch, daß die Abbildung sogar bijektiv ist.

Da die Einschränkung der Zeitmodelldomänen auf $|D| = 3$ die Anzahl der Klausel-Blöcke nur um den Faktor 3 (d. h. $\mathcal{O}(1)$) vergrößert, läßt sich der Beweis von Satz 3.4 mit den adaptierten Klausel-Strukturen direkt wiederholen.

C^1	C^2	C^3	x_1	x_2	x_3	Bemerkung
0	0	0	÷	0	÷	nicht implementierbar
0	0	1	0	÷	÷	nicht implementierbar
0	0	2	0	0	1	
0	1	0	÷	0	÷	nicht implementierbar
0	1	1	÷	÷	÷	nicht implementierbar
0	1	2	÷	0	1	nicht implementierbar
0	2	0	÷	÷	÷	nicht implementierbar
0	2	1	0	÷	÷	nicht implementierbar
0	2	2	0	÷	1	nicht implementierbar
1	0	0	÷	÷	÷	nicht implementierbar
1	0	1	0	÷	÷	nicht implementierbar
1	0	2	0	÷	1	nicht implementierbar
1	1	0	÷	÷	0	nicht implementierbar
1	1	1	÷	÷	0	nicht implementierbar
1	1	2	÷	÷	1	nicht implementierbar
1	2	0	÷	÷	0	nicht implementierbar
1	2	1	0	1	0	
1	2	2	0	1	1	
2	0	0	÷	0	÷	nicht implementierbar
2	0	1	÷	÷	÷	nicht implementierbar
2	0	2	÷	0	1	
2	1	0	1	0	0	
2	1	1	1	÷	0	nicht implementierbar
2	1	2	1	0	1	
2	2	0	1	÷	0	nicht implementierbar
2	2	1	1	1	0	
2	2	2	1	1	1	

Tabelle 3.1: gültige Implementierungen für Klausel- und Variablen-Blöcke (÷ bedeutet, daß keine gültige Implementierung für den Variablen-Block existiert)

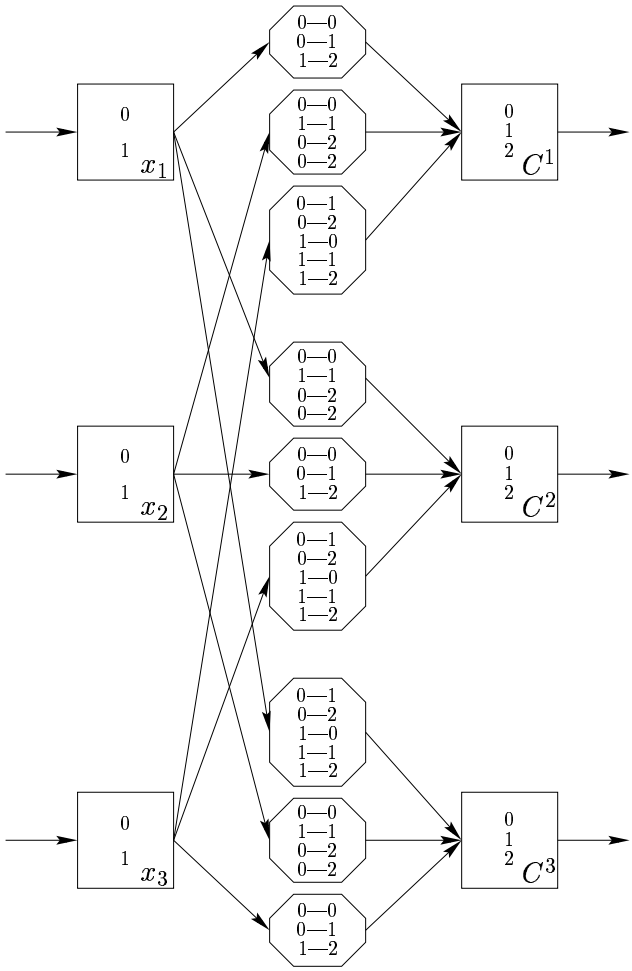


Abbildung 3.3: Systemgraph $\mathfrak{S}$ für die Klausel $C = x_1 \vee x_2 \vee x_3$, mit $|D| = 3$

Es gilt also die Behauptung. □

In diesem Kapitel wurde gezeigt, daß *PART* $\mathcal{NP}$-vollständig ist und auch, daß dies nicht nur für die allgemeine Form von *PART*, bei der die Menge der Zeitmodelldomänen beliebig sein darf, gilt, sondern daß auch die praxisrelevante Version mit auf $|D| = 3$ eingeschränkter Zeitmodelldomänen-Menge $\mathcal{NP}$-vollständig ist.

Aus den gezeigten Sätzen ergibt sich somit, daß für die Entscheidung, ob es für einen Systemgraphen eine Implementierung gibt oder nicht, exponentielle Laufzeit benötigt wird (sofern $\mathcal{P} \neq \mathcal{NP}$). Wie eingangs schon erwähnt, bedeutet dies weiterhin, daß auch $PART_m^{\text{opt}}$ ein $\mathcal{NP}$-vollständiges Optimierungsproblem ist, also auch das Finden einer optimalen Implementierung (bzgl. einer Kostenfunktion m) exponentielle Zeit kostet.

Aber auch aus dem Beweis selbst lassen sich Schlußfolgerungen für das Lösen von $PART$ bzw. die Optimierung von $PART^{\text{opt}}$ ziehen.

Es war bisher nicht klar, welche Eigenschaft des Systemgraphen das Problem „schwierig“ macht. Eine Vermutung war, daß die (erlaubten) Signalrückführungen oder rekonvergente Signalwege für die $\mathcal{NP}$-Vollständigkeit verantwortlich seien. Für den Beweis wurden aber weder Zykel noch Rekonvergenzen benötigt. Vielmehr ist für die Komplexität des Problems die „Breite“ (d. h. die parallel angeordneten Funktionsblöcke) des Systemgraphen verantwortlich (das wird in Kapitel 9 deutlich) sowie die inhärent nicht auflösbaren Abhängigkeitsverhältnisse zwischen jedem Funktionsblock und seinen benachbarten Wandlern bezüglich deren Zeitmodelldomänen.

Daraus folgt, daß es vermutlich keine sinnvollen Klassen (wie z. B. Systemgraphen ohne Zyklen (also Schaltnetze) oder Systemgraphen ohne Rekonvergenzen) gibt, die nur polynomielle Zeit zur Lösung bzw. Optimierung benötigen. Jedoch konnte für die Klasse der zykelfreien Systemgraphen mit logarithmischer Breite ein polynomieller Optimierungsalgorithmus gefunden werden, sofern bzgl. einer linearen Kostenfunktion optimiert werden soll (siehe Kapitel 9.1.2). Diese Klasse läßt sich sogar noch erweitern, indem man logarithmisch viele Zykel zuläßt.

Kapitel 4

Approximation

Aufgrund der Ergebnisse von Kapitel 3 ist klar, daß $PART^{\text{opt}}$ für allgemeine Systemgraphen nicht effizient (d. h. in Polynomialzeit) gelöst werden kann.

Daher stellt sich die Frage, ob es nicht möglich ist, einen Algorithmus zu finden, der in Polynomialzeit eine Lösung findet, die fast so gut ist wie die optimale Lösung.

Für viele $\mathcal{NP}$-vollständige Probleme existieren Approximationsalgorithmen, die beliebig nahe an die optimale Lösung heranreichen (Approximationsschema) oder zumindest einen konstanten, maximalen Verlustfaktor garantieren.

Im folgenden wird gezeigt, daß es für $PART^{\text{opt}}$ keinen Approximationsalgorithmus geben kann, der garantiert, daß seine Lösungen für beliebige Systemgraphen um höchstens einen konstanten Faktor schlechter ist als die Kosten der jeweils optimalen Implementierung. Für jeden Approximationsalgorithmus und jeden beliebigen Verlustfaktor α existiert ein Systemgraph, dessen vom Approximationsalgorithmus gefundene Kosten um einen Faktor $> \alpha$ schlechter sind als die optimale Lösung.

Das bedeutet zwar nicht, daß man grundsätzlich keine approximative Lösung erhalten kann, aber daß die Approximationsgüte vom Systemgraphen abhängig ist.

4.1 Theoretische Grundlagen

Definition 4.1. Eine Implementierung $\mathfrak{l}^*$ heißt *optimal* (bzgl. der Kostenfunktion m) für den Systemgraphen $\mathfrak{S}$, falls

$$m(\mathfrak{S}, \mathfrak{l}^*) = PART_m^{\text{opt}}(\mathfrak{S}).$$

Eine optimale Lösung verursacht also minimale Kosten bezüglich m.

Definition 4.2. Ein Algorithmus A heißt α-*Approximationsalgorithmus* für

$PART_m^{\text{opt}}$, falls für alle Systemgraphen $\mathfrak{S}$ gilt:

$$\alpha \cdot PART_m^{\text{opt}}(\mathfrak{S}) \geq A(\mathfrak{S})$$

Das Ergebnis eines α-Approximationsalgorithmus ist also für beliebige Systemgraphen um höchstens Faktor α schlechter als die Kosten einer optimalen Lösung. α wird auch *Verlustfaktor* oder *Approximationsgarantie* genannt.

Wenn man einen polynomiellen Approximationsalgorithmus mit Approximationsgarantie für $PART^{\text{opt}}$ finden würde, könnte man die Kosten der optimalen Lösung abschätzen und sogar verschiedene Heuristiken mit dem Optimum vergleichen.

Daher ist es interessant zu wissen, ob solch ein polynomieller Approximationsalgorithmus überhaupt existieren kann.

Satz 4.1. *Sei P ein beliebiges Minimierungsproblem, S ein $\mathcal{NP}$-vollständiges Sprachenproblem und τ eine polynomielle Reduktion von S auf P. Es gelte:*

$$f \in S \Longrightarrow P(\tau(f)) = 1$$
$$f \notin S \Longrightarrow P(\tau(f)) > 1 + \varepsilon$$

Dann existiert für P kein polynomieller Approximationsalgorithmus mit $\alpha \leq 1 + \varepsilon$ (vorausgesetzt $\mathcal{P} \neq \mathcal{NP}$) [20].

Beweis. Angenommen es existiere für P ein polynomieller Approximationsalgorithmus A mit $\alpha \leq 1 + \varepsilon$.

Dann gilt aufgrund der Approximationsgarantie

$$f \in S \Longrightarrow A(\tau(f)) \in [1, 1+\varepsilon] \left(\in [-\infty, 1+\varepsilon]\right)$$
$$f \notin S \Longrightarrow A(\tau(f)) \in (1+\varepsilon, \infty)$$

also mittels Kontraposition

$$A(\tau(f)) > 1 + \varepsilon \Longrightarrow f \notin S$$
$$A(\tau(f)) \leq 1 + \varepsilon \Longrightarrow f \in S.$$

Da $A \in \mathcal{P}$ und $\tau \in \mathcal{P}$ kann somit $f \in S$, bzw. $f \notin S$ mit polynomiellem Zeitaufwand durch die Berechnung von $A(\tau(f))$ entschieden werden. Dies steht aber im Widerspruch zur $\mathcal{NP}$-Vollständigkeit von S, sofern $\mathcal{P} \neq \mathcal{NP}$. □

4.2 Nicht-Approximierbarkeit von $PART^{\mathbf{opt}}$

Mit Hilfe von Satz 4.1 läßt sich zeigen, daß es für $PART^{\text{opt}}$ i. a. keine Approximationsgarantie geben kann.

Es gibt jedoch triviale Ausnahmen, da neben dem Systemgraphen auch noch die Kostenfunktion das Optimierungsergebnis bestimmt. Wenn man z. B. die Kostenfunktion

$$m(\mathfrak{S}, \mathfrak{l}) := \begin{cases} 2 & \text{falls } \sum_{l \in \mathfrak{l}} l \geq const \\ 1 & \text{sonst} \end{cases}$$

wählt, erhält man einen polynomiellen 2-Approximationsalgorithmus A indem dieser stets 2 zurückgibt. Selbst wenn $PART_m^{\text{opt}}(\mathfrak{S}) = 2$ sein sollte, ist das Ergebnis von A nur um Faktor 2 schlechter.

Die praxisrelevanten Kostenfunktionen sind jedoch nach oben unbeschränkt (d. h. $\nexists K \in \mathbb{R}\ \forall \mathfrak{S}, \mathfrak{l}: \ m(\mathfrak{S}, \mathfrak{l}) \leq K$) und sind von den nichtfunktionalen Eigenschaften *aller* Blöcke und Wandler abhängig. Es ist nicht möglich, den Beweis für diese sehr allgemeine Funktionsklasse zu zeigen. Stattdessen soll hier nur die Nicht-Approximierbarkeit für $PART_m^{\text{opt}}$ gezeigt werden, wenn m aus der Klasse der linearen Funktionen (für Definition und Eigenschaften von linearen Funktionen sei z. B. auf [13] verwiesen) stammt, denn lineare Funktionen sind die vermutlich am häufigsten verwendeten Kostenfunktionen. An andere Kostenfunktionen läßt sich der Beweis leicht anpassen.

Satz 4.2. *Sei m eine nichtkonstante, lineare Abbildung. Dann gibt es für $PART_m^{opt}$ — unter der Voraussetzung $\mathcal{P} \neq \mathcal{NP}$ — keinen polynomiellen Approximationsalgorithmus mit konstantem Verlustfaktor.*

Beweis. Sei A ein polynomieller α-Approximationsalgorithmus (α konstant) für $PART_m^{\text{opt}}$.

Das Ziel ist es nun, eine polynomielle Reduktion τ zu finden, so daß gilt

$$f \in \textit{3-KNF-SAT} \Longrightarrow PART_m^{\text{opt}}(\tau(f)) = 1$$
$$f \notin \textit{3-KNF-SAT} \Longrightarrow PART_m^{\text{opt}}(\tau(f)) > \alpha,$$

denn dann läßt sich die Aussage auf Satz 4.1 zurückführen.

O. B. d. A. sei

$$m(\mathfrak{S}, \mathfrak{l}) := \sum_{k \in V} p_1^{k, l_k} + \sum_{(i,j) \in E} p_1^{(i,j), l_{(i,j)}}$$

also nur von einer Eigenschaft abhängig.

Wir nutzen den gleichen Systemgraphen wie im Beweis zu Satz 3.4. Allerdings erhalten die Klauselblöcke noch eine zusätzliche achte Zeitmodelldomäne, die die Nichterfüllbarkeit derjenigen Klausel repräsentiert. Dies ist notwendig, weil für die Lösbarkeit von $PART_m^{\text{opt}}(\mathfrak{S})$ der Systemgraph $\mathfrak{S}$ implementierbar sein muß. Also muß $\tau(f)$ einen implementierbaren Systemgraphen ergeben, unabhängig davon, ob $f \in$ *3-KNF-SAT* oder nicht.

$$B_{C_i,8} := (-,-,8,-,-) \quad \forall i = 1,\ldots,m$$

Die Wandlerblöcke müssen entsprechend angepaßt werden, damit die neuen Implementierungsmöglichkeiten der Klausel-Blöcke bei beliebigen Implementierungen der Variablen-Blöcke (d. h. auch bei den nichterfüllenden Belegungen) genutzt werden können.

$$\left.\begin{aligned} B_{(x_j,C_i),8} &:= (-,-,0,8,-) \\ B_{(x_j,C_i),9} &:= (-,-,1,8,-) \end{aligned}\right\} \quad \forall j = 1,\ldots,n\ ,\ i = 1,\ldots,m$$

Da nicht die Belegung der Variablen interessiert, sondern nur, ob alle Klauseln erfüllbar sind, werden die nichtfunktionalen Eigenschaften der Variablen-Blöcke und der Wandler auf 0 gesetzt.

$$p_1^{x_j,l} := 0 \quad \forall j = 1,\ldots,n\ ,\ l = 0,1$$
$$p_1^{(x_j,C_i),l} := 0 \quad \forall j = 1,\ldots,n\ ,\ i = 1,\ldots,m\ ,\ l = 1,\ldots,9$$

Die nichtfunktionalen Eigenschaften der Klauselblöcke sollen davon abhängen, ob die Klausel durch die Variablenbelegung erfüllt wird oder nicht. Denn wenn eine Formel nicht erfüllbar ist, dann ist bei jeder Variablenbelegung mindestens eine Klausel nicht erfüllt. In diesem Fall sollen die Kosten für den entsprechenden Systemgraphen so hoch sein, daß $PART_m^{\text{opt}}(\mathfrak{S}) > \alpha$. Andererseits soll $PART_m^{\text{opt}}(\mathfrak{S}) = 1$ gelten, wenn die Formel erfüllbar ist, d. h. alle Klauseln gleichzeitig erfüllbar sind.

Daher liegt es nahe, die Implementierungen 1–7 der Klausel-Blöcke (also „Klausel wird erfüllt“) nur mit $\frac{1}{m}$ (wobei m der Anzahl der Klauseln entspricht) zu gewichten.

$$p_1^{C_i,l} := \frac{1}{m} \quad \forall i = 1,\ldots,m\ ,\ l = 0,\ldots,7$$

Das Gewicht für die Implementierung 8 hingegen muß so groß gewählt sein, daß bereits eine nichterfüllte Klausel die Gesamtkosten auf $> \alpha$ zwingt.

$$p_1^{C_i,8} := \alpha \quad \forall i = 1,\ldots,m$$

O. B. d. A. sei die Anzahl der Klauseln $m \geq 2$, damit $\frac{1}{m} < \alpha$ garantiert ist.

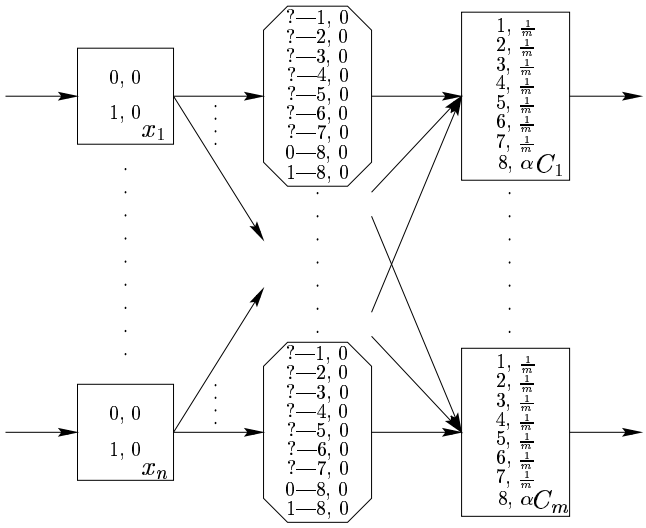


Abbildung 4.1: Optimierbarer, mit nichtfunktionalen Eigenschaften versehener Systemgraph $\mathfrak{S}$ der Funktion f in *KNF*

Wenn f erfüllbar ist, ergibt $PART_m^{\text{opt}}(\tau(f))$ somit $m \cdot \frac{1}{m} = 1$. Wenn f nicht erfüllbar ist, dann ergibt $PART_m^{\text{opt}}(\tau(f))$ einen Wert $\geq (m-1) \cdot \frac{1}{m} + \alpha > \alpha$, weil bei jeder Variablenbelegung mindestens eine Klausel nicht erfüllt werden kann.

Damit wissen wir nach Satz 4.1, daß es keinen polynomiellen Approximationsalgorithmus für $PART_m^{\text{opt}}$ mit Approximationsgarantie $\leq \alpha$ geben kann, ohne daß $\mathcal{P} = \mathcal{NP}$. Dies ist aber ein Widerspruch zur Annahme, daß A ein polynomieller α-Approximationsalgorithmus sei. □

Satz 4.2 besagt, daß es keinen Approximationsalgorithmus mit konstantem Verlustfaktor für lineare Kostenfunktionen geben kann. Er sagt aber nicht aus, daß es *keine* Approximationsalgorithmen geben kann. Die Güte der Approximation wird aber von der Kostenfunktion *und* der Gestalt des Systemgraphen abhängen.

Kapitel 5

Reduktion auf bekannte Probleme

Grundsätzlich kann jedes Problem, das in $\mathcal{NP}$ enthalten ist, auf ein beliebiges anderes $\mathcal{NP}$-vollständiges Problem abgebildet werden (aufgrund von Definition 3.4).

Dies kann man für $PART$ erreichen, indem man z. B. die $PART$ lösende, nicht-deterministische Turingmaschine auf $KNF\text{-}SAT$ reduziert und anschließend die konjunktive Normalform auf ein weiteres $\mathcal{NP}$-vollständiges Problem reduziert.

Bei einer solchen Reduktion verliert man im allgemeinen die Struktur des Problems, da es auf ein „artfremdes“ Problem abgebildet wird. Aber gerade die Struktur von $PART$ und $PART^{\text{opt}}$ sollte ausgenutzt werden, damit der Lösungsraum auf die in Frage kommenden Lösungen eingeschränkt werden kann und nicht sämtliche mögliche Lösungen betrachtet werden müssen.

Es werden also $PART^{\text{opt}}$-ähnliche Probleme gesucht, die bereits eingehend untersucht wurden und für die gute Lösungsstrategien existieren.

5.1 Das Konstruktionsproblem

In [35] wird das sogenannte *Konstruktionsproblem* vorgestellt:

> „ ... Will man irgendein System erstellen, z. B. eine Maschine, die bestimmte Funktionen erfüllt, oder eine Organisation, die bestimmte Leistungen erbringen kann, so hat man i. d. R. mehrere Möglichkeiten, ein solches System zu konstruieren. Meist hat man für eine einzelne Funktion mehrere Elemente oder Elementkombinationen (Subsysteme), die diese Funktionen realisieren. Allerdings gibt es oft Elemente oder Elementkombinationen, die miteinander unverträglich sind. Man kann nun im allgemeinen

> mehrere Systeme entwerfen, die die geforderten Leistungen erbringen. Man wird dasjenige realisieren, das ein vorgegebenes Ziel am besten erfüllt. Man erkennt leicht, daß wir es hier mit einem kombinatorischen Problem zu tun haben, mit der Besonderheit, daß gewisse Kombinationen unzulässig sind. ... "

Das Konstruktionsproblem optimiert also eine Auswahl an Elementen, bei der bestimmte Kombinationen nicht erlaubt sind. Wenn man die einzelnen Implementierungsmöglichkeiten des Systemgraphen als die alternativen Elemente auffaßt, und die Zeitmodell-Abhängigkeiten zwischen Funktionsblöcken und benachbarten Wandlern explizit auflistet, kann man $PART^{\text{opt}}$ auf das Konstruktionsproblem abbilden.

Das Konstruktionsproblem wird in [35] als konjunktive Normalform dargestellt,wobei man diejenige erfüllende Belegung sucht, die die Kosten der Konstruktion minimiert. Man will also *KNF-SAT* mit der kostengünstigsten Variablen-Belegung lösen. Da *KNF-SAT* nach [22] in $\mathcal{O}(1,5045^n)$ entschieden werden kann, läßt sich $PART^{\text{opt}}$ auf diese Weise in $\mathcal{O}(1,5045^n \cdot \text{poly}(n))$ lösen (wobei n die Anzahl der Variablen, also die Summe aller Implementierungsmöglichkeiten, ist).

Nachteilig daran ist, daß eine approximative Lösung von *KNF-SAT* nicht brauchbar ist, um $PART^{\text{opt}}$ zu approximieren. Dies ergibt sich aus der Tatsache, daß eine Approximation von *KNF-SAT* nur einige Klauseln erfüllt und andere nicht. Das bedeutet für den Systemgraphen, daß nicht alle Zeitmodell-Abhängigkeiten erfüllt werden würden, das Ergebnis also unbrauchbar wäre.

In [35] wird leider kein weiterer Lösungsvorschlag gemacht und auch keine Heuristiken für das Konstruktionsproblem angegeben. Auch wurde keine weitere Literatur gefunden, die das Konstruktionsproblem behandelt.

5.2 Probleme aus der Betriebswirtschaftslehre

Es wäre zu erwarten, daß man im Bereich Operations Research der Betriebswirtschaftslehre einige ähnlich zu $PART^{\text{opt}}$ gelagerte Optimierungsprobleme findet.

In der Literatur beschäftigt man sich jedoch hauptsächlich mit Standard-Problemen der Informatik, wie z. B. das Rucksackproblem, Traveling Salesman, Lagerhaltungs-, Transport- und Zuordnungsprobleme. Das gegebene Problem wird dann üblicherweise auf diese Standardprobleme reduziert und anschließend mittels Linearer Programmierung gelöst.

Auf die Anwendung von Heuristiken wird im allgemeinen verzichtet. Stattdessen begnügt man sich meistens mit kleinen Problemgrößen, um die Laufzeit in einem akzeptablen Rahmen zu halten.

Daher konnte im betriebswirtschaftlichen Bereich (außer dem Konstruktionsproblem) kein geeignetes Problem gefunden werden, auf das $PART^{\text{opt}}$ adäquat reduziert werden kann.

5.3 Probleme aus der Informatik

Da $PART^{\text{opt}}$ ein $\mathcal{NP}$-vollständiges Optimierungsproblem ist, kann man sich bei der Suche nach ähnlichen Problemen auf die Klasse $\mathcal{NP}$-vollständiger Optimierungsprobleme beschränken. Die vermutlich vollständigste Auflistung von Problemen dieser Klasse ist in [6] zu finden. Aber auch in [6] ist kein auf direktem Weg an $PART^{\text{opt}}$ adaptierbares Problem enthalten.

Der Grund hierfür ist im Systemgraphen zu suchen. Die bisherigen Graphen-Probleme bieten keine Möglichkeit, die Belegung eines Knotens oder einer Kante aus mehreren Alternativen auszuwählen. Wenn man nun für jede Implementierungsmöglichkeit eines Funktionsblocks einen separaten Knoten erzeugen würde, müßte man fordern, daß genau ein Knoten aus der Menge der alternativen Knoten (d. h. eine Implementierungsmöglichkeit aller Implementierungsmöglichkeiten eines Funktionsblocks) gewählt wird, während die anderen unberücksichtigt bleiben. Auch dieses können die (untersuchten) Graphen-Probleme nicht leisten.

Mengentheoretische Probleme, die eine Menge an Alternativen für ein Element zulassen, sind hingegen nicht in der Lage, die Abhängigkeiten zwischen der Implementierung eines Blocks und seiner benachbarten Wandler zu berücksichtigen.

Zudem ist es aber auch möglich, indirekt zu argumentieren — denn ein $\mathcal{NP}$-vollständiges, nicht mit konstantem Verlustfaktor approximierbares Optimierungsproblem, kann nicht direkt auf ein Optimierungsproblem abgebildet werden, für das ein Approximationsalgorithmus mit konstantem Verlustfaktor existiert. Auf diese Weise kann bereits ein Großteil der in Frage kommenden Probleme ausgeschlossen werden.

Zwar läßt sich $PART^{\text{opt}}$ auf das Konstruktionsproblem zurückführen, da aber keine Lösungsstrategien, bzw. Heuristiken vorliegen, ist es notwendig, eigene Verfahren zur Optimierung, bzw. Approximation von $PART^{\text{opt}}$ zu entwickeln.

Aufgrund dieser Ergebnisse kann man davon ausgehen, daß die Optimierung eines Systemgraphen ein völlig neuartiges Problem ist, das sich nicht auf bereits bekannte und genauer untersuchte Optimierungsprobleme zurückführen läßt.

Kapitel 6

Einzelziel-Optimierung

Unter der Einzelziel-Optimierung versteht man die Optimierung, die nur eine Zielsetzung verfolgt, z. B. die Fläche oder die Laufzeit zu minimieren. Es können zwar auch mehrere nichtfunktionale Eigenschaften betrachtet werden, aber nur in der Art, daß sie zu einer einzigen zusammengefaßt werden können. Zum Beispiel wäre es möglich, den Mittelwert des Flächenbedarfs und des Leistungsverbrauchs pro Funktionsblock bzw. Wandler zu berechnen und diesen Mittelwert als nichtfunktionale Eigenschaft des Blocks bzw. Wandlers zu betrachten. Daher wird im folgenden angenommen, daß die Blöcke und Wandler nur eine nichtfunktionale Eigenschaft besitzen, bzw. nur eine Eigenschaft betrachtet wird. Der Wert dieser nichtfunktionalen Eigenschaft von Implementierung l des Funktionsblocks bzw. Wandlers i wird mit $p^{i,l}$ bezeichnet.

Wie schon in Abschnitt 2.7 angedeutet, lassen sich die Gesamtkosten der verschiedenen nichtfunktionalen Eigenschaften nicht nach einem einheitlichen Schema berechnen. Die Gesamtfläche, totaler Leistungsverbrauch und Gesamt-Fertigungskosten berechnen sich als Summe der jeweiligen Kosten der einzelnen Blöcke und Wandler. Sie haben also eine lineare Kostenfunktion. Ebenso verhält sich die Chipausbeute, wenn man die Kosten der einzelnen Blöcke logarithmiert (wg. $y = \prod_i x_i = \exp\left(\sum_i \ln x_i\right)$).

Die Laufzeit, bzw. Verzögerungszeit der Schaltung läßt sich hingegen nicht durch eine lineare Funktion ausdrücken, da die Verzögerungszeit der Schaltung nur durch die Laufzeit des kritischen Pfades bestimmt ist, die aber nur sehr ungenau mittels einer linearen Funktion abgeschätzt werden kann. Der kritische Pfad läßt sich zwar z. B. mit Hilfe eines leicht modifizierten SSSP-Algorithmus (Single Source Shortest Path) berechnen (siehe [18, S. 27–29]), aber durch die Nichtlinearität kann man die Verzögerungszeit mit z. B. dem Flächenbedarf nicht in eine einzelne nichtfunktionale Eigenschaft zusammenfassen.

Wenn nach nur einem Ziel optimiert werden soll, werden im allgemeinen

zusätzliche Randbedingungen wie maximal zur Verfügung stehende Chipfläche nicht berücksichtigt, da dies ein weiteres Ziel darstellen würde. Es ist aber trotzdem möglich, diese Randbedingungen während der Laufzeit der Optimierungs-Algorithmen ständig zu überprüfen und (Teil-) Ergebnisse gegebenenfalls als nicht implementierbar zu markieren, indem ihre Implementierungskosten auf ∞ gesetzt werden. Für ein zu minimierendes Ziel ergibt sich somit folgende, die Randbedingungen berücksichtigende Kostenfunktion:

$$m'(\mathfrak{x}) := \begin{cases} m(\mathfrak{x}) & \text{falls alle Randbedingungen erfüllt sind} \\ \infty & \text{sonst} \end{cases}$$

Die Berücksichtigung dieser Randbedingungen wird erst bei der Mehrziel-Optimierung von Bedeutung, da auf diese Weise ungünstige Implementierungen aus dem Lösungsraum ausgeschlossen werden.

6.1 Einfache Approximationsgarantien

Mit s_{min} werden die minimal möglichen Kosten des Systemgraphen bezeichnet.

$$s_{min} := \min_{\mathfrak{l} \in \mathbb{I}} m(\mathfrak{S}, \mathfrak{l})$$

Dies ist die untere Schranke der Kosten des Systemgraphen und wird im allgemeinen von keiner Implementierung erreicht, da die Abhängigkeiten zwischen Blöcken und Wandlern nicht berücksichtigt werden.

s_{max} bezeichnet dagegen die Kosten der ungünstigsten, also teuersten Implementierung.

$$s_{max} := \max_{\mathfrak{l} \in \mathbb{I}} m(\mathfrak{S}, \mathfrak{l})$$

Das entspricht einer oberen Schranke der Implementierungskosten des Systemgraphen $\mathfrak{S}$ und ist ebenfalls im allgemeinen nicht erreichbar.

Damit ergibt sich folgende, für die Abschätzung der Approximationsgüte wichtige Beziehung:

$$s_{min} \leq PART_m^{\text{opt}}(\mathfrak{S}) \leq \hat{s} \leq s_{max}$$

(wobei $\hat{s}$ die Kosten einer beliebigen Implementierung von $\mathfrak{S}$ darstellt).

Wenn aus dem Zusammenhang hervorgeht, welcher Systemgraph gemeint ist und bzgl. welcher Kostenfunktion zu optimieren ist, werden die Kosten der optimalen Lösung statt $PART_m^{\text{opt}}(\mathfrak{S})$ im folgenden mit s^* bezeichnet.

$$s^* := PART_m^{\text{opt}}(\mathfrak{S})$$

Die Qualität eines Approximationsalgorithmus wird an seiner Laufzeit gemessen und an seiner Approximationsgarantie. In Kapitel 4 wurde gezeigt, daß es keine Approximationsgarantie mit konstantem Verlustfaktor (unabhängig vom Systemgraphen) geben kann. Jeder Approximationsalgorithmus wird $PART_m^{\text{opt}}(\mathfrak{S})$ aber in jedem Fall bis auf Faktor

$$\alpha_{pri} \leq \frac{s_{max}}{s^*} \leq \frac{s_{max}}{s_{min}} \tag{6.1}$$

approximieren. Der Faktor $\frac{s_{max}}{s_{min}}$ kann somit nicht von einem Approximationsalgorithmus überschritten werden, ist also die triviale Approximationsgarantie für den gegebenen Graphen $\mathfrak{S}$ und die Kostenfunktion m. Da sie bereits vor dem Ausführen des Algorithmus bekannt ist, ist sie eine *a-priori-Abschätzung.*

Wenn man das Ergebnis $\hat{s}$ eines Approximationsalgorithmus schon kennt, kann die Approximationsgüte durch

$$\alpha_{post} \leq \frac{\hat{s}}{s^*} \leq \frac{\hat{s}}{s_{min}}$$

noch genauer abgeschätzt werden. Diese wird *a-posteriori-Abschätzung* genannt.

6.2 Preprocessing

Es ist möglich, daß in einem Systemgraphen für einen Block zwei verschiedene Implementierungsmöglichkeiten existieren, die beide in der gleichen Zeitmodelldomäne operieren.

Wenn man nach nur einem Ziel optimiert und die Kostenfunktion monoton (d. h. $\|x\| \leq \|y\| \Rightarrow f(x) \leq f(y)$) ist, dann genügt es, bei der Optimierung für jeden Funktionsblock nur diejenigen Implementierungsmöglichkeiten zu betrachten, für die es keine kostengünstigere Implementierungsmöglichkeit mit gleicher Zeitmodelldomäne gibt. Die gleiche Argumentation kann auch auf die Wandler angewendet werden.

Somit reicht es aus, für jeden Funktionsblock und Wandler nur 3, bzw. 9 Implementierungsmöglichkeiten zu betrachten, sofern bzgl. einer monotonen Kostenfunktion nach nur einem Ziel optimiert wird.

Alle in Kapitel 2.7 vorgestellten Kostenfunktionen sind monoton. Es sind auch keine sinnvollen Kostenfunktionen vorstellbar, die nicht monoton sind, denn das würde bedeuten, daß man einen teureren Funktionsblock oder Wandler einem — die gleiche Zeitmodelldomäne besitzenden — günstigeren Block vorziehen müßte, weil sich daraus eine kostengünstigere Gesamtimplementierung ergibt.

Außerdem ist es sinnvoll, bereits vor der Optimierung zu überprüfen, ob bestimmte Implementierungsmöglichkeiten von Funktionsblöcken oder Wandlern überhaupt benutzt werden können, weil die benachbarten Wandler, bzw. Funktionsblöcke eventuell gar keine Implementierungsmöglichkeit für die entsprechende Zeitmodelldomäne besitzen.

Dieses Preprocessing ist mit Hilfe einer einfachen Traversierung des Systemgraphen durchführbar und benötigt $\mathcal{O}(|V| + |E| + \sum_{k \in V \cup E} |L_k|)$ Laufzeit.

Es ist also durchaus lohnenswert, die unbenutzbaren Implementierungsmöglichkeiten der Blöcke eines Systemgraphen vor der Optimierung zu entfernen, da für die Optimierung des Systemgraphen exponentielle Laufzeit benötigt wird, und durch das Preprocessing die Größe des Lösungsraums eines Systemgraphen deutlich verringert werden kann.

6.3 Optimierungsverfahren

Die nachfolgend vorgestellten Optimierungsverfahren lassen sich in verschiedene Klassen einteilen.

Unter exakten Optimierungsverfahren werden Algorithmen verstanden, die in jedem Fall das optimale Ergebnis zurückliefern. Darunter fällt die lineare Programmierung (Kapitel 7), das Branch & Bound-Verfahren (Kapitel 8) und die dynamische Programmierung (Kapitel 9).

Aufgrund der $\mathcal{NP}$-Vollständigkeit von $PART^{\text{opt}}$ ist klar, daß sich $PART^{\text{opt}}$ nicht in Polynomialzeit lösen läßt (außer eventuell in Spezialfällen, bzw. falls $\mathcal{P} = \mathcal{NP}$). Daher werden exakte Optimierungsverfahren mindestens exponentielle Zeit zum Optimieren von Systemgraphen benötigen. Jedoch versuchen die exakten Optimierungsverfahren durch geschicktes Einschränken des Suchraums und Eliminierung nicht-optimaler Lösungsgebiete den Rechenaufwand möglichst gering zu halten, damit auch größere und kompliziertere Systemgraphen noch mit realistischem Zeitaufwand optimiert werden können.

Im Gegensatz dazu berechnen Approximations-Algorithmen nur annähernd optimale Ergebnisse, versuchen also das Optimum zu approximieren. Approximationsalgorithmen garantieren, daß eine bestimmte Approximationsschranke α von ihren berechneten Lösungen eingehalten wird.

Häufig werden auch Algorithmen in diese Klasse gezählt, die eine von der Eingabe abhängige Approximationsschranke garantieren. Daher kann man auch die in Kapitel 13 erwähnten Greedy-Algorithmen zu der Klasse der Approximations-Algorithmen zählen.

Die letzte Klasse ist die der Heuristiken. Unter einer Heuristik versteht man einen Algorithmus, der im allgemeinen eine Lösung liefert, die nahe am Optimum ist. Es gibt allerdings durchaus Konfigurationen, bei denen

der Algorithmus „versagt“ und nur eine sehr schlechte Lösung liefert. Eine Heuristik wird durch die Eigenschaft charakterisiert, daß man aus ihr und ihrer Eingabe keine Approximationsgarantie herleiten kann.

Zu den Heuristiken zählen Simulated Annealing (Kapitel 10), Tabu Search (Kapitel 11) und Genetische Algorithmen (Kapitel 12).

Kapitel 7

Lineares Programmieren

Der vermutlich allgemeinste Ansatz zur Optimierung komplexer Probleme ist die Lineare Programmierung. Darunter versteht man Algorithmen, die eine lineare Funktion minimieren (bzw. maximieren) wobei die Lösung innerhalb eines durch lineare Ungleichungen vorgegebenen Suchraums liegen muß.

7.1 Grundlagen

In der Standardform sind n reellwertige Variablen $x_1, \ldots, x_n$ gegeben, die m lineare Gleichungen

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i \quad i = 1, \ldots, m$$

unter der Nichtnegativitätsbedingung

$$x_j \geq 0 \quad j = 1, \ldots, n$$

erfüllen sollen. Unter allen Lösungsvektoren $\mathfrak{x} = (x_1, \ldots, x_n)$ wird nun derjenige gesucht, der die lineare *Zielfunktion*

$$(x_1, \ldots, x_n) \mapsto \sum_{j=1}^{n} c_j x_j$$

minimiert. Die m linearen Gleichungen werden auch als *Restriktionen* oder *Constraints* bezeichnet [31].

Wenn man nun $a_{11}, \ldots, a_{mn}$ zu einer Matrix $\mathfrak{A}$ und $b_1, \ldots, b_m$ und $c_1, \ldots, c_n$ zu Vektoren $\mathfrak{b}$ und $\mathfrak{c}$ zusammenfaßt, erhält man die *kanonische Form eines linearen Programms*:

Definition 7.1. Die kanonische Form einer Aufgabe der *linearen Programmierung* (*LP*) lautet:

$$\begin{aligned} &\min \mathfrak{c}^{\mathsf{T}}\mathfrak{x}, \text{ so daß} \\ &\quad \mathfrak{A}\mathfrak{x} \geq \mathfrak{b} \\ &\quad \mathfrak{x} \geq 0 \end{aligned} \tag{LP}$$

Bemerkung 7.1.

- Die Nichtnegativitätsbedingung ist keine echte Einschränkung, da man $x \in \mathbb{R}$ durch $x^+ - x^-$ mit $x^+, x^- \in \mathbb{R}^+$ nachbilden kann.
- Wenn die Zielfunktion zu maximieren ist, so läßt sich dies durch

  $$\max \mathfrak{c}^{\mathsf{T}}\mathfrak{x} = -\min -\mathfrak{c}^{\mathsf{T}}\mathfrak{x},$$

  also durch Minimieren der negativen Zielfunktion und anschließendem Invertieren des Ergebnisses erreichen.
- Lineare Gleichungen kann man durch zwei Ungleichungen darstellen (z. B. $x = b \leftrightarrow x \geq b, -x \geq -b$), sie sind also auch in einem linearen Programm als Restriktionen zulässig.

In der Literatur wird bei der Definition der linearen Programmierung statt $\mathfrak{A}\mathfrak{x} \geq \mathfrak{b}$ häufig auch die Gleichheit $\mathfrak{A}\mathfrak{x} = \mathfrak{b}$ gefordert. Diese Variante ist aber äquivalent zu *LP*, da man sogenannte *Slack-Variablen* s_i einführen kann, die nicht in die Kostenfunktion eingehen, und die Gleichheit durch die Eigenschaft

$$\sum_{j=1}^{n} a_{ij}xj \geq b_i \Longleftrightarrow \sum_{j=1}^{n} a_{ij}xj + s_i = b_i,\ s_i \geq 0$$

„aufweichen“.
Zur weiteren Arbeit mit der linearen Programmierung sind die folgenden Begriffe nützlich.

Definition 7.2.

- $\mathfrak{x}$ heißt *Lösung*, falls gilt: $\mathfrak{A}\mathfrak{x} \geq \mathfrak{b},\ \mathfrak{x} \geq 0$
- Ein lineares Programm heißt *lösbar*, wenn eine Lösung $\mathfrak{x}$ existiert.

  $$\mathcal{L}(\mathfrak{A}, \mathfrak{b}) := \left\{\mathfrak{x} \in \mathbb{R}^n \middle| \mathfrak{A}\mathfrak{x} \geq \mathfrak{b},\ \mathfrak{x} \geq 0\right\}$$

  heißt *Lösungsmenge* oder *Lösungsraum*.

- Mit $\mathfrak{x}^*_{LP} \in \mathcal{L}(\mathfrak{A}, \mathfrak{b})$ bezeichnet man die *optimale Lösung*, das heißt:

$$\mathfrak{c}^\mathsf{T}\mathfrak{x}^*_{LP} = \inf\left\{\mathfrak{c}^\mathsf{T}\mathfrak{x} \middle|\ \mathfrak{x} \in \mathcal{L}(\mathfrak{A}, \mathfrak{b})\right\}$$

Der Lösungsraum läßt sich geometrisch interpretieren, indem die einzelnen Restriktionen als Halbräume in einem n-dimensionalen Raum aufgefaßt werden. Somit ist die Lösungsmenge der Schnitt von m Halbräumen und n Nichtnegativitätsbedingungen. Eine solche Figur wird auch *Polyeder* genannt. Falls der Polyeder beschränkt ist, spricht man von einem *Polytop*.

Da Halbräume konvexe Mengen sind und der Schnitt konvexer Mengen wiederum eine konvexe Menge ergibt (siehe z. B. [14]), ist der Lösungsraum folglich auch eine konvexe Menge. Damit ist er insbesondere auch zusammenhängend.

Die Ecken der Lösungsmenge sind folgendermaßen definiert:

Definition 7.3. Eine Lösung $\mathfrak{x}$ ist eine *Ecke* der nichtleeren Lösungsmenge $\mathcal{L}(\mathfrak{A}, \mathfrak{b})$, wenn kein $\mathfrak{y} \neq 0$ existiert mit $\mathfrak{x} - \mathfrak{y},\ \mathfrak{x} + \mathfrak{y} \in \mathcal{L}(\mathfrak{A}, \mathfrak{b})$.

Anschaulich gesprochen, ist eine Ecke also ein Punkt, der nur Endpunkt von Strecken innerhalb des Lösungsraums sein kann und nicht auf deren Mitte zu liegen kommen kann (siehe auch Abbildung 7.1).

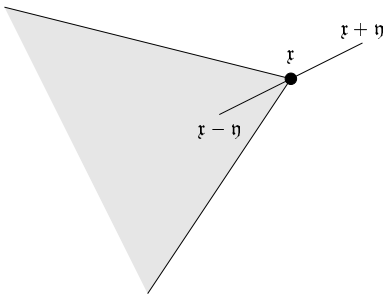


Abbildung 7.1: Ecke $\mathfrak{x}$ eines Polyeders

Falls der Lösungsraum leer ist, existiert keine optimale Lösung. Falls die Lösungsmenge derart unbeschränkt ist, daß $\inf\left\{\mathfrak{c}^\mathsf{T}\mathfrak{x} \middle|\ \mathfrak{x} \in \mathcal{L}(\mathfrak{A}, \mathfrak{b})\right\} = -\infty$, kann ebenfalls keine optimale Lösung existieren. Falls aber eine optimale Lösung existiert, so existiert auch eine optimale Ecke.

Satz 7.1. *Sei* $\mathcal{L}(\mathfrak{A}, \mathfrak{b}) \neq \emptyset$ *und* $\inf\left\{\mathfrak{c}^\mathsf{T}\mathfrak{x} \middle|\ \mathfrak{x} \in \mathcal{L}(\mathfrak{A}, \mathfrak{b})\right\}$ *endlich. Dann existiert eine Ecke, die optimal ist.*

Beweis. Siehe z. B. [31, S. 121-122]. □

Beispiel 7.1. Sei folgendes lineares Programm gegeben:

$$\begin{aligned}\max\ & x_1 + x_2, \text{ so daß}\\ & 2x_1 - x_2 \leq 4\\ & 2x_1 + x_2 \leq 7\\ & 2x_1 - 5x_2 \geq -10\end{aligned}$$

Die linearen Ungleichungen und der Lösungsraum sind in Abbildung 7.2 dargestellt.

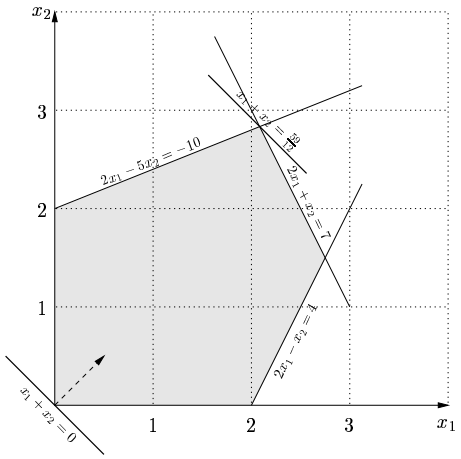


Abbildung 7.2: Lösungsraum von Beispiel 7.1

Dieses lineare Programm läßt sich umformulieren in die kanonische Form der linearen Programmierung.

$$\begin{aligned}-\min\ & -x_1 - x_2, \text{ so daß}\\ & -2x_1 + x_2 \geq -4\\ & -2x_1 - x_2 \geq -7\\ & 2x_1 - 5x_2 \geq -10\end{aligned}$$

Wenn man nun noch die Matrizenschreibweise anwendet, erhält man

$$\begin{aligned}-\min\ & \begin{pmatrix}-1\\-1\end{pmatrix}\begin{pmatrix}x_1\\x_2\end{pmatrix}, \text{ so daß}\\ & \begin{pmatrix}-2 & 1\\-2 & -1\\2 & -5\end{pmatrix}\begin{pmatrix}x_1\\x_2\end{pmatrix} \geq \begin{pmatrix}-4\\-7\\-10\end{pmatrix}\end{aligned}$$

Wie sich in diesem Beispiel noch anhand der grafischen Darstellung des Lösungsraums und der Berechnung des Eckpunktes erkennen läßt, liegt die

optimale Lösung bei

$$\mathfrak{x}^*_{LP} = \begin{pmatrix} x_1^* \\ x_2^* \end{pmatrix} = \begin{pmatrix} \frac{17}{6} \\ \frac{25}{12} \end{pmatrix}$$

Der zugehörige Optimalwert der Zielfunktion ist somit $\frac{59}{12}$.
Man beachte, daß die Ecken nicht ganzzahlig sind. Die Tatsache, daß alle Koeffizienten der Matrizen aus $\mathbb{Z}$ sind, ist dafür unerheblich.

Da die Matrizenschreibweise die Problemstellung unübersichtlicher macht, wird im folgenden hauptsächlich mit den linearen Gleichungen direkt gearbeitet und auch keine einheitliche Richtung der Ungleichungen ($\geq$, $\leq$) eingehalten.

Für die lineare Programmierung gibt es Verfahren, die in Polynomialzeit laufen, z. B. die Interior-Point-Methode mit $\mathcal{O}(n^3 L)$ (wobei L die Länge der Matrizen $\mathfrak{A}$, $\mathfrak{b}$ und $\mathfrak{c}$ in Binärdarstellung angibt).
Somit liegt also die Sprachenversion der linearen Programmierung (LP)

$$LP := \Big\{ (\mathfrak{A}, \mathfrak{b}, \mathfrak{c}, w) \in \mathbb{R}^{m \cdot n} \times \mathbb{R}^m \times \mathbb{R}^n \times \mathbb{R} \;\Big|\; \exists \mathfrak{x} \in \mathbb{R}^n : \; x \geq 0,\; \mathfrak{A}\mathfrak{x} \geq \mathfrak{b},\; \mathfrak{c}^{\mathsf{T}}\mathfrak{x} \leq w \Big\}$$

in $\mathcal{P}$.

7.2 Ganzzahliges lineares Programmieren

Um $PART^{\text{opt}}$ mit linearer Programmierung lösen zu können, muß das Verfahren ganzzahlige optimale Lösungen liefern. Das wird im allgemeinen aber nicht der Fall sein, da die Ecken nicht ganzzahlig sein müssen, wie in Beispiel 7.1 schon angedeutet.
Um also ganzzahlige Lösungen zu erhalten, muß man den Lösungsraum der linearen Programmierung weiter einschränken und fordern, daß die Lösungen ganzzahlig sind.

Definition 7.4. Die kanonische Form einer Aufgabe der *ganzzahligen linearen Programmierung* (IP) lautet:

$$\begin{aligned} &\min \mathfrak{c}^{\mathsf{T}}\mathfrak{x}, \text{ so daß} \\ &\quad \mathfrak{A}\mathfrak{x} \geq \mathfrak{b} \\ &\quad \mathfrak{x} \geq 0 \\ &\quad x_i \in \mathbb{Z} \quad \forall i = 1, \ldots, n \end{aligned} \tag{IP}$$

Die optimale Lösung von IP wird mit $\mathfrak{x}^*_{IP}$ bezeichnet [9].

Aufgrund der zusätzlichen Bedingung wird der Lösungsraum eines ganzzahligen linearen Programms kleiner als der des gleichen linearen Programms ohne Ganzzahligkeits-Bedingungen. Daraus ergibt sich direkt, daß

$$c^{\mathsf{T}}\mathfrak{x}^*_{IP} \geq c^{\mathsf{T}}\mathfrak{x}^*_{LP}$$

also die Kosten für die optimale Lösung eines linearen Programms niedriger sind als für die seines ganzzahligen Pendants.

Weiterhin ist zu bemerken, daß — falls die optimale Lösung $\mathfrak{x}^*_{LP}$ eines linearen Programms nur ganzzahlige Werte besitzt — dies auch die optimale Lösung $\mathfrak{x}^*_{IP}$ des gleichen ganzzahligen linearen Programms ist (also $\mathfrak{x}^*_{IP} = \mathfrak{x}^*_{LP}$). Diese Tatsache findet in fast allen Lösungsverfahren von *IP* Anwendung.

Man könnte vermuten, daß die zusätzliche Ganzzahligkeitsbedingung das Problem nicht schwieriger macht, da der Lösungsraum kleiner wird. Dem ist aber nicht so, da bei der linearen Programmierung die optimale Lösung in einer Ecke zu finden ist, während die ganzzahlige lineare Programmierung ein kombinatorisches Problem darstellt. D. h. schlimmstenfalls müßten (fast) alle Elemente der Lösungsmenge auf Optimalität überprüft werden.

Dies bestätigt auch der folgende Satz:

Satz 7.2. *Die ganzzahlige lineare Programmierung (IP) löst $\mathcal{NP}$-vollständige Probleme.*

Beweis. In Abschnitt 7.3 wird $PART^{\text{opt}}$ als ganzzahliges lineares Programm beschrieben. Da $PART^{\text{opt}}$ $\mathcal{NP}$-vollständig ist, gilt die Behauptung. □

Ein anderer Beweis findet sich z. B. in [24]. Dort wird der Beweis über die Anwendung der ganzzahligen linearen Programmierung auf *KNF-SAT* geführt.

Wenn man die ganzzahlige lineare Programmierung, ebenso wie *LP*, als Sprachenproblem

$$IP := \Big\{(\mathfrak{A}, \mathfrak{b}, \mathfrak{c}, w) \in \mathbb{R}^{m\cdot n} \times \mathbb{R}^m \times \mathbb{R}^n \times \mathbb{R} \;\Big|\; \exists \mathfrak{x} \in \mathbb{Z}^n : \; x \geq 0,\; \mathfrak{A}\mathfrak{x} \geq \mathfrak{b},\; \mathfrak{c}^{\mathsf{T}}\mathfrak{x} \leq w\Big\}$$

auffaßt, erhält man — zusammen mit der Tatsache, daß $IP \in \mathcal{NP}$ (da man die optimale Lösung „raten“ und in Polynomialzeit verifizieren kann) — somit die Aussage:

Satz 7.3. *IP ist $\mathcal{NP}$-vollständig.*

Das Ziel ist, ganzzahlige lineare Programme mit Hilfe der linearen Programmierung — die keine Ganzzahligkeit der Ergebnisse garantiert — zu lösen. Dafür muß der Begriff der *Relaxation* eingeführt werden.

Definition 7.5. Unter der *Relaxation* eines linearen Programms versteht man die Vergrößerung seines Lösungsraums durch das Entfernen von Nebenbedingungen bzw. Ersetzen von Nebenbedingungen durch Schwächere.

Das bedeutet, daß insbesondere die Überführung eines ganzzahligen linearen Programms in ein nichtganzzahliges lineares Programm durch das Entfernen der Ganzzahligkeitsbedingungen eine Relaxation darstellt. Da diese Relaxation von besonderer Bedeutung ist, erhält sie ihren eigenen Namen:

Definition 7.6. Die Relaxation der Ganzzahligkeitsbedingung eines ganzzahligen linearen Programms wird *LP-Relaxation* oder *Linearisierung* genannt.

Zur Lösung eines ganzzahligen linearen Programms liegt die Idee nahe, einfach auf die Ganzzahligkeitsforderung zu verzichten, das linearisierte Problem mittels linearer Programmierung zu lösen und anschließend die nichtganzzahligen Werte so zu runden, daß ein nahezu optimales Ergebnis herauskommt.

Dies wäre ein durchaus praktikabler Ansatz, wenn die Wertebereiche der Variablen groß sind und somit die Rundung nicht allzu sehr ins Gewicht fallen würde.

Es sind jedoch zwei Punkte zu beachten:

- Das Runden der Werte könnte aufgrund der Nebenbedingungen gar nicht möglich sein.
- Wenn die gerundete optimale Lösung in der Lösungsmenge liegt, kann sie von der optimalen Lösung der ganzzahligen linearen Programmierung weit entfernt sein.

Die folgenden zwei Beispiele verdeutlichen die in den beiden Bemerkungen angesprochenen Probleme.

Beispiel 7.2 (aus [9]). Sei folgendes ganzzahlige lineare Programm gegeben:

$$\begin{aligned} &\max\ 2x_1 + x_2, \text{ so daß} \\ &\quad 2x_1 - 15x_2 \geq -60 \\ &\quad 10x_1 - 9x_2 \leq 30 \\ &\quad x_1, x_2 \geq 0 \\ &\quad x_1, x_2 \in \mathbb{Z} \end{aligned} \tag{7.1}$$

Bei Vernachlässigung der Ganzzahligkeitsbedingung ergibt sich eine optimale Lösung von (Punkt P in Abbildung 7.3)

$$\mathfrak{x}^*_{LP} = \begin{pmatrix} x_1^* \\ x_2^* \end{pmatrix} = \begin{pmatrix} \frac{15}{2} \\ 5 \end{pmatrix}.$$

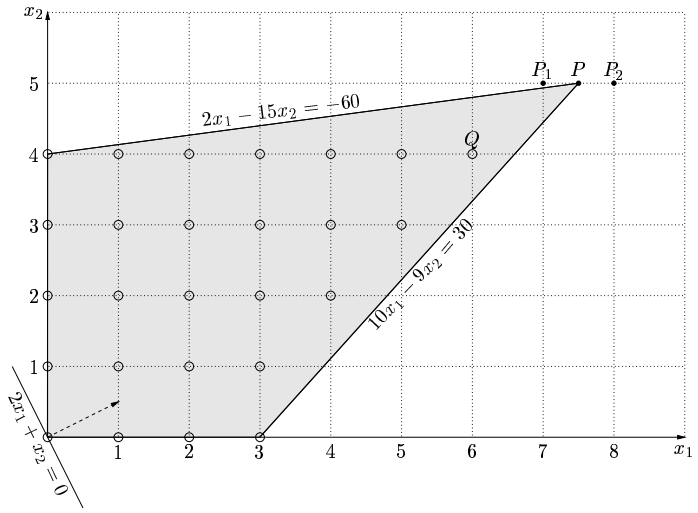


Abbildung 7.3: Lösungsraum des linearen Programms (7.1) (schattierte Fläche: linearisierte Lösung, Punkte: ganzzahlige Lösungen)

Somit würde man beim Runden von x_1^* die Lösungen $\hat{x}_1^* = 7$ oder $\hat{x}_1^* = 8$ erhalten, aber weder $\mathfrak{x}^{*\mathsf{T}} = (x_1^*, \hat{x}_2^*) = (7, 5)$ noch $\mathfrak{x}^{*\mathsf{T}} = (8, 5)$ (Punkte P_1 bzw. P_2 in Abbildung 7.3) erfüllen die Nebenbedingungen des ganzzahligen linearen Programms (7.1).

Vielmehr ist die optimale Lösung von (7.1) (Punkt Q in Abbildung 7.3)

$$\mathfrak{x}_{IP}^* = \begin{pmatrix} x_1^* \\ x_2^* \end{pmatrix} = \begin{pmatrix} 6 \\ 4 \end{pmatrix}.$$

Beispiel 7.3. Ein Beispiel für die zweite Bemerkung ist das folgende ganzzahlige lineare Programm

$$\begin{aligned} &\max\ x_1 + 5x_2, \text{ so daß} \\ &\quad x_1 + 6x_2 \leq 39 \\ &\quad 11x_1 + 4x_2 \leq 88 \\ &\quad x_1, x_2 \geq 0 \\ &\quad x_1, x_2 \in \mathbb{Z} \end{aligned} \tag{7.2}$$

Wenn man das Problem linearisiert, erhält man als optimale Lösung den Vektor ${\mathfrak{x}_{LP}^*}^{\mathsf{T}} = (x_1^*, x_2^*) = \left(6, \frac{11}{2}\right)$ (Punkt P in Abbildung 7.4). Somit ergäbe sich als gerundete optimale ganzzahlige Lösung der Vektor $\mathfrak{x}^{*\mathsf{T}} = (6, 5)$ (Punkt P_1 in Abbildung 7.4), der auch innerhalb des Lösungsraums liegt. Der zugehörige Wert der Zielfunktion ist 31.

Der Optimalwert des ganzzahligen linearen Programms hingegen ist 33 mit dem zugehörigen Lösungsvektor ${\mathfrak{x}_{IP}^*}^{\mathsf{T}} = (3, 6)$.

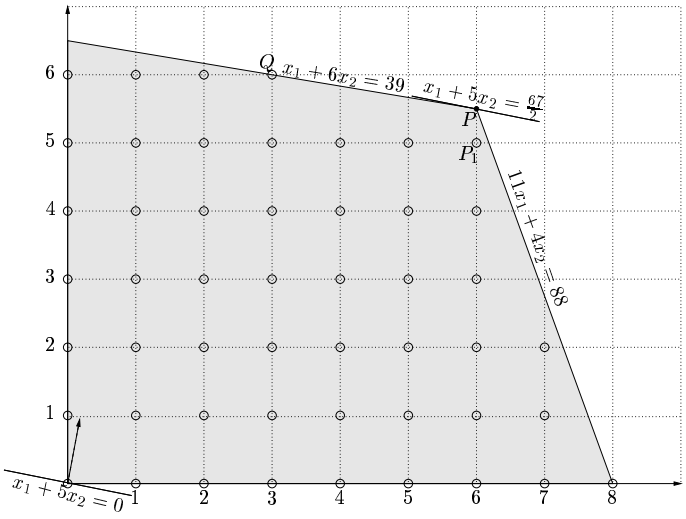


Abbildung 7.4: Lösungsraum des linearen Programms (7.2) (schattierte Fläche: ohne Ganzzahligkeitsbedingung, Punkte: ganzzahlige Lösungen)

Die Differenz zwischen tatsächlichem Optimum des ganzzahligen Programms und dem gerundeten Optimum des LP-relaxierten Programms läßt sich durch Hinzunahme weiterer Variablen und/oder Nebenbedingungen beliebig erhöhen.

Es gilt sogar:

Satz 7.4. *Jeder α-Approximationsalgorithmus (mit konstantem α) für IP ist $\mathcal{NP}$-hart.*

Beweis. In Abschnitt 7.3 wird $PART_m^{\text{opt}}$ (mit beliebiger linearer Abbildung m) als ganzzahliges lineares Programm beschrieben.

Da aber $PART_m^{\text{opt}}$ nach Satz 4.2 nicht mit einem Polynomialzeit-Algorithmus mit konstantem Verlustfaktor approximiert werden kann (sofern $\mathcal{P} \neq \mathcal{NP}$), kann insbesondere auch ein *beliebiges* ganzzahliges lineares Programm nicht in polynomieller Zeit mit konstantem Verlustfaktor approximiert werden. □

Die ganzzahlige lineare Programmierung ist sogar derart mächtig, daß auch gewisse nichtlineare Funktionen mit ihr beschrieben werden können (siehe z. B. [26]). Das ist der Grund, weshalb die LP-Relaxation auch als Linearisierung bezeichnet wird.

Beispiel 7.4. Eine Nebenbedingung eines ganzzahligen Programms laute

$$y = \max(x_1, x_2)$$

und die oberen Schranken von x_1 und x_2 seien o_1 bzw. o_2. In vielen Fällen sind die oberen Schranken der Variablen direkt aufgrund der Nebenbedingungen bekannt, ansonsten kann man sie mittels linearer Programmierung (ohne Ganzzahligkeitsforderung) leicht berechnen, indem man die Zielfunktion durch $\max x_1$ (bzw. der entsprechenden Variablen) ersetzt und die nichtlinearen Nebenbedingungen nicht berücksichtigt.

$y = \max(x_1, x_2)$ läßt sich auch beschreiben durch

$$y \geq x_1 \quad \text{and} \quad y \geq x_2$$

und

$$y \leq x_1 \quad \text{or} \quad y \leq x_2.$$

Da y gleich dem größeren Wert von x_1 und x_2 ist, gelten die Nebenbedingungen

$$y \geq x_1$$
$$y \geq x_2.$$

Weiterhin gilt $y \leq x_1$ oder $y \leq x_2$. Hierfür werden nun zwei zusätzliche Variablen $\hat{x}_1, \hat{x}_2 \in \{0, 1\}$ eingeführt, die beschreiben, welche der beiden Variablen x_1 und x_2 die Maximale ist. Da die oberen Schranken o_1 und o_2 von x_1 bzw. x_2 bekannt sind, läßt sich die Bedingung $y \leq x_1$ oder $y \leq x_2$ formulieren als

$$\hat{x}_1 \leq 1$$
$$\hat{x}_2 \leq 1$$
$$\hat{x}_1 + \hat{x}_2 = 1$$
$$y - o_1\hat{x}_1 \leq x_1$$
$$y - o_2\hat{x}_2 \leq x_2.$$

Die ersten drei Bedingungen belegen $\hat{x}_1$ und $\hat{x}_2$ so, daß genau eine der beiden Variablen 1 ist und die andere 0. Die letzten beiden Bedingungen nutzen die oberen Schranken der Variablen x_1 und x_2, um eine der beiden Bedingungen (und zwar die mit der nichterfüllbaren Schranke) trivial zu erfüllen.

Sofern $x_1 \neq x_2$, werden die beiden hinzugekommenen Variablen $\hat{x}_1$ und $\hat{x}_2$ eindeutig belegt, da ansonsten die Nebenbedingung $y - o_1\hat{x}_1 \leq x_1$ (falls $x_1 < x_2$) bzw. $y - o_2\hat{x}_2 \leq x_2$ (falls $x_1 > x_2$) nicht erfüllt werden kann.

Es ist also durch Hinzunahme von zwei Variablen und sieben neuen Neben-

bedingungen

$$\hat{x}_1 \leq 1$$
$$\hat{x}_2 \leq 1$$
$$\hat{x}_1 + \hat{x}_2 = 1$$
$$y \geq x_1$$
$$y \geq x_2$$
$$y - o_1\hat{x}_1 \leq x_1$$
$$y - o_2\hat{x}_2 \leq x_2$$

mit der ganzzahligen linearen Programmierung möglich, eine Nebenbedingung der Art $y = \max(x_1, x_2)$ zu benutzen.

Somit wäre es auch möglich, $PART^{\text{opt}}$ mit Hilfe der ganzzahligen linearen Programmierung bezüglich der Laufzeit zu optimieren.

7.3 Formulierung von $PART^{\text{opt}}$ als ganzzahliges lineares Programm

In Definition 2.15 wurde $PART^{\text{opt}}$ festgelegt durch die Funktion

$$PART_m^{\text{opt}}(\mathfrak{S}) := \min\Big\{ m(\mathfrak{S}, \mathfrak{l}) \Big| \ \mathfrak{l} \in \mathbb{l},\ d_{out}^{v,l_v} = d_{in}^{(v,w),l_{(v,w)}} \wedge d_{out}^{(v,w),l_{(v,w)}} = d_{in}^{w,l_w}\ \forall (v,w) \in E \Big\}.$$

Jede Bedingung dieser Funktion läßt sich nun als Nebenbedingung eines ganzzahligen linearen Programms formulieren.

Für jeden Funktionsblock und Wandler $k \in V \cup E$ existieren L_k verschiedene Implementierungsmöglichkeiten. Für jede dieser Implementierungsmöglichkeiten wird eine Variable $x_{k,l}$ (mit $l \in L_k$) zur Verfügung gestellt, die angibt, ob die Implementierung gewählt wird ($x_{k,l} = 1$) oder nicht ($x_{k,l} = 0$).

$$x_{k,l_k} \leq 1 \quad \forall k \in V \cup E,\ l_k \in L_k$$

Die Bedingung, daß alle Variablen ≥ 0 sein müssen, wird durch die Nichtnegativitätsbedingung garantiert. Somit können die Variablen aufgrund der Ganzzahligkeits-Forderung nur die Werte 0 oder 1 annehmen.

Da für jeden Block bzw. Wandler nur genau eine Implementierung gewählt werden darf, muß zusätzlich noch garantiert werden, daß nur eine Variable pro Block 1 ist, während die anderen 0 sind.

$$\sum_{l \in L_k} x_{k,l} = 1 \quad \forall k \in V \cup E$$

Zum Schluß muß noch erreicht werden, daß die Ausgangs-Zeitmodelldomänen der Blöcke (Wandler) mit den Eingangs-Zeitmodelldomänen der nachfolgenden Wandler (Blöcke) übereinstimmen. Dafür müssen die Zeitmodelldomänen in Konstanten kodiert werden.

Zeitmodelldomäne	Konstante
DTSS	0
DEVS	1
DESS	2

Die einzelnen Implementierungsmöglichkeiten der Blöcke und Wandler besitzen je eine Zeitmodelldomäne für den Ein- und den Ausgang. Da nur genau eine Implementierung pro Block (Wandler) gewählt werden kann, und die gewählte Implementierung kompatibel zur gewählten Implementierung des Nachbarwandlers (-blocks) sein soll, lassen sich die notwendigen Bedingungen folgendermaßen beschreiben:

$$\left.\begin{aligned} \sum_{l \in L_i} d_{out}^{i,l} x_{i,l} - \sum_{l \in L_{(i,j)}} d_{in}^{(i,j),l} x_{(i,j),l} = 0 \\ \sum_{l \in L_{(i,j)}} d_{out}^{(i,j),l} x_{(i,j),l} - \sum_{l \in L_j} d_{in}^{j,l} x_{j,l} = 0 \end{aligned}\right\} \quad \forall (i,j) \in E$$

wobei $d_{in}^{k,l}$ bzw. $d_{out}^{k,l}$ die Zahlen-Kodierungen der Zeitmodelldomänen der Block- und Wandler-Implementierungsmöglichkeiten bezeichnen.

Damit ist die Menge der gültigen Implementierungen von $PART_m^{\text{opt}}$ als Lösungsraum eines ganzzahligen linearen Programms beschrieben. Aber um eine bezüglich der linearen Funktion $m : \mathbb{S} \times \mathbb{l} \to \mathbb{R}$ optimale Lösung zu erhalten, muß m an das ganzzahlige Programm entsprechend angepaßt werden. Da ein bestimmter Systemgraph bereits vorliegt, hängt m nicht mehr vom Systemgraphen $\mathfrak{S}$ als Eingabe ab. Die Werte können stattdessen direkt in der Funktion angegeben werden. Andererseits wird die gewählte Implementierung des Graphen nicht auf die übliche Weise kodiert, sondern pro Block bzw. Wandler als One-Hot-Codierung.

Damit ergibt sich zum Beispiel für eine flächenoptimierende Kostenfunktion m (wobei — wie in Kapitel 6 bereits defininiert — die Fläche der Implementierung l des Blocks k durch $p^{k,l}$ angegeben werde) die Zielfunktion

$$\min \sum_{k \in V} \sum_{l_k \in L_k} p^{k,l_k} \, x_{k,l_k} + \sum_{(i,j) \in E} \sum_{l_{(i,j)} \in L_{(i,j)}} p^{(i,j),l_{(i,j)}} \, x_{(i,j),l_{(i,j)}}.$$

Beliebige andere lineare Kostenfunktionen lassen sich nach diesem Verfahren ebenfalls anpassen.

Zur Beschreibung von $PART^{\text{opt}}$ mit Hilfe der ganzzahligen linearen Programmierung benötigt man also $\sum_{k \in V \cup E} L_k$ verschiedene Variablen mit dem

Wertebereich $\{0,1\}$. Damit werden $\sum_{k\in V\cup E} L_k$ Nebenbedingungen benötigt, um die Beschränkung auf Werte ≤ 1 zu erhalten.

Zur Auswahl von genau einer Implementierung pro Block bzw. Wandler sind weitere $2\,(|V|+|E|)$ nötig. Der Vorfaktor 2 entsteht durch die Beschreibung der Nebenbedingungen als Ungleichungen.

Um die Kompatibilität der Zeitmodelldomänen von Ein- und Ausgang der Wandler und Blöcke zu garantieren, werden für jeden Wandler zwei Bedingungen gebraucht. Da sie wiederum mittels Ungleichungen beschrieben werden, ergeben sich so $4\,|E|$ Nebenbedingungen.

Insgesamt werden also

$$2\,|V| + 6\,|E| + \sum_{k\in V\cup E} L_k$$

Nebenbedingungen zur Beschreibung des Systemgraphen benötigt, plus die Ganzzahligkeits- und Nichtnegativitätsbedingungen für die $\sum_{k\in V\cup E} L_k$ Variablen.

Mit einer linearen Zielfunktion m (z. B. zur Minimierung der Fläche) läßt sich somit $PART_m^{\text{opt}}$ durch Zusammenfassen der benötigten Nebenbedingungen als ganzzahliges lineares Programm beschreiben.

$$\begin{aligned}
&\min \sum_{k\in V\cup E} \sum_{l_k\in L_k} p^{k,l_k}\, x_{k,l_k}, \text{ so daß} \\
&\sum_{l\in L_k} x_{k,l} \geq 1 && \forall k \in V\cup E \\
&-\sum_{l\in L_k} x_{k,l} \geq -1 && \forall k \in V\cup E \\
&\sum_{l\in L_i} d_{out}^{i,l} x_{i,l} - \sum_{l\in L_{(i,j)}} d_{in}^{(i,j),l} x_{(i,j),l} \geq 0 && \forall (i,j) \in E \\
&-\sum_{l\in L_i} d_{out}^{i,l} x_{i,l} + \sum_{l\in L_{(i,j)}} d_{in}^{(i,j),l} x_{(i,j),l} \geq 0 && \forall (i,j) \in E \\
&\sum_{l\in L_{(i,j)}} d_{out}^{(i,j),l} x_{(i,j),l} - \sum_{l\in L_j} d_{in}^{j,l} x_{j,l} \geq 0 && \forall (i,j) \in E \\
&-\sum_{l\in L_{(i,j)}} d_{out}^{(i,j),l} x_{(i,j),l} + \sum_{l\in L_j} d_{in}^{j,l} x_{j,l} \geq 0 && \forall (i,j) \in E \\
&-x_{k,l_k} \geq -1 && \forall k \in V\cup E,\ l_k \in L_k \\
&x_{k,l_k} \geq 0 && \forall k \in V\cup E,\ l_k \in L_k \\
&x_{k,l_k} \in \mathbb{Z} && \forall k \in V\cup E,\ l_k \in L_k
\end{aligned}$$

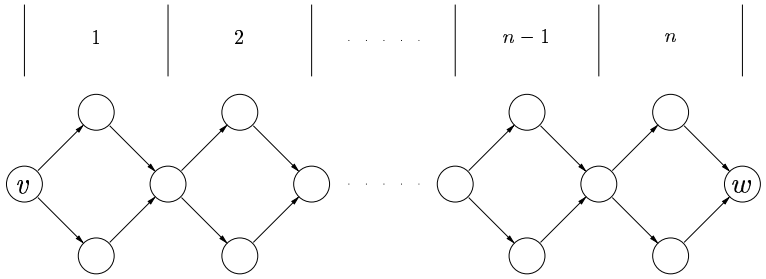


Abbildung 7.5: Graph mit exponentiell vielen Pfaden von v nach w

7.4 Zielfunktionen für $PART^{\mathbf{opt}}$

Lineare Kostenfunktionen, wie z. B. Funktionen zur Optimierung der Chipfläche, Leistungsverbrauch oder Fertigungskosten bzw. linearisierbare Kostenfunktionen, wie z. B. die Optimierung der Chipausbeute (siehe auch Abschnitt 2.7), lassen sich direkt als Zielfunktion der ganzzahligen linearen Programmierung darstellen.

Bei der Optimierung nach der Genauigkeit oder der Verzögerungszeit des Systems ist dies nicht so einfach möglich, da es sich hierbei nicht um lineare Funktionen handelt.

In Beispiel 7.4 wurde gezeigt, daß sich die Funktion max als ganzzahliges lineares Programm darstellen läßt. Die Gesamtverzögerungszeit des Systems berechnet sich aus der maximalen Verzögerungszeit *aller* Pfade von den Eingängen zu den Ausgängen des Systems. Mit einem Standard-Algorithmus (z. B. einem leicht modifizierten Single-Source-Shortest-Path-Algorithmus (siehe [18]) oder auch mit einem separaten linearen Programm (siehe [38])) könnte man die Verzögerungszeit effizient berechnen, da man viele Pfade aufgrund der bereits berechneten Teil-Verzögerungszeiten unberücksichtigt lassen kann. Da man aber die Formel zur Berechnung der Verzögerungszeit als Zielfunktion bereits vor dem Lösen des ganzzahligen Programms (also statisch) vorgeben muß, müssen grundsätzlich die Verzögerungszeiten aller Pfade miteinander verglichen werden.

Es existieren in einem Graphen unter Umständen jedoch exponentiell viele Pfade. Als Beispiel hierfür betrachte man den Graphen in Abbildung 7.5. Er besitzt $3n + 1$ Knoten und $4n$ Kanten. Aufgrund der speziellen Struktur existieren 2^n verschiedene Pfade von Knoten v nach Knoten w, also exponentiell viele.

Daher benötigt man im worst-case auch exponentiell viele Vergleiche (d. h. max-Operationen), woraus sich nach Beispiel 7.4 auch exponentiell viele zusätzliche Variablen und Nebenbedingungen ergeben. Es wird also nicht sinnvoll sein — auch wenn es theoretisch möglich ist — mit Hilfe der ganzzahligen linearen Programmierung $PART^{\mathrm{opt}}$ bezüglich der Laufzeit zu opti-

mieren.

Zusätzliche Randbedingungen, die sich als lineare Ungleichung formulieren lassen, können bei der Optimierung von $PART^{\text{opt}}$ berücksichtigt werden, indem sie als Nebenbedingungen in das lineare Programm eingefügt werden.

Wenn z. B. der maximale Flächenbedarf auf $\leq c_{\text{Fläche}}$ beschränkt werden soll, so ist dies durch die zusätzliche Nebenbedingung

$$\sum_{k \in V \cup E} \sum_{l_k \in L_k} p^{k,l_k}_{\text{Fläche}} \, x_{k,l_k} \leq c_{\text{Fläche}}$$

möglich.

Nichtlineare Randbedingungen (wie z. B. maximale Verzögerungszeit) lassen sich hingegen nicht, oder nur mit exponentiellem Aufwand in den Nebenbedingungen des ganzzahligen linearen Programms realisieren. Dies ergibt sich aus der Tatsache, daß eine zusätzliche Randbedingung letztendlich eine in den Nebenbedingungen versteckte Zielfunktion ist, deren Ergebnis einen bestimmten Wertebereich einzuhalten hat.

7.5 Lösungsmethoden für *IP*

Da das lineare Programm von $PART^{\text{opt}}$ nur Variablen mit Wertebereich $\{0, 1\}$ enthält, läßt sich die in Abschnitt 7.2 vorgeschlagene Heuristik, die sich der Rundung der Variablen des linearisierten Programms bedient, nicht sinnvoll realisieren.

Aufgrund der $\mathcal{NP}$-Vollständigkeit von *PART* (siehe Kapitel 3.2) ist auch nicht anzunehmen, daß man durch Linearisierung (bzw. Relaxation generell) ein polynomielles Lösungsverfahren, wie z. B. für total unimodulare Matrizen (siehe [38]) erhalten kann — außer falls $\mathcal{P} = \mathcal{NP}$.

Es gibt aber verschiedene Grund-Methoden zur exakten Lösung ganzzahliger linearer Programme, auf die sich auch die meisten Verfahren zurückführen lassen. Dies sind

- vollständige Enumeration
- Schnittebenen-Verfahren
- Branch & Bound-Verfahren
- Lagrange-Relaxation.

Unter der vollständigen Enumeration versteht man das systematische Belegen sämtlicher Variablen mit allen Werten ihres Wertebereichs. Dabei wird garantiert auch diejenige Variablenbelegung erreicht, die die Kostenfunktion minimiert. Das Verfahren benötigt recht wenig Speicherplatz, da nur die

aktuelle Belegung sowie die Kosten und Variablenbelegung für das bisherige Optimum gespeichert werden müssen. Allerdings beträgt die Laufzeit des Verfahrens $\mathcal{O}\left(\prod_{x \text{ Variable}} \text{card}(x)\right)$, wobei $\text{card}(x)$ die Größe des Wertebereichs von x angibt. Das bedeutet, daß für $PART^{\text{opt}}$ die vollständige Enumeration eine Laufzeit von $\mathcal{O}\left(2^{\sum_{k \in V \cup E} L_k}\right)$ benötigt, was das Verfahren für das vorliegende Problem zwar theoretisch anwendbar, aber praktisch unbrauchbar macht.

Bei Schnittebenen-Verfahren wird das ganzzahlige Problem als lineares Programm ohne die Ganzzahligkeitsforderungen gelöst. Für die Variablen, die im Ergebnis mit nichtganzzahligen Werten belegt sind, werden zusätzliche Nebenbedingungen in das Problem eingefügt, so daß die erhaltene Lösung nicht mehr im Lösungsraum des neuen Problems enthalten ist, andererseits aber der Lösungsraum des ganzzahligen Problems nicht beeinflußt wird. Dies wird solange wiederholt, bis man zu einem ganzzahligen Ergebnis gelangt.

Die Branch & Bound-Verfahren setzen ebenfalls auf der Lösung des Problems mit Hilfe der nichtganzzahligen linearen Programmierung auf. Nach der Berechnung des linearisierten Programms wird das Problem in zwei Teilprobleme zerlegt, falls das Ergebnis noch Variablen mit nichtganzzahliger Lösung besitzt. Sei x eine Variable, die mit dem nichtganzzahligen Wert a belegt wurde. Dann wird ein Teilproblem mit der zusätzlichen Nebenbedingung $x \leq \lfloor a \rfloor$ erzeugt, und ein anderes mit der zusätzlichen Nebenbedingung $x \geq \lceil a \rceil$. Diese werden rekursiv mit der nichtganzzahligen linearen Programmierung optimiert, bis das optimale ganzzahlige Ergebnis feststeht. Dabei kann man durch geschickte Auswahl des als nächsten zu bearbeitenden Teilproblems und dem Abtrennen irrelevanter Teilprobleme viel Zeit einsparen.

Das Ziel der Lagrange-Relaxation ist das Finden der optimalen ganzzahligen Lösung, indem ein Teil der Nebenbedingungen in die Zielfunktion eingebunden wird (siehe z. B. [38]). Die Lagrange-Relaxation liefert nicht zwangsläufig ein Ergebnis und erscheint ungeeignet für Probleme mit $\{0, 1\}$-wertigen Variablen. Daher wird sie hier nicht weiter betrachtet.

Da $PART^{\text{opt}}$ $\mathcal{NP}$-vollständig ist, werden alle Lösungsverfahren, die $PART^{\text{opt}}$ exakt lösen, im worst-case exponentielle Laufzeit benötigen. Das bedeutet aber nicht, daß das Problem nicht lösbar ist. Vielmehr lassen sich nur Systemgraphen kleiner Größe mit realistischem Zeitaufwand lösen. Mit Hilfe guter Verfahren ist es jedoch möglich, diese Größenschranke deutlich zu erhöhen.

7.5.1 Schnittebenen-Verfahren

Wie bereits in Abschnitt 7.2 beschrieben, liegt der Wert der optimalen Lösung eines linearisierten ganzzahligen Programms niedriger als das tatsächli-

che Optimum. Das ergab sich aus der Tatsache, daß die Ecken eines linearen Programms nicht ganzzahlig sein müssen.

Die konvexe Hülle conv $(\mathcal{L}_{IP}(\mathfrak{A}, \mathfrak{b}))$ des Lösungsraums $\mathcal{L}_{IP}(\mathfrak{A}, \mathfrak{b})$ besitzt einerseits nur ganzzahlige Ecken, andererseits sind alle ganzzahligen Elemente der konvexen Hülle auch Lösungen des ganzzahligen linearen Programms, also

$$\text{conv}\,(\mathcal{L}_{IP}(\mathfrak{A}, \mathfrak{b})) \cap \mathbb{Z}^n = \mathcal{L}_{IP}(\mathfrak{A}, \mathfrak{b}).$$

Zudem läßt sich die konvexe Hülle als Schnitt linearer Ungleichungen beschreiben.

Damit kann die Ganzzahligkeitsforderung entfallen, die optimale Lösung der konvexen Hülle mittels nichtganzzahliger linearer Programmierung berechnet werden und das Ergebnis entspricht der optimalen Lösung des ganzzahligen linearen Programms.

Im allgemeinen werden jedoch exponentiell viele Ungleichungen benötigt, um die konvexe Hülle eines ganzzahligen Lösungsraums zu beschreiben, wodurch diese Methode ineffizient wird (siehe [38]).

Es ist jedoch möglich, die linearen Ungleichungen der konvexen Hülle dem ganzzahligen linearen Programm erst bei Bedarf hinzuzufügen.

Die optimale Lösung des linearisierten Programms wird mit einem Standardverfahren berechnet. Erhält man einen nichtganzzahligen Lösungsvektor, so wird dem Problem eine lineare Ungleichung hinzugefügt, die einerseits den Lösungsraum des ganzzahligen linearen Programms nicht verändert, andererseits aber die gerade berechnete Lösung aus dem Lösungsraum ausschließt (zur Illustration siehe Abbildung 7.6). Diese Ungleichung wird *Schnittebene* genannt. Weiterhin sollte die neu hinzugefügte Schnittebene eine Kante der konvexen Hülle sein, so daß man möglichst schnell ganzzahlige Ecken erhält. Dies ist aber keine notwendige Bedingung.

Diese beiden Schritte werden nun so lange wiederholt, bis man eine ganzzahlige Lösung erhält oder das Problem unlösbar wird (dann existiert keine ganzzahlige Lösung).

Da die Lösungsmenge des ganzzahligen linearen Programms durch die zusätzlichen Schnittebenen nicht verändert wird, andererseits aber

$$\mathfrak{x}^*_{LP} \leq \mathfrak{x}^*_{IP}$$

gilt, ist die erhaltene ganzzahlige Lösung auch die optimale Lösung des ganzzahligen linearen Programms.

In Algorithmus 7.1 wird das Schnittebenenverfahren nochmals verdeutlicht.

Für die Berechnung geeigneter Schnittebenen bieten sich verschiedene Verfahren an, wie z. B. die Berechnung von Chvátal-Gomory-Ungleichungen.

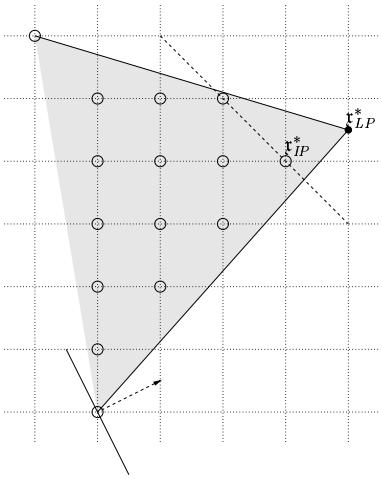


Abbildung 7.6: Schnittebene (gestrichelte Gerade) trennt berechnetes nichtganzzahliges Optimum $\mathfrak{x}^*_{LP}$ aus dem Lösungsraum heraus ($\mathfrak{x}^*_{IP}$ sei das tatsächliche, ganzzahlige Optimum)

Für tiefergehende Betrachtungen der Schnittebenenverfahren sei z. B. auf [26, 38, 33, 9] verwiesen.

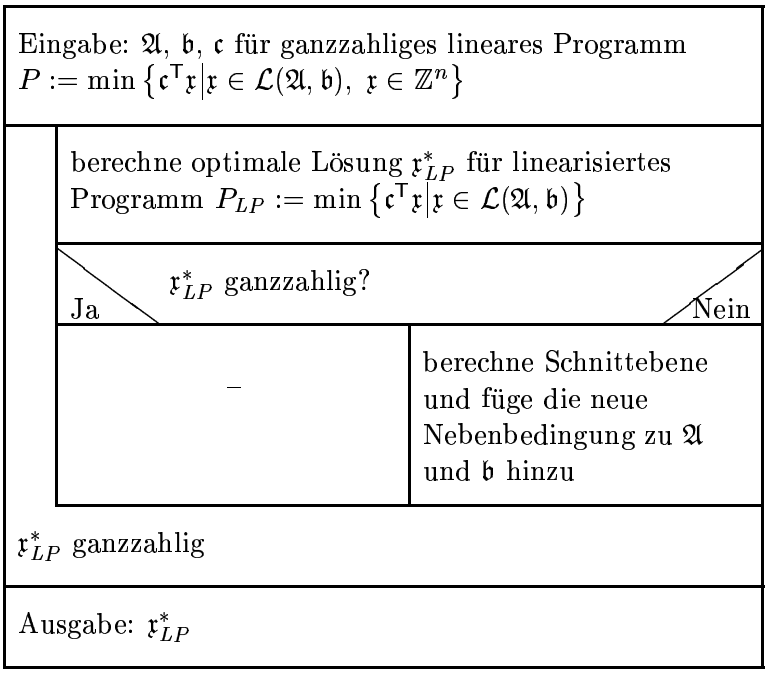
Anwendbarkeit auf $PART^{\mathbf{opt}}$

Die Endlichkeit des Schnittebenenverfahrens hängt stark von der Wahl geeigneter Schnittebenen ab. Die Chvátal-Gomory-Ungleichungen führen zu einem endlichen Verfahren, falls die Zielfunktion nur ganzzahlige Werte enthält. Eine solche Zielfunktion wäre für $PART^{\mathrm{opt}}$ erhältlich, indem die Kosten der einzelnen Implementierungsmöglichkeiten mit einem entsprechenden Faktor multipliziert werden.

Chvátal-Gomory-Ungleichungen sind nicht zwangläufig lineare Ungleichungen der konvexen Hülle des ganzzahligen Problems, sondern deutlich schwächere Ungleichungen. Ungleichungen der konvexen Hülle — sogenannte *strong inequalities* — würden eine schnelle Lösung des Problems garantieren. Die Berechnung der benötigten strong inequalities ist jedoch wiederum $\mathcal{NP}$-vollständig, so daß auch auf diese Weise keine effiziente Lösung zu erwarten ist.

Ob der Einsatz von Schnittebenenverfahren für das vorliegende Problem sinnvoll ist, kann — aufgrund der relativ schlechten theoretischen Erkenntnisse über das Laufzeitverhalten — nicht ohne ausführliche Testläufe mit $PART^{\mathrm{opt}}$-Problemen beurteilt werden.

Kommerzielle Optimierungstools benutzen keine Schnittebenenverfahren.



Algorithmus 7.1: Schnittebenenverfahren zur Lösung ganzzahliger linearer Programme

7.5.2 Branch & Bound-Verfahren

Branch & Bound-Verfahren hingegen werden in fast allen Optimierungstools benutzt, wobei diverse zusätzliche „Tricks“ und Heuristiken zur Effizienzsteigerung verwendet werden (siehe [26, 38]).

Im folgenden bezeichne P_{IP} das ganzzahlige lineare Programm und P_{LP} das linearisierte Programm von P_{IP}.

Die Idee des Branch & Bound-Verfahrens ist, daß man das gegebene Problem in mehrere Teilprobleme zerlegt und diese dann getrennt voneinander löst. Für die ganzzahlige lineare Programmierung bedeutet das, daß man den Lösungsraum in mehrere — hier zwei — Teile zerlegt, indem man einzelne Variablen auf bestimmte Wertebereiche einschränkt.

Die Wahl der einzuschränkenden Variablen wird getroffen, indem man das linearisierte Problem P_{LP} löst und sich eine Variable aussucht, die in der optimalen linearisierten Lösung keinen ganzzahligen Wert besitzt. Sei $\hat{x}$ die Variable und a ihre nichtganzzahlige Belegung.

Daraufhin wird das Problem P_{LP} in zwei Teilprobleme P^1_{LP} und P^2_{LP} unterteilt, die folgendermaßen definiert sind:

$$P^1_{LP} := P_{LP} \cap \{\hat{x} \leq \lfloor a \rfloor\}$$
$$P^2_{LP} := P_{LP} \cap \{\hat{x} \geq \lceil a \rceil\}$$

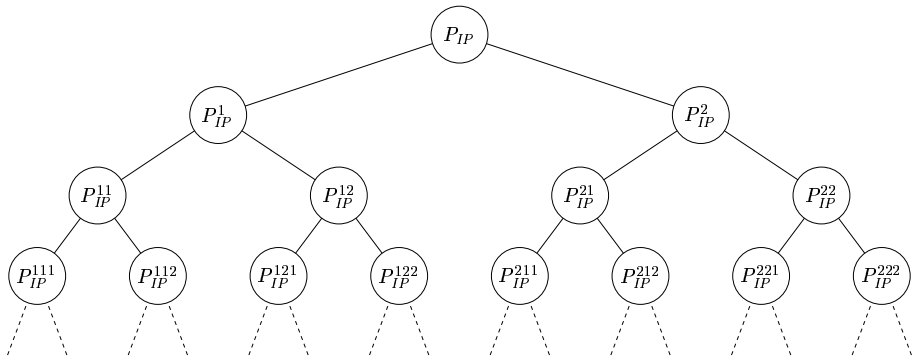


Abbildung 7.7: Von Branch & Bound aufgespannter Baum der Teilprobleme

Dieser Teil des Verfahrens ist der *Branch-Schritt.*

Der Lösungsraum $P^1_{LP} \cup P^2_{LP}$ der beiden Teilprobleme enthält weiterhin den gesamten Lösungsraum des ganzzahligen Problems, also

$$P^1_{IP} \cup P^2_{IP} = P_{IP}$$

und somit gilt für das optimale Ergebnis

$$\mathfrak{c}^\mathsf{T}\mathfrak{x}_{IP} = \min\{\mathfrak{c}^\mathsf{T}\mathfrak{x}^1_{IP}, \mathfrak{c}^\mathsf{T}\mathfrak{x}^2_{IP}\}. \tag{7.3}$$

Dieses Verfahren kann man nun rekursiv fortsetzen, bis jedes Teilproblem nur noch ganzzahlige optimale Lösungen besitzt bzw. unlösbar wird. Unter diesen ganzzahligen Lösungen läßt sich dann die minimale ganzzahlige Lösung finden.

Dieses Verfahren läßt sich gut als Baum repräsentieren (Abbildung 7.7), aus dem man auch sofort erkennt, daß das Verfahren exponentielle Laufzeit besitzen muß. Diesem exponentiellen Wachstum des Baumes versucht der *Bound-Schritt* entgegenzutreten, indem ganze Teilbäume entfernt werden, die die optimale Lösung nicht enthalten können.

Für jedes Teilproblem P^i gilt (Abschnitt 7.2)

$$\mathfrak{c}^\mathsf{T}\mathfrak{x}^{*^i}_{LP} \leq \mathfrak{c}^\mathsf{T}\mathfrak{x}^{*^i}_{IP}$$

Wenn also für ein beliebiges Teilproblem P^i_{IP} bereits die optimale Lösung $\mathfrak{x}^{*^i}_{IP}$ feststeht und für ein anderes linearisiertes Teilproblem P^j_{LP} die optimale Lösung $\mathfrak{x}^{*^j}_{LP} \geq \mathfrak{x}^{*^i}_{IP}$ ist, braucht dieses Teilproblem aufgrund von (7.3) nicht weiter betrachtet zu werden, da es keine bessere Lösung als die von Teilproblem P^i_{IP} liefern kann. Somit kann man diesen Teilbaum abschneiden.

Wenn man diese Methode geschickt anwendet, können große Teilbäume abgeschnitten werden, so daß das Verfahren auch für schwierige Probleme interessant bleibt. Dabei stellen sich zwei Fragen:

- Welche nichtganzzahlige Variable soll man im Branch-Schritt wählen?
- Welches Teilproblem soll man als nächstes betrachten, um möglichst schnell zu einem möglichst guten, ganzzahligen Ergebnis zu gelangen?

Wahl der Branch-Variablen

Das ganzzahlige lineare Programm von $PART^{\text{opt}}$ besitzt keine ausgezeichneten Variablen; und weil alle Variablen aus $\{0,1\}$ sind, wird die ausgewählte Variable im Endeffekt während des Branch-Schrittes auf ihren Wert festgelegt. Da ein möglichst ausgeglichener Baum wünschenswert ist, um Entartungen zu vermeiden, liegt es nahe, diejenige nichtganzzahlige Variable x_k^* zu wählen, die am nächsten bei $\frac{1}{2}$ liegt, also

$$\arg \max_{k \in V \cup E} \min\{x_k^*, 1 - x_k^*\}.$$

Diese Wahl der Branch-Variablen ist auch in der Praxis üblich ([38]).

Es ist jedoch fraglich, ob diese Strategie für $PART^{\text{opt}}$ geeignet ist. Für jeden Block und Wandler werden genausoviele Variablen benötigt, wie sie Implementierungsmöglichkeiten besitzen, pro Block bzw. Wandler ist aber nur eine Variable $= 1$ und die restlichen sind $= 0$. Dies wird für den Block k durch die Nebenbedingung

$$\sum_{l \in L_k} x_{k,l} = 1$$

garantiert.

Wenn nun einige der Variablen nicht ganzzahlig sind und eine von ihnen $(x_{k,j})$ als Branch-Variable gewählt wird, so wird das Problem aufgeteilt in die beiden Teilprobleme

$$P_{LP}^1 := P_{LP} \cap \{x_{k,j} = 0\}$$
$$P_{LP}^2 := P_{LP} \cap \{x_{k,j} = 1\}.$$

Damit wird in Teilproblem P_{LP}^1 nur eine Variable belegt, während durch Teilproblem P_{LP}^2 L_k Variablen festgelegt werden. Auf diese Weise ergibt sich ein stark unbalancierter Baum.

Einen balancierteren Baum erhält man mittels sogenanntem *GUB-Branching* (Generalized Upper Bound; siehe auch [38]). Die Aufteilung des Problems in die Teilprobleme erfolgt hier durch das Schema

$$P_{LP}^1 := P_{LP} \cap \{x_{k,i} = 0 : \ i = 1, \ldots, r\}$$
$$P_{LP}^2 := P_{LP} \cap \{x_{k,i} = 0 : \ i = r+1, \ldots, L_k\}$$

wobei $r = \min\left\{t : \sum_{i=1}^{t} x^*_{k,i} \geq \frac{1}{2}\right\}$ ist. P^1_{LP} fordert also, daß für Block k eine Implementierungsmöglichkeit aus $\{r+1, \ldots, L_k\}$ gewählt wird, während P^2_{LP} nur Implementierungsmöglichkeiten aus $\{1, \ldots, r\}$ für Block k zuläßt.

Es ist zu erwarten, daß GUB-Branching einen deutlich balancierteren Baum für $PART^{\text{opt}}$ erzeugt, als die Standard-Methode. Außerdem wird der Baum deutlich weniger Knoten, d. h. Teilprobleme besitzen, weil pro Verzweigungsschritt mehrere Variablen gleichzeitig gebunden werden.

Wahl des nächsten zu bearbeitenden Teilproblems

Ziel ist es, möglichst schnell zu einem Blatt des Baumes, d. h. einer ganzzahligen optimalen Lösung zu gelangen, denn erst wenn man eine ganzzahlige Lösung besitzt, kann man den Bound-Schritt anwenden. Daher ist eine Breitensuch-Strategie völlig ungeeignet — vielmehr wird eine Tiefensuch-Strategie zu wählen sein.

Nun kann man noch zwischen zwei Kindern, d. h. Teilproblemen auswählen, welches zuerst bearbeitet werden soll. Da man hoffen kann, daß der Knoten mit der kleineren unteren Schranke $\mathfrak{c}^\mathsf{T}\mathfrak{x}^{*^i}_{LP}$ derjenige ist, der auch das kleinere ganzzahlige Optimum $\mathfrak{c}^\mathsf{T}\mathfrak{x}^{*^i}_{IP}$ besitzt, wird man diesen als erstes betrachten. Außerdem wird ein Teilbaum mit größerer Wahrscheinlichkeit abgeschnitten, je größer seine untere Schranke ist. Diese Strategie heißt *Best-Node-First*.

Dies ist eine Standardmethode für das Branch & Bound-Verfahren auf ganzzahligen linearen Programmen, die auch für $PART^{\text{opt}}$ sinnvoll erscheint.

Wenn man auf die ganzzahlige lineare Programmierung verzichtet und ein problemspezifisches Branch & Bound-Verfahren entwickelt, ergeben sich eine Reihe weiterer Methoden zur Aufteilung in Teilprobleme und zur Wahl des nächsten Teilproblems. Diese werden in Kapitel 8 vorgestellt.

7.5.3 Preprocessing

Angenommen ein Funktionsblock besitzt nur eine einzige Implementierungsmöglichkeit. Dann wäre es prinzipiell gar nicht notwendig, daß für diese eine Implementierungsmöglichkeit eine Variable benutzt wird, die grundsätzlich 1 sein muß. Außerdem wäre es nicht notwendig, die Zeitmodelldomänen der benachbarten Wandler mit der Zeitmodelldomäne des Funktionsblocks zu vergleichen.

Stattdessen benötigt man für diesen Funktionsblock keine Nebenbedingung und keine Variable, weil die Ein- bzw. Ausgangszeitmodelldomänen der benachbarten Wandler konstant auf die Zeitmodelldomäne des Funktionsblocks gesetzt werden können. Weiterhin kann man die nicht in Frage kommenden Implementierungsmöglichkeiten der Wandler schon vor der Berechnung herausfiltern.

Wenn man bzgl. einer linearen Kostenfunktion, wie z. B. der Minimierung der Fläche, optimieren will, ist es unter Umständen gar nicht notwendig, alle Implementierungsmöglichkeiten der Blöcke zu berücksichtigen. Ein Block habe z. B. zwei Implementierungsmöglichkeiten innerhalb einer Zeitmodelldomäne. Dann ist es nicht notwendig, die Implementierungsmöglichkeit mit dem größeren Flächenbedarf zu berücksichtigen, denn diese kann nicht in der optimalen Partitionierung enthalten sein. Auf diese Weise kann man die Anzahl der Implementierungsmöglichkeiten für Blöcke auf drei und für Wandler auf neun beschränken (siehe auch Kapitel 6.2).

Durch ein solches Preprocessing kann man je nach gegebenem Systemgraphen einige bis viele Variablen und Nebenbedingungen schon vor dem Berechnen der optimalen Lösung des ganzzahligen linearen Programms eliminieren, und somit die Laufzeit erheblich verringern.

Kapitel 8

Branch & Bound

In Kapitel 7.5.2 wurde das Branch & Bound-Verfahren zur Lösung ganzzahliger linearer Programme benutzt. Dabei wird jedoch die Struktur des vorliegenden Problems nicht oder nur unzulänglich berücksichtigt. Dabei ist Branch & Bound prädestiniert, um an spezielle Eigenschaften des zu lösenden Problems angepaßt zu werden.

8.1 Grundlagen

Wie der Name schon andeutet, besteht das Verfahren aus zwei grundlegenden Schritten, nämlich dem Verzweigen in Teilprobleme und dem Abschätzen der einzelnen Teilprobleme. Dabei wird aber nicht das Problem selbst, sondern dessen Lösungsraum unterteilt und für die einzelnen Teilprobleme die jeweils optimale Lösung (rekursiv) bestimmt. Die beste der optimalen Teilproblem-Lösungen ist dann das globale Optimum des gegebenen Problems.

Sei S der Lösungsraum eines Minimierungsproblems P. s^* bezeichne die Kosten der optimalen Lösung $\mathfrak{x}^*$ von P und $\underline{s} \leq s^*$ eine untere Schranke der optimalen Kosten für P. Während sich $\underline{s}$ effizient berechnen läßt, benötige die Berechnung der optimalen Lösung $\mathfrak{x}^*$ und der zugehörigen minimalen Kosten s^* exponentielle Zeit (sonst macht die Anwendung von Branch & Bound keinen Sinn, da ein effizientes Lösungsverfahren für P existiert).

Nun läßt sich der Lösungsraum S in mehrere kleinere Teil-Lösungsräume $S_1, \ldots, S_n$ aufteilen, so daß

$$S = S_1 \cup \ldots \cup S_n$$

gilt. Weiterhin wäre es sinnvoll, wenn auch nicht notwendig, daß die Teil-Lösungsräume paarweise disjunkt zueinander ($S_i \cap S_j = \emptyset \ \forall i, j = 1, \ldots, n$, $i \neq j$) und ungefähr gleichmächtig ($|S_i| \approx |S_j| \ \ \forall i, j = 1, \ldots, n$) sind.

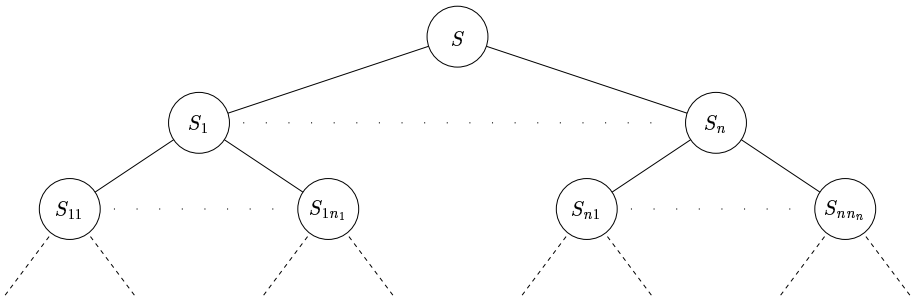


Abbildung 8.1: Von Branch & Bound aufgespannter Baum

Dieses Verzweigen kann man nun rekursiv für die Teilprobleme $S_1, \ldots, S_n$ fortführen, bis es sinnvoll erscheint, das Teilproblem nicht mehr weiter aufzuspalten, sondern direkt zu lösen. Man erhält also eine Baumstruktur (siehe Abbildung 8.1). Dabei entspricht ein Knoten einem Problem und dessen Kinder den Teilproblemen, die gelöst werden müssen, bevor das Problem selbst gelöst werden kann.

Da der Lösungsraum S des ursprünglichen Problems von den Lösungsräumen $S_1, \ldots, S_n$ der erzeugten Teilprobleme überdeckt wird, gilt für die Kosten der optimalen Lösung von P

$$s^* = \min\{s_1^*, \ldots, s_n^*\}. \tag{8.1}$$

Entsprechend ist die optimale Lösung $\mathfrak{x}^*$ gleich der optimalen Lösung des Teilproblems mit den geringsten Kosten.

Da die Anzahl der Teilprobleme im allgemeinen exponentiell zur Problemgröße ist, ist es sinnvoll, daß nicht-optimale Teilprobleme bereits frühzeitig erkannt und eliminiert werden, damit diese erst gar nicht weiter zerlegt und untersucht werden müssen.

Wenn die optimalen Kosten s_i^* für das Teilproblem P_i bereits berechnet sind, so gilt nach 8.1

$$s^* \leq s_i^*.$$

Das bedeutet, daß s_i^* eine obere Schranke für P darstellt ($\overline{s} := s_i^*$). Die obere Schranke $\overline{s}$ entspricht immer den Kosten der günstigsten bisher gefundenen Lösung.

Wenn nun ein anderes Teilproblem P_j eine untere Schranke $\underline{s}_j$ besitzt, die größer ist, als die obere Schranke $\overline{s}$ des Problems P (also $\underline{s}_j > \overline{s}$), dann braucht das Teilproblem P_j gar nicht erst gelöst zu werden, da es die optimale Lösung nicht enthalten kann. Teilbäume, die auf diese Weise abgeschnitten werden, bezeichnet man auch als *ausgelotet* [32].

Um möglichst viele Teilprobleme ausloten zu können, ist es notwendig, daß

1. das Problem in Teilprobleme zerlegt wird, die sich in ihrer Güte stark unterscheiden,

2. sehr schnell obere Schranken gefunden werden, die nahe am Optimum des ursprünglichen Problems liegen und

3. die unteren Schranken sehr nahe am Optimum des jeweiligen Teilproblems liegen.

Die Zerlegung des Problems in die Teilprobleme ist entscheidend für das Ausloten der einzelnen Teilbäume. Denn wenn die Teilprobleme nahezu die gleichen optimalen Kosten besitzen, liegen ihre unteren Schranken vermutlich unter den bisherigen optimalen Kosten. Somit können sie nicht ausgelotet werden, sondern müssen alle untersucht und miteinander verglichen werden.

Für den zweiten Punkt ist es zwar auch wichtig, eine geschickte Aufteilung des Problems in die einzelnen Teilprobleme zu finden, aber noch wichtiger ist es, auch eine gute Strategie zu finden, um herauszubekommen, welches Teilproblem als nächstes betrachtet werden sollte, um möglichst schnell möglichst exakte obere Schranken zu erhalten.

Der letzte Punkt ist fast noch wichtiger als die anderen beiden, denn wenn man nur schwache untere Schranken für die Teilprobleme berechnen kann, kann man nur wenige Teilbäume ausloten, so daß man letztendlich fast den ganzen Baum untersuchen muß — auch wenn man eine geschickte Zerlegung des Problems und gute obere Schranken findet.

8.2 Anwendung auf $PART^{opt}$

Aufgrund der speziellen Struktur des Systemgraphen, nämlich daß jeder Block und Wandler mehrere verschiedene Implementierungsmöglichkeiten besitzt, ist es naheliegend, den Systemgraphen in mehrere Teilgraphen zu zerlegen, bei denen jeweils nur eine Implementierungsmöglichkeit eines ausgewählten Blockes zugelassen wird. Dieser Block wird auch als *Branch-Variable* bezeichnet, da durch ihn das Problem in mehrere Teilprobleme zerlegt wird, und sich so an dieser Stelle der von Branch & Bound erzeugte Baum verzweigt.

Sei z. B. $\mathfrak{S}$ der Systemgraph und k der gewählte Block (mit $|L_k|$ verschiedenen Implementierungsmöglichkeiten). Dann wird der Systemgraph in $|L_k|$ verschiedene Teilprobleme $\mathfrak{S}_1, \ldots, \mathfrak{S}_{|L_k|}$ zerlegt, wobei in jedem Teilproblem der Block k nur genau eine Implementierungsmöglichkeit besitzt, der

restliche Systemgraph aber unverändert bleibt:

$$\mathfrak{S}_i := (G, \mathfrak{B}_i) \text{ mit } \mathfrak{B}_i := \left(B_v^i\right)_{v \in V \cup E}$$
$$\text{wobei } B_v^i := \begin{cases} B_v \text{ falls } v \in V \cup E \setminus \{k\} \\ \{B_{v,i}\} \text{ falls } v = k \end{cases}$$

Es ist natürlich nicht notwendig, daß für jedes Teilproblem ein kompletter Systemgraph gespeichert werden muß. Es reicht aus, den einzelnen Teilproblemen die noch verfügbaren Implementierungsmöglichkeiten für jeden Funktionsblock zuzuordnen.

Es werden zur Zerlegung in die Teilprobleme nur Funktionsblöcke ausgewählt, da die optimale Implementierung eines Wandlers (bei der Einzelzieloptimierung mittels einer monotonen Funktion) aufgrund seiner benachbarten zwei Funktionsblöcke eindeutig festgelegt wird.

Weil die Unterteilung des Systemgraphen durch das Fixieren der Implementierungsmöglichkeiten der Branch-Variablen geschieht, wird der aufgespannte Baum zwar einerseits sehr breit, andererseits ist der Baum ausbalanciert und relativ flach; er erreicht höchstens Tiefe $|V|$, also gleich der Anzahl der Funktionsblöcke.

Andere Unterteilungen des Systemgraphen in Teilgraphen (wie z. B. die in Kapitel 7.5.2 aufgezeigte binäre Unterteilung) erscheinen nicht sinnvoll, da sie nur ungenügend auf die Graphenstruktur eingehen und so der Systemgraph pro Branch-Schritt nicht ausreichend vereinfacht wird.

Das Abweichen von der Zerlegung in Teilprobleme durch Eliminieren von Implementierungsmöglichkeiten (also z. B. Aufteilung aufgrund von speziellen Kostenfunktionen o. ä.) ist auch nicht praktikabel, da die einzelnen Teilprobleme dann u. U. nicht mehr strukturell zusammenhängend sind, und außerdem die Berechnung der oberen und unteren Schranken sehr aufwendig werden kann.

8.2.1 Wahl der Branch-Variablen

Es stellt sich jedoch die Frage, welcher Block des Systemgraphen gewählt werden sollte, um die Teilgraphen zu erzeugen.

Wie in Abschnitt 8.1 bereits erwähnt, soll mit einer geschickten Wahl der Branch-Variablen erreicht werden, daß der Systemgraph in Teilgraphen zerlegt wird, die sich deutlich in ihren optimalen Kosten unterscheiden.

Es sind verschiedene Auswahlverfahren denkbar:

- wähle Block mit den wenigsten Implementierungsmöglichkeiten (*take-smallest*)
- wähle Block mit den höchsten maximalen Kosten (*take-max*)

- wähle Block mit dem größten Optimierungspotential (*take-best*)

Wenn man den Block mit den wenigsten Implementierungsmöglichkeiten (take-smallest) wählt, wird man einen Branch & Bound-Baum erhalten, der nahe der Wurzel wenig verzweigt ist (also einen kleinen Grad besitzt), in der Tiefe aber immer breiter wird. Wenn man erwartet, daß man bereits frühzeitig, d. h. nahe der Wurzel Teilbäume ausloten kann, erscheint diese Wahlstrategie sinnvoll. Jedoch existieren pro Block nur drei Zeitmodelldomänen, so daß für monotone Kostenfunktionen (die der Normalfall sind) auch nur drei Implementierungsmöglichkeiten betrachtet werden müssen (siehe Kapitel 6.2). Damit ist die Auswahl der Blöcke bezüglich ihrer Implementierungsmöglichkeiten sicher nicht praktikabel. Außerdem wird diese Strategie der Forderung nach Teilgraphen mit möglichst unterschiedlichen optimalen Kosten nicht nachkommen.

Die take-max-Strategie erscheint sinnvoller, da die Blöcke mit hohen Implementierungskosten auch den Hauptanteil an den Gesamtkosten besitzen. Jedoch ist es ungeschickt, einen Block zu wählen, der nur Implementierungsmöglichkeiten besitzt, die alle nahezu genauso teuer sind. Denn dann wird wiederum die Forderung nach unterschiedlich teuren Teilproblemen nicht erfüllt.

Wenn man stattdessen den Block mit dem größten Optimierungspotential wählt, erhält man Teilprobleme mit größtmöglichen Unterschieden bezüglich der Implementierungskosten. Das Optimierungspotential o_k des Blockes k läßt sich auf verschiedene Arten messen. Eine Möglichkeit ist, die Differenz zwischen der teuersten und der günstigsten Implementierung zu betrachten

$$o_k := \max_{l \in L_k} p^{k,l} - \min_{l \in L_k} p^{k,l}. \tag{8.2}$$

Eine andere Möglichkeit wäre, den Quotienten zwischen teuerster und günstigster Implementierung als Maß zu benutzen:

$$o_k := \frac{\max_{l \in L_k} p^{k,l}}{\min_{l \in L_k} p^{k,l}}. \tag{8.3}$$

Beide Maße haben ihre Berechtigung, denn falls alle Implementierungsmöglichkeiten eines Blockes hohe Kosten verursachen, kann Gleichung (8.3) recht klein sein, während Gleichung (8.2) das Optimierungspotential besser widerspiegelt. Bei geringen Implementierungskosten verhält es sich genau umgekehrt. Daher scheint es sinnvoll, die beiden Maße in geeigneter Art und Weise miteinander zu kombinieren (z. B. durch ein gewichtetes arithmetisches oder geometrisches Mittel).

Man kann die Maße auch noch weiter verfeinern, indem man die Kosten der benachbarten Wandler mit einbezieht. Auch hierfür gibt es wieder verschiedene Varianten. Man könnte zum Beispiel stets die Kosten für die

günstigsten kompatiblen Wandler berücksichtigen (hier für lineare Kostenfunktionen):

$$o_k := \frac{\max_{l \in L_k} \left\{ p^{k,l} + \underline{w}_k^l \right\}}{\min_{l \in L_k} \left\{ p^{k,l} + \underline{w}_k^l \right\}}$$

$$\text{mit } \underline{w}_k^l := \sum_{(i,k) \in E} \min_{\substack{l_{(i,k)} \in L_{(i,k)} \\ d_{in}^{k,l} = d_{out}^{i,l_{(i,k)}}}} p^{(i,k),l_{(i,k)}} + \sum_{(k,j) \in E} \min_{\substack{l_{(k,j)} \in L_{(k,j)} \\ d_{out}^{k,l} = d_{in}^{j,l_{(k,j)}}}} p^{(k,j),l_{(k,j)}}$$

Im allgemeinen werden die Wandler, deren Ein- und Ausgangs-Zeitmodelldomänen identisch sind, keine Kosten verursachen, so daß $\underline{w}_k^l$ üblicherweise 0 sein wird.

Andererseits könnten Maße, die die maximalen Kosten der Nachbarwandler berücksichtigen, wie z. B.

$$o_k := \frac{\max_{l \in L_k} \left\{ p^{k,l} + \overline{w}_k^l \right\}}{\min_{l \in L_k} \left\{ p^{k,l} + \overline{w}_k^l \right\}}$$

$$\text{mit } \overline{w}_k^l := \sum_{(i,k) \in E} \max_{\substack{l_{(i,k)} \in L_{(i,k)} \\ d_{in}^{k,l} = d_{out}^{i,l_{(i,k)}}}} p^{(i,k),l_{(i,k)}} + \sum_{(k,j) \in E} \max_{\substack{l_{(k,j)} \in L_{(k,j)} \\ d_{out}^{k,l} = d_{in}^{j,l_{(k,j)}}}} p^{(k,j),l_{(k,j)}}$$

oder sogar

$$o_k := \frac{\max_{l \in L_k} \left\{ p^{k,l} + \overline{w}_k^l \right\}}{\min_{l \in L_k} \left\{ p^{k,l} + \underline{w}_k^l \right\}}$$

das Optimierungspotential ebenfalls nicht korrekt widergeben.

Vermutlich erzielt man mit den gemittelten Kosten der Zeitmodell-kompatiblen Wandlerblock-Implementierungsmöglichkeiten die geeignetsten Werte für das Optimierungspotential. Z. B.[1]

$$o_k := \frac{\max_{l \in L_k} \left\{ p^{k,l} + \hat{w}_k^l \right\}}{\min_{l \in L_k} \left\{ p^{k,l} + \hat{w}_k^l \right\}}$$

$$\text{mit } \hat{w}_k^l := \sum_{(i,k) \in E} \frac{\sum_{\substack{l_{(i,k)} \in L_{(i,k)} \\ d_{in}^{k,l} = d_{out}^{i,l_{(i,k)}}}} p^{(i,k),l_{(i,k)}}}{\sum_{\substack{l_{(i,k)} \in L_{(i,k)} \\ d_{in}^{k,l} = d_{out}^{i,l_{(i,k)}}}} 1} + \sum_{(k,j) \in E} \frac{\sum_{\substack{l_{(k,j)} \in L_{(k,j)} \\ d_{out}^{k,l} = d_{in}^{j,l_{(k,j)}}}} p^{(k,j),l_{(k,j)}}}{\sum_{\substack{l_{(k,j)} \in L_{(k,j)} \\ d_{out}^{k,l} = d_{in}^{j,l_{(k,j)}}}} 1}$$

[1] Die Summen im Nenner von $\hat{w}_k^l$ entsprechen der Summe der kompatiblen Implementierungsmöglichkeiten aller benachbarten Wandler.

8.2.2 Berechnung der unteren Schranke

Die Genauigkeit der unteren Schranke der einzelnen Teilprobleme ist für die Laufzeit des Branch & Bound-Algorithmus von großer Bedeutung. Je schwächer die untere Schranke ist (d. h. je weiter sie von dem tatsächlichen Optimum entfernt ist), desto unwahrscheinlicher wird es, daß ein Teilgraph ausgelotet wird, weil man nicht erkennt, daß er die optimale Lösung nicht enthalten kann. Andererseits darf die Berechnung der unteren Schranke nicht allzuviel Zeit in Anspruch nehmen, da sie für jedes einzelne Teilproblem ausgeführt werden muß, um entscheiden zu können, ob es ausgelotet werden kann, oder nicht.

Einfache untere Schranke

Die einfachste Möglichkeit, eine untere Schranke der Kosten eines Systemgraphen zu berechnen (bei einer linearen Kostenfunktion), besteht darin, daß man die minimalen Kosten der einzelnen Funktionsblöcke und Wandler addiert, ohne die Kompatibilität ihrer Zeitmodelldomänen zu berücksichtigen. Also

$$\underline{s} := \sum_{k \in V \cup E} \min_{l_k \in L_k} p^{k,l_k}.$$

Diese untere Schranke läßt sich in $\mathcal{O}(|V|+|E|+L)$ (wobei $L := \sum_{k \in V \cup E} |L_k|$ die Summe aller Implementierungsmöglichkeiten bezeichne) berechnen, wie direkt aus der Formel ersichtlich ist. Im allgemeinen wird $\underline{s}$ jedoch relativ weit von den tatsächlichen minimalen Kosten des Systemgraphen entfernt sein, da die minimalen Kosten für die Wandler meistens 0 (für Implementierungen mit Eingangzeitmodelldomäne = Ausgangszeitmodelldomäne) sind und somit im Endeffekt nur die minimalen Kosten der Funktionsblöcke berücksichtigt werden.

Untere Schranke durch maximales Matching

Anstatt die untere Schranke aus den minimalen Kosten der jeweiligen Funktionsblöcke und Wandler zu berechnen, ohne die Kompatibilität der Ein- und Ausgangs-Zeitmodelldomänen zu berücksichtigen, kann man (bei linearen Kostenfunktionen) eine „größere Granularität“ wählen, indem man zwei benachbarte Funktionsblöcke und den dazwischenliegenden Wandler zusammenfaßt. Für diese Einheiten kann man nun die Kosten bestimmen, die sie minimal benötigen, wenn die entsprechenden Ein- und Ausgangs-Zeitmodelldomänen der benachbarten Blöcke bzw. Wandler zueinander passen. Diese minimalen Kosten für die Blöcke i und j sowie für den entspre-

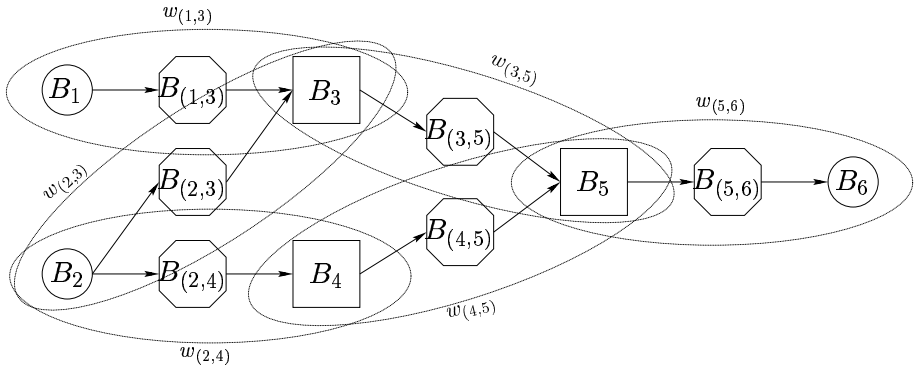


Abbildung 8.2: Block-/Wandlermengen, aus denen sich die minimalen Teil-Kosten $w_{(i,j)}$ berechnen

chenden Wandler (i,j) werden mit $w_{(i,j)}$ bezeichnet[2].

$$w_{(i,j)} := \min_{\substack{l_i \in L_i,\, l_j \in L_j \\ l_{(i,j)} \in L_{(i,j)}}} \left\{ p^{i,l_i} + p^{j,l_j} + p^{(i,j),l_{(i,j)}} \right\} \tag{8.4}$$

Nun darf man nicht einfach alle berechneten Gewichte addieren, denn die Kosten der Funktionsblöcke würden sonst mehrfach eingerechnet werden (siehe Abbildung 8.2). Man darf nur eine Untermenge aller Kanten (d. h. Wandler) $E' \subseteq E$ auswählen, so daß es keinen Knoten (d. h. Funktionsblock) gibt, der Endpunkt von zwei Kanten aus E' ist.

Auf diese Weise ergibt sich eine untere Schranke für die Kosten des Systemgraphen, denn die $w_{(i,j)}$ besitzen kleinere Werte als die Kosten, die die entsprechenden Blöcke und der Wandler in der optimalen Systemgraph-Partitionierung verursachen (wg. Gleichung (8.4)).

$$\begin{aligned}
s^* &= \min_{l \in \mathbb{l}} \left\{ \sum_{k \in V} p^{k,l_k} + \sum_{e \in E} p^{e,l_e} \right\} \\
&\geq w_{(i,j)} + \min_{l \in \mathbb{l}} \left\{ \sum_{k \in V \setminus \{i,j\}} p^{k,l_k} + \sum_{e \in E \setminus \{(i,j)\}} p^{e,l_e} \right\} \\
&\geq \cdots \\
&\geq \sum_{e' \in E'} w_{e'} + \min_{l \in \mathbb{l}} \left\{ \sum_{k \in V \setminus V'} p^{k,l_k} + \sum_{e \in E \setminus E'} p^{e,l_e} \right\} \\
&\geq \sum_{e' \in E'} w_{e'} + \sum_{k \in V \setminus V'} \min_{l_k \in L_k} p^{k,l_k} + \sum_{e \in E \setminus E'} \min_{l_e \in L_e} p^{e,l_e}
\end{aligned} \tag{8.5}$$

[2]Da jeder Wandler mit genau seinen zwei benachbarten Funktionsblöcken verbunden ist, reicht es aus, die minimalen Kosten mit dem Wandler-Namen zu identifizieren.

wobei

$$V' := \left\{ i \in V \mid \exists j : \ (i,j) \in E' \vee (j,i) \in E' \right\}$$

die Menge aller von E' überdeckten Knoten (also Blöcke) ist.

Damit die auf diese Weise berechnete untere Schranke möglichst nahe am tatsächlichen Optimum s^* ist, sollten die Gesamtkosten $\sum_{e' \in E'} w_{e'}$ maximiert werden unter allen in Frage kommenden Kantenmengen $E' \subseteq E$.

Dieses Problem ist äquivalent zu *Maximal Weight Matching in allgemeinen Graphen*, für das u. a. in [27] ein Algorithmus beschrieben wird und das in $\mathcal{O}(|V|\,|E|)$ gelöst werden kann.

Nun kann es jedoch passieren, daß nicht alle Funktionsblöcke durch das Matching überdeckt werden (d. h. $V' \neq V$) oder daß es Kanten gibt, die nicht im Matching enthalten sind, aber trotzdem minimale Implementierungskosten > 0 besitzen (in dem in Abbildung 8.2 gezeigten Systemgraphen zum Beispiel wird es auf jeden Fall Wandlerblöcke geben, die nicht vom Matching überdeckt werden). Die minimalen Kosten dieser Blöcke und Wandler dürfen nach Ungleichung 8.5 auf das Ergebnis des Maximal Weight Matching-Algorithmus addiert werden, ohne daß die Eigenschaft der unteren Schranke verloren geht.

Für die Berechnung der einzelnen Teil-Kosten $w_{(i,j)}$ wird $\mathcal{O}(|L_i| \left|L_{(i,j)}\right| |L_j|)$ Zeit benötigt. Da man die Anzahl der Implementierungsmöglichkeiten bei einer linearen Kostenfunktion und Einzelzieloptimierung auf 3 für Funktionsblöcke und 9 für Wandler beschränken kann, darf man die Laufzeit hierfür als konstant ansehen (siehe Kapitel 6.2). Es müssen $|E|$ verschiedene Kosten $w_{(\cdot,\cdot)}$ berechnet werden, so daß sich die Laufzeit für das „Preprocessing“ auf $\mathcal{O}(|E|)$ ergibt.

Die nicht vom Matching E' überdeckten Blöcke und Wandler lassen sich in Zeit $\mathcal{O}(|V| + |E|)$ finden. Die Zeit zur Berechnung der minimalen Kosten dieser Blöcke und Wandler kann wiederum als konstant angesehen werden, sofern ein Preprocessing durchgeführt wurde.

Somit ergibt sich die Gesamtlaufzeit zur Bestimmung der unteren Schranke eines Systemgraphen mit Hilfe des maximalen Matchings zu

$$\mathcal{O}(|V|\,|E|) + \mathcal{O}(|E|) + \mathcal{O}(|V| + |E|) = \mathcal{O}(|V|\,|E|).$$

Untere Schranke durch lineare Programmierung

Die vermutlich genaueste, effizient berechenbare untere Schranke liefert die lineare Programmierung. Wie in Kapitel 7 bereits dargestellt, läßt sich $PART^{\mathrm{opt}}$ als ganzzahliges lineares Programm beschreiben. Da es sich hierbei um ein Minimierungsproblem handelt, erhält man durch die Linearisierung von $PART^{\mathrm{opt}}$ ein optimales, vermutlich nichtganzzahliges Ergebnis, dessen

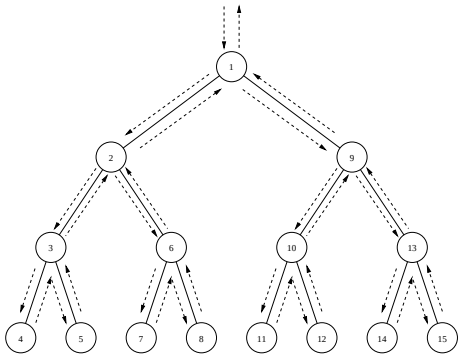


Abbildung 8.3: Exploration eines Baumes mittels Depth-First-Strategie

Kosten kleiner oder gleich den Kosten der optimalen Implementierung sind. Somit erhält man mittels linearer Programmierung eine untere Schranke für einen Systemgraphen.

Jedoch benötigen Algorithmen zur Lösung linearer Programme — auch wenn sie asymptotisch gesehen nur polynomielle Laufzeit besitzen — in der Realität recht lange zur Berechnung des Ergebnisses.

Es ist experimentell zu überprüfen, welche der drei genannten Verfahren untere Schranken liefern, die einerseits genau genug sind, um möglichst viele Teilgraphen auszuloten, andererseits aber aber auch nicht zu viel Zeit verbrauchen, um die unteren Schranken zu berechnen.

8.2.3 Finden einer oberen Schranke

Wenn man schnell eine gute obere Schranke für die minimalen Implementierungskosten findet, kann man frühzeitig Teilgraphen (also Mengen von Partitionierungsmöglichkeiten) ausloten, weil sie nicht die optimale Lösung enthalten können.

Um eine oberen Schranke zu erhalten, ist es notwendig, einen Pfad bis zu seinem Blatt zu verfolgen, also die Implementierungen aller Funktionsblöcke festzulegen. Damit dies möglichst schnell geschieht, ist es sinnvoll eine Depth-First-Strategie (siehe z. B. [5]; Abbildung 8.3) anzuwenden. Diese hat gegenüber einer Breadth-First-Strategie ([5]; Abbildung 8.4) auch den Vorteil, daß deutlich weniger Speicherplatz benötigt wird, weil die bereits bearbeiteten Konfigurationen nicht mehr gebraucht werden, und somit wieder gelöscht werden können.

Da der Baum balanciert ist, müssen in jedem Fall $|V|$ Teilprobleme erzeugt werden, um eine erste obere Schranke zu erhalten — unabhängig davon, welches der $|L.|$ Teilprobleme in jedem Rekursionsschritt gewählt wird. Jedoch läßt sich die Güte der berechneten oberen Schranke durch die Wahl des als

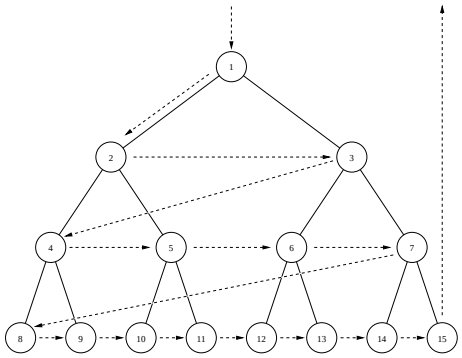


Abbildung 8.4: Exploration eines Baumes mittels Breadth-First-Strategie

nächstes zu betrachtenden Teilproblems beeinflussen. Wenn man aus den Teilproblemen $\mathfrak{S}_{...1}, \ldots, \mathfrak{S}_{...|L.|}$ dasjenige auswählt, das die minimale untere Schranke besitzt, erwartet man, daß dieses Teilproblem auch eine niedrige obere Schranke ergibt, d. h. eine nahe an die optimalen Kosten heranreichende Implementierung enthält. Daher erscheint es als geschickt, dieses Teilproblem als nächstes zu untersuchen. Diese Strategie der Traversierung des Branch & Bound-Baumes wird auch als *Best-Node-First* bezeichnet.

Es sind auch noch diverse Mischformen von Depth-First, Breadth-First und Best-Node-First denkbar (siehe [29]). Deren Qualität ist jedoch sehr empfindlich gegenüber des zu optimierenden Problems, so daß diese nur mit Hilfe repräsentativer Tests auf ihre Anwendbarkeit überprüft werden können.

Da die Implementierungskosten einer beliebigen gültigen Implementierung bereits eine obere Schranke darstellen, kann man eine erste obere Schranke auch mit Hilfe anderer Algorithmen berechnen. Es ist durchaus ein sinnvolles Verfahren, zuerst eine gute obere Schranke mit einer Heuristik, wie z. B. Simulated Annealing (Kapitel 10) oder Tabu Search (Kapitel 11) zu berechnen, und anschließend das exakte Optimum mittels Branch & Bound zu ermitteln.

8.2.4 Laufzeitbetrachtung

Im worst-case besitzt das Branch & Bound-Verfahren für $PART^{\mathrm{opt}}$ exponentielle Laufzeit. Dies ist verständlich, da $PART^{\mathrm{opt}}$ ein $\mathcal{NP}$-vollständiges Optimierungsproblem ist, und Branch & Bound eine Lösung mit minimalen Kosten liefert und nicht nur eine annähernd minimale Lösung.

Daher wird man nur relativ kleine Systemgraphen mit akzeptablem Zeitaufwand lösen können. Wenn man aber eine geeignete Strategie zur Wahl der Branch-Variablen, zur Wahl des nächsten zu betrachtenden Teilproblems und zur Berechnung der unteren Schranke benutzt, kann man die Größe der noch partitionierbaren Systemgraphen deutlich erhöhen.

Durch die Anwendung einer Heuristik zur Berechnung einer ersten oberen Schranke (siehe Abschnitt 8.2.3) für die Implementierungskosten kann man die Laufzeit eventuell nochmals verringern.

Während der Abarbeitung des Branch & Bound-Algorithmus kann man fortwährend die bisherige Approximations-Güte abschätzen. Dies ergibt sich aus der Tatsache, daß eine obere und eine untere Schranke zur Verfügung stehen, so daß sich der bisher erreichte Approximationskoeffizient durch

$$\alpha \leq \overline{s}/\underline{s}$$

abschätzen läßt.

Falls eine bestimmte Approximationsschranke für die Partitionierung ausreichen sollte, kann man also nach Erreichen eines geeigneten Approximationskoeffizienten den Algorithmus abbrechen. Die zu der oberen Schranke $\overline{s}$ gehörende Implementierung genügt dann der gewünschten Approximationsschranke.

Man kann den Algorithmus aber auch zu einem beliebigen Zeitpunkt abbrechen und erhält so eine Heuristik mit einer a-posteriori-Abschätzung von $\alpha_{post} \leq \overline{s}/\underline{s}$.

Somit läßt sich das Branch & Bound-Verfahren auch als Approximationsalgorithmus oder sogar als Heuristik verwenden.

8.2.5 Berechnung der Schranken bei nichtlinearen Kostenfunktionen

Wenn nicht bzgl. der Fläche oder anderer linearer Kostenfunktionen optimiert werden soll, kann die untere Schranke nicht mehr mit den in Abschnitt 8.2.2 vorgestellten Methoden abgeschätzt werden.

Branch & Bound stellt jedoch an die Kostenfunktion keine Anforderungen, da das Berechnen der unteren Schranken, sowie die Auswahl von Branch-Variable und nächstem zu betrachtenden Teilproblem von separaten, in sich abgeschlossenen Verfahren übernommen werden kann.

Daher können diese Teile entsprechend dem benötigten Optimierungsziel angepaßt werden. Dies ist recht problemspezifisch, so daß es hier beispielhaft nur für die Minimierung der Laufzeit des Gesamtsystems gezeigt werden soll.

Für die Berücksichtigung zusätzlicher Randbedingungen müssen diese Teil-Verfahren jedoch nicht verändert werden, da die Randbedingungen, wie in Kapitel 6 bereits gezeigt, in der Kostenfunktion berücksichtigt werden können. Dadurch ist die Kostenfunktion zwar nicht mehr linear, aber die notwendigen Eigenschaften zur Berechnung der unteren Schranke, zur Auswahl der Branch-Variablen und zur Wahl des nächsten zu betrachtenden Teilproblems bleiben erhalten.

Berechnung der Schranken zur Optimierung bzgl. der Laufzeit

Die Auswahl der Branch-Variablen sollte, ebenso wie bei linearen Kostenfunktionen, nach der take-best-Strategie erfolgen. Da die Betrachtung aller benachbarten Wandler aber keinen Sinn macht, da die maximale Laufzeit des Systems durch den kritischen Pfad bestimmt wird (der nur von dem Block und jeweils zwei benachbarten Wandlern abhängt, und nicht allen), kommen nur Maße der Form (8.2) oder (8.3) in Frage.

Die Berechnungen der unteren Schranken der einzelnen Teilprobleme kann ähnlich zu den in Abschnitt 8.2.2 entwickelten Ideen erfolgen. Die Berechnung mittels linearer Programmierung scheidet jedoch aufgrund der in Kapitel 7.4 erläuterten Probleme aus.

Man kann aber für jeden Funktionsblock und Wandler seine minimale Laufzeit anhand seiner Implementierungsmöglichkeiten ermitteln und mit diesen Werten die maximale Verzögerungszeit durch

$$\underline{s} := \max_{\lambda \subset E \text{ Pfad}} \sum_{\substack{k \in V \\ \exists j \in V : (k,j) \in \lambda \vee (j,k) \in \lambda}} \min_{l_k \in L_k} p^{k,l_k} + \sum_{(i,j) \in \lambda} \min_{l_{(i,j)} \in L_{(i,j)}} p^{(i,j),l_{(i,j)}}$$

bestimmen. Diese untere Schranke läßt sich mit dem in [18] vorgestellten Algorithmus zur Bestimmung des kritischen Pfades in $\mathcal{O}(|V| + |E|)$ berechnen.

Dieses Verfahren kann auch für die Laufzeit-Optimierung verfeinert werden, indem man, wie bereits in Kapitel 8.2.2 für lineare Kostenfunktionen ausgeführt, ein Matching E' berechnet, und anschließend mit Hilfe des Matchings neue Gewichte

$$w'_{(i,j)} := \begin{cases} w_{(i,j)} & \text{falls } (i,j) \in E' \\ \min_{l \in L_{(i,j)}} p^{(i,j),l} & \text{sonst} \end{cases}$$

$$w'_k := \begin{cases} 0 & \text{falls } k \in V' \\ \min_{l \in L_k} p^{k,l} & \text{sonst} \end{cases}$$

erzeugt. Dies garantiert, daß die Kosten von Blöcken und Wandlern nicht mehrfach gezählt werden und andererseits eine möglichst gute untere Abschätzung erreicht wird. Nun kann man wiederum den Algorithmus zur Bestimmung des kritischen Pfades, jedoch bezüglich der neuen Gewichte $w'_.$, anwenden.

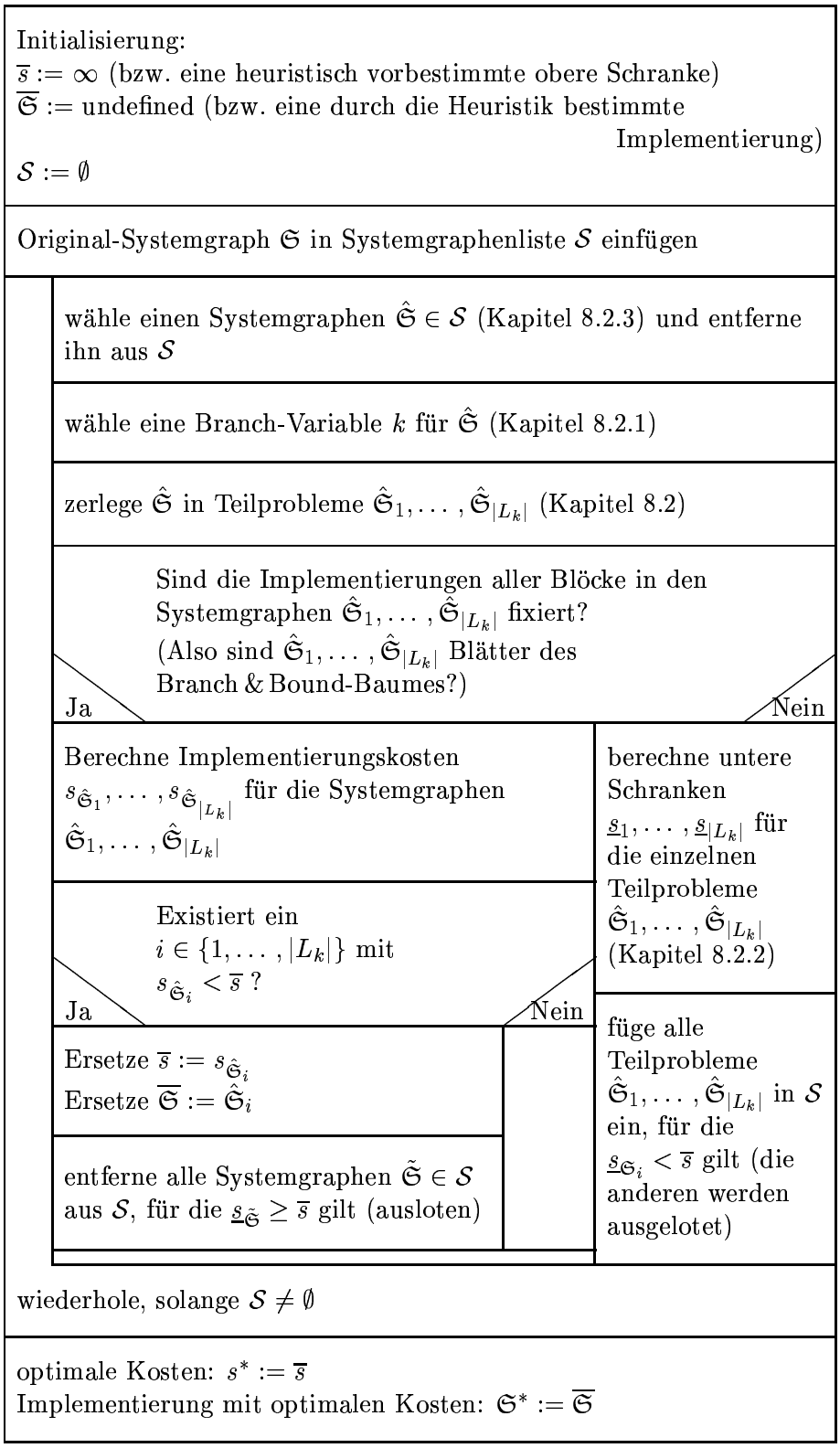
Da in dieser Variante die Kosten für die Mehrzahl der Funktionsblöcke aber auf die Wandlerkosten aufgeschlagen wurden, ist es möglich, daß die errechnete untere Schranke noch niedriger ist als die der einfachen Schätzung. Dies ergibt sich aus der Möglichkeit, daß die Kosten eines auf dem kritischen Pfad liegenden Funktionsblocks in einem Gewicht (also einer Kante) $w'_.$ berücksichtigt wird, das nicht im kritischen Pfad liegt.

Für das Finden von guten oberen Schranke gilt das in Abschnitt 8.2.3 gesagte, denn für die Effektivität der dort genannten Auswahl-Verfahren des nächsten Teilproblems ist die Monotonie-Eigenschaft der Kostenfunktion entscheidend und nicht die Linearität.

Aufgrund der $\mathcal{NP}$-Vollständigkeit von $PART^{\text{opt}}$ ist klar, daß das Finden der optimalen Implementierung eines Systemgraphen $\mathfrak{S}$ mit Branch & Bound im schlechtesten Fall exponentielle Laufzeit benötigen muß.

Wenn eine optimale Lösung gefordert wird, ist aber Branch & Bound der ganzzahligen linearen Programmierung durchaus vorzuziehen, da es viel stärker an die speziellen Eigenschaften von $PART^{\text{opt}}$ angepaßt werden kann. Außerdem ist es, im Gegensatz zur linearen Programmierung, mit Branch & Bound möglich, $PART_m^{\text{opt}}$ bzgl. einer nichtlinearen Kostenfunktion m zu optimieren, weil einzelne Teile des Verfahrens an die Kostenfunktion angepaßt werden können.

Zudem kann Branch & Bound zu beliebigen Zeitpunkten abgebrochen werden und jederzeit eine approximative Lösung liefern. Dies ist mit der ganzzahligen linearen Programmierung ebenfalls nicht möglich, da die Zwischenlösungen die Ganzzahligkeitsbedingungen verletzen und somit keine gültige Implementierung ergeben.



Algorithmus 8.1: Branch & Bound-Verfahren zur Bestimmung der optimalen Kosten und der zugehörigen System-Implementierung

Kapitel 9

Dynamische Programmierung

Die Dynamische Programmierung ist eine Erweiterung des Divide & Conquer-Verfahrens, bei dem das gegebene Problem in mehrere Teilprobleme zerlegt wird, die separat gelöst und schließlich wieder zu einer vollständigen Lösung zusammengesetzt werden. Im Gegensatz zu Branch & Bound wird aber nicht der Lösungsraum zerlegt, sondern das Problem selbst.

Als Erweiterung von Divide & Conquer werden bei der Dynamischen Programmierung die Ergebnisse der berechneten Teilprobleme gespeichert, um eventuelle Mehrfachberechnungen zu vermeiden. Dadurch ergeben die Berechnungsabhängigkeiten keinen Baum mehr, sondern einen gerichteten azyklischen Graphen (siehe Abbildung 9.1).

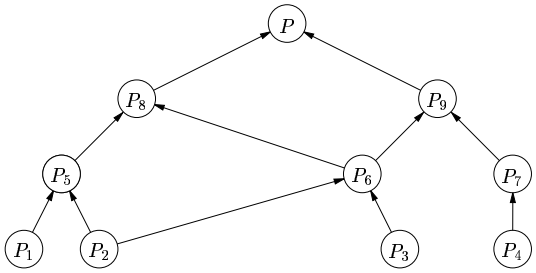


Abbildung 9.1: Durch die Berechnungsabhängigkeiten beim Dynamischen Programmieren aufgespannter gerichteter azyklischer Graph

Üblicherweise bearbeitet man aber nicht, wie bei Divide & Conquer, rekursiv die einzelnen Teilprobleme im Top-Down-Verfahren vom großen zum kleinen Problem, sondern geht (Bottom-Up) von atomischen, direkt lösbaren Teilproblemen aus und erweitert diese iterativ, bis man die Lösung des eigentlich zu lösenden Problems erhalten hat (Abbildung 9.2). Dabei werden in jedem

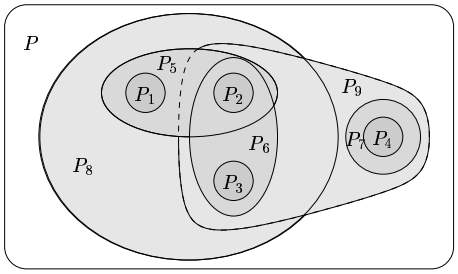


Abbildung 9.2: Sukzessives Lösen der Teilprobleme 1, 2, ... bis das gegebene Problem P gelöst ist (gleiche Abhängigkeiten wie in Abbildung 9.1).

Iterationsschritt die Ergebnisse gespeichert, so daß auf bereits berechnete Teilprobleme in Konstantzeit zugegriffen werden kann. Die Bottom-Up-Strategie hat den Vorteil, daß Daten, die im weiteren Verlauf des Algorithmus nicht mehr benötigt werden, wieder gelöscht werden können[1]. Z. B. wird nach dem Lösen des Teilproblems 7 aus Abbildung 9.1 die Lösung von Teilproblem 4 nicht mehr benötigt, kann also gelöscht werden.

9.1 Anwendung auf $PART^{\mathbf{opt}}$

Die geeignete Zerlegung des Systemgraphen in Teilprobleme soll zuerst an einem Beispiel motiviert werden:

Beispiel 9.1. Sei der in Abbildung 9.3 dargestellte Systemgraph $\mathfrak{S}$ bzgl. seines Flächenbedarfs zu optimieren. Der Einfachheit wegen sind unter den Blöcken und Wandlern jeweils (informal) ihre Implementierungsmöglichkeiten mit den Zeitmodelldomänen und dem Flächenbedarf angegeben.

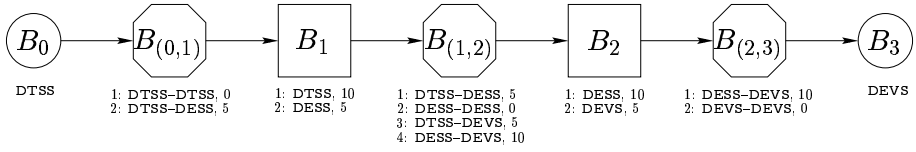


Abbildung 9.3: Systemgraph $\mathfrak{S}$

Angenommen der Systemgraph bestehe nur aus dem Eingangsblock B_0 und dem Wandler $B_{(0,1)}$ (Abbildung 9.4). Dieser Systemgraph $\mathfrak{S}_0$ besitzt zwei verschiedene implementierbare Zeitmodelldomänen für den Ausgang, nämlich DTSS und DESS. Da die Ausgabe-Blöcke eigentlich nur eine Zeitmodelldomäne besitzen dürfen, wird der Systemgraph zwischen diesen unter-

[1] Meistens werden die Daten aber noch benötigt, weil man in einem zweiten Durchlauf aus den berechneten optimalen Kosten die dazugehörige Lösung bestimmt.

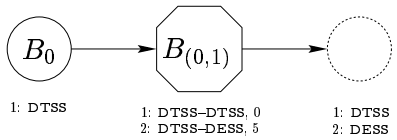


Abbildung 9.4: Systemgraph $\mathfrak{S}_0$

schieden: Der Systemgraph mit Ausgang in DTSS wird mit $\mathfrak{S}_0(\mathtt{DTSS})$ bezeichnet, der Systemgraph mit Ausgang in DESS mit $\mathfrak{S}_0(\mathtt{DESS})$.

Da die beiden Systemgraphen $\mathfrak{S}_0(\mathtt{DTSS})$ und $\mathfrak{S}_0(\mathtt{DESS})$ aus nur einem Wandler bestehen, läßt sich der jeweils minimale Platzbedarf direkt ablesen:

$$s_0^*(\mathtt{DTSS}) = p^{(0,1),1} = 0$$
$$s_0^*(\mathtt{DESS}) = p^{(0,1),2} = 5$$

Jetzt kann man diesen Systemgraphen $\mathfrak{S}_0$ erweitern um den Block B_1 und den Wandler $B_{(1,2)}$ und erhält so den Systemgraphen $\mathfrak{S}_1$ (d. h. $\mathfrak{S}_1(\mathtt{DESS})$ und $\mathfrak{S}_1(\mathtt{DEVS})$; Abbildung 9.5). Deren minimaler Flächenbedarf berechnet

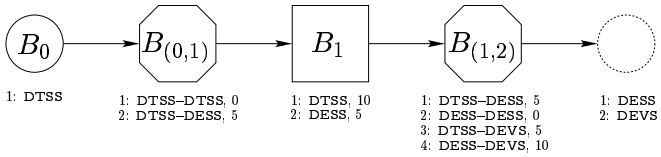


Abbildung 9.5: Systemgraph $\mathfrak{S}_1$

sich als Minimum aller gültigen Kombinationen der optimalen Ergebnisse von $\mathfrak{S}_0$ und der Kosten der neu hinzugekommenen B_1 und $B_{(1,2)}$.

$$\begin{aligned} & s_1^*(\mathtt{DESS}) \\ & \quad = \min\left\{ s_0^*(\mathtt{DTSS}) + p^{1,1} + p^{(1,2),1},\ s_0^*(\mathtt{DESS}) + p^{1,2} + p^{(1,2),2} \right\} \\ & \quad = \min\{0 + 10 + 5,\ 5 + 5 + 0\} \\ & \quad = 10 \end{aligned}$$

$$\begin{aligned} & s_1^*(\mathtt{DEVS}) \\ & \quad = \min\left\{ s_0^*(\mathtt{DTSS}) + p^{1,1} + p^{(1,2),3},\ s_0^*(\mathtt{DESS}) + p^{1,2} + p^{(1,2),4} \right\} \\ & \quad = \min\{0 + 10 + 5,\ 5 + 5 + 10\} \\ & \quad = 15 \end{aligned}$$

Durch die Erweiterung von $\mathfrak{S}_1$ um Block B_2 und Wandler $B_{(2,3)}$ erreicht man den vollständigen Systemgraphen $\mathfrak{S}$. Auch hier ergibt sich die minimale

Fläche wieder durch das Minimum aller gültigen Kombinationen der Optima von $\mathfrak{S}_1$ und den kompatiblen Implementierungsmöglichkeiten von B_2 und $B_{(2,3)}$. Da die Zeitmodelldomäne des Ausgangs auf DEVS festgelegt ist, berechnet sich das Minimum als

$$\begin{aligned} s^* :=& s_2^*(\mathtt{DEVS}) \\ =& \min\left\{ s_1^*(\mathtt{DESS}) + p^{2,1} + p^{(2,3),1},\ s_1^*(\mathtt{DEVS}) + p^{2,2} + p^{(2,3),2} \right\} \\ =& \min\{10 + 10 + 10,\ 15 + 5 + 0\} \\ =& 20 \end{aligned}$$

Die minimal erreichbare Fläche der durch den Systemgraphen spezifizierten Schaltung beträgt also 20. Die optimale Implementierung der einzelnen Funktionsblöcke und Wandler erhält man, indem man für die einzelnen Teilgraphen neben den optimalen Kosten auch die dazugehörigen Implementierungen speichert. Für das Beispiel ergibt sich die optimale Implementierung

$$\mathfrak{l}^* := (l^*_{(0,1)}, l^*_1, l^*_{(1,2)}, l^*_2, l^*_{(2,3)}) = (1, 1, 3, 2, 2).$$

Die in dem Beispiel angewandte Optimierungs-Strategie läßt sich ebenso auf allgemeine (zykelfreie) Systemgraphen erweitern. Zuerst werden für alle möglichen Kombinationen die Kosten der Wandler berechnet, die direkt an den Eingangsblöcken hängen. Dann werden die an diesen Wandlern hängenden Funktionsblöcke mit ihren nachfolgenden Wandlern hinzugenommen und die optimalen Kosten für alle Ausgangszeitmodell-Kombinationen berechnet. Anschließend werden die von diesen Blöcken abhängigen Funktionsblöcke inklusive nachfolgender Wandler in die Optimierung mit einbezogen, u. s. w.

Der Systemgraph läßt sich mit Hilfe der Signalabhängigkeiten in verschiedene Ebenen zerlegen, wobei alle in einer Ebene enthaltenen Funktionsblöcke voneinander unabhängig sind. Die nachfolgenden Wandler der Funktionsblöcke werden zu der gleichen Ebene gezählt.

9.1.1 Zerlegung des Systemgraphen in einzelne Ebenen

Die einzelnen Ebenen lassen sich mit Hilfe eines leicht veränderten Algorithmus zur topologischen Sortierung (siehe z. B. [27]) berechnen. Auf diese Weise erhält man Mengen V_i $(i = 0, \ldots, n)$ von Funktionsblöcken, deren Eingabe-Signale von den Ausgaben der Funktionsblöcke der Menge V_{i-1} abhängen. Es wird also eine Teilordnung über alle Blöcke $v \in V$ gebildet, die die Signalabhängigkeiten der Funktionsblöcke charakterisiert.

Anhand der in V_i enthaltenen Funktionsblöcke lassen sich die Wandler ermitteln, die an den Ausgängen der Funktionsblöcke liegen, also zu der Ebene i gehören ($E_i := \{(v, w) \in E |\ v \in V_i\}$).

Initialisierung:
V: Block-Menge des Systemgraphen
E: Wandler-Menge des Systemgraphen
$i := 0$

solange $V \neq \emptyset$ (d. h. solange noch nicht alle Blöcke in eine Ebene eingeordnet sind)

- $V_i := \emptyset$ (neue Ebene initialisieren)
- $\forall v \in V :\ \text{indeg}(v) = 0$
  (alle Blöcke, deren Eingangssignale nur von Blöcken aus $V_1, \ldots, V_{i-1}$ abhängig sind)
  - $V_i := V_i \cup \{v\}$ (Block v der Ebene i hinzufügen)
  - $V := V \setminus \{v\}$ (Block v aus der Menge der noch nicht einsortierten Blöcke entfernen)
  - $E := E \setminus \{(v, v') \in E \mid v' \in V\}$ (Alle Wandler, die von Block v ausgehen, entfernen)
- $i := i + 1$ (Ebene i abschließen)

$V_1, \ldots, V_n$ entsprechen den einzelnen Ebenen

Algorithmus 9.1: Einteilung des Systemgraphen in die einzelnen Abhängigkeitsebenen

Der modifizierte Algorithmus 9.1 benötigt $\mathcal{O}(|V| + |E|)$ Laufzeit, falls die Kanten (d. h. Wandler) mit einer Adjazenzliste repräsentiert werden [27].

In Abbildung 9.6 wurde der in Kapitel 2.6 angegebene Beispiel-Systemgraph (Abbildung 2.5) in seine einzelnen Ebenen unterteilt.

Aus dem Verfahren zur Aufteilung des Systemgraphen in die einzelnen Ebenen kann man direkt eine wichtige Anforderung an den Systemgraphen erkennen. Und zwar terminiert der Algorithmus nur, wenn der Systemgraph azyklisch ist. Diese Einschränkung ist aber eliminierbar, wie in Abschnitt 9.1.3 gezeigt wird.

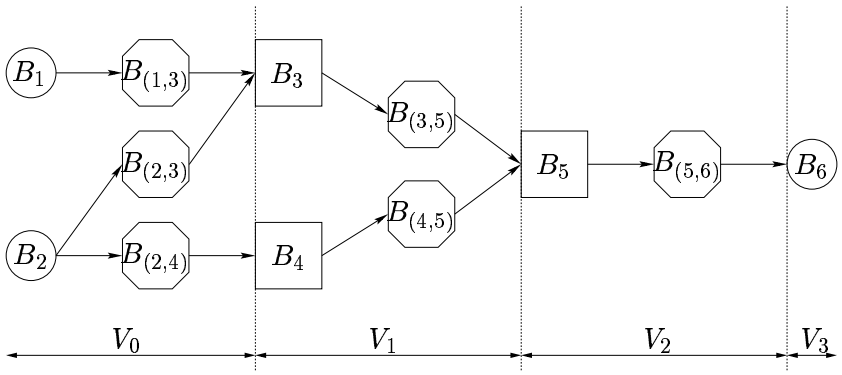


Abbildung 9.6: Einteilung eines Systemgraphen in Ebenen

9.1.2 Berechnung der optimalen Implementierung

Nachdem der Systemgraph in Ebenen unterteilt ist, kann man bei der ersten Ebene beginnend, diesen Teilgraphen für jede mögliche Kombination der Ausgangs-Zeitmodelldomänen der Wandler dieser Ebene optimieren und den Teilgraphen sukzessive um eine Ebene vergrößern.

Für jeden Teil-Systemgraphen $\mathfrak{S}_i$, der aus den Blöcken der Ebenen $0, \ldots, i$ besteht, ergeben sich bei linearer Kostenfunktion die minimalen Gesamtkosten $s_i^*(\mathfrak{d}_i)$ durch

$$s_i^*(\mathfrak{d}_i) = \min_{\mathfrak{d}_{i-1} \in D^{|\mathfrak{d}_{i-1}|}} \left\{ s_{i-1}^*(\mathfrak{d}_{i-1}) + \sum_{k \in V_i} \min_{\substack{l_k \in L_k \\ d_{in}^{k,l_k} = d_{i-1}^k}} \left\{ p^{k,l_k} + \sum_{(k,j) \in E} \min_{\substack{l_{(k,j)} \in L_{(k,j)} \\ d_{in}^{(k,j),l_{(k,j)}} = d_{out}^{k,l_k} \\ d_{out}^{(k,j),l_{(k,j)}} = d_i^j}} p^{(k,j),l_{(k,j)}} \right\} \right\}. \quad (9.1)$$

Dabei ist $\mathfrak{d}_i := (d_i^k)_{k \in V_i}$ der Vektor der Zeitmodelldomänen aller Ausgänge des von $V_1, \ldots, V_i$ und deren Wandlern erzeugten Systemgraphen $\mathfrak{S}_i$. Man beachte, daß $\min\{\} := \infty$, so daß nicht implementierbare Ausgangs-Kombinationen automatisch mit unendlichen Kosten belegt werden. Ein Algorithmus kann diese Belegungen direkt abfangen und braucht deren Kosten $s_i^*(\mathfrak{d}_i)$ erst gar nicht zu berechnen. Außerdem ist es nicht notwendig, die optimalen Kosten für eine Kombination $\mathfrak{d}_i$ zu berechnen, wenn erkennbar ist, daß die Zeitmodelldomänen-Kombination zu den Implementierungsmöglichkeiten der Blöcke $k \in V_{i+1}$ der nächsten Ebene $i+1$ nicht kompatibel ist.

Zur Bestimmung des innersten Minimums von Gleichung (9.1) sind höchstens $\max_{(k,j) \in E} |L_{(k,j)}|$ Vergleiche notwendig. Da die i-te Ebene $|\mathfrak{d}_{i-1}|$ Eingabe- und $|\mathfrak{d}_i|$ Ausgabe-Signale besitzt, kann es pro Ebene höchstens

$|\mathfrak{d}_{i-1}|\,|\mathfrak{d}_i|$ Wandler geben und genau $|\mathfrak{d}_{i-1}|$ Funktionsblöcke. Da $|D|^{|\mathfrak{d}_{i-1}|}$ verschiedene Kombinationen der Zeitmodelldomänen für die Eingabe-Leitungen der Ebene i existieren, ergibt sich ein Gesamt-Rechenaufwand für $s_i^*(\mathfrak{d}_i)$ von

$$\mathcal{O}\left(|D|^{|\mathfrak{d}_{i-1}|}\left(|\mathfrak{d}_{i-1}|\,|\mathfrak{d}_i|\max_{(k,j)\in E}|L_{(k,j)}| + |\mathfrak{d}_{i-1}|\max_{k\in V_i}|L_k|\right)\right).$$

Die Anzahl der Implementierungsmöglichkeiten läßt sich für lineare Kostenfunktionen und Einzelzieloptimierung auf 9 pro Wandler und 3 pro Funktionsblock mittels Preprocessing reduzieren (siehe Kapitel 6.2), so daß diese als konstant angesehen werden können. Außerdem ist $|D| = 3$. Damit ergibt sich der Berechnungsaufwand für $s_i^*(\mathfrak{d}_i)$ zu

$$\mathcal{O}\left(3^{|\mathfrak{d}_{i-1}|}\,|\mathfrak{d}_{i-1}|\,|\mathfrak{d}_i|\right).$$

Wenn die optimalen Kosten $s_{i-1}^*(\mathfrak{d}_{i-1})$ für alle Kombinationen $\mathfrak{d}_{i-1} \in D^{|\mathfrak{d}_{i-1}|}$ der Ebene $i-1$ bereits bekannt sind, können die optimalen Kosten $s_i^*(\mathfrak{d}_i)$ für *sämtliche* Ausgangs-Zeitmodelldomänen-Kombinationen $\mathfrak{d}_i \in D^{|\mathfrak{d}_i|}$ der Ebene i in Zeit

$$\mathcal{O}\left(3^{|\mathfrak{d}_i|}3^{|\mathfrak{d}_{i-1}|}\,|\mathfrak{d}_{i-1}|\,|\mathfrak{d}_i|\right) = \mathcal{O}\left(3^{|\mathfrak{d}_i|+|\mathfrak{d}_{i-1}|}\,|\mathfrak{d}_{i-1}|\,|\mathfrak{d}_i|\right)$$

berechnet werden.

Dies erscheint auf den ersten Blick nicht praktikabel, da exponentielle Zeit zur Berechnung benötigt wird. Wenn jedoch die Ebenen relativ klein sind, der Systemgraph also nur wenig „Parallelität“ aufweist, sind die optimalen Kosten der einzelnen Ebenen durchaus in akzeptabler Zeit berechenbar.

Die Berechnung der $s_i^*(\cdot)$ durch Gleichung 9.1 ist aber nur möglich, wenn bzgl. einer linearen Kostenfunktion optimiert wird. Dies ergibt sich aus der Tatsache, daß nur lineare Funktionen die hier ausgenutzte Eigenschaft $\min_x f(x) + \min_y f(y) = \min_{x,y} f(x+y)$ garantieren.

Für nichtlineare Kostenfunktionen, wie z. B. für die Optimierung bzgl. der maximalen Verzögerungszeit, lassen sich wahrscheinlich keine Adaptionen an die Dynamische Programmierung finden, deren Komplexität unter der der vollständigen Enumeration liegt.

Das sich aus Gleichung (9.1) ergebende Dynamische Programm zur Optimierung von $PART^{\text{opt}}$ bzgl. einer linearen Kostenfunktion ist in Algorithmus 9.2 angegeben. Da in der letzten Ebene n keine Wandler mehr enthalten sind, sind die optimalen Kosten nicht abhängig von den Zeitmodelldomänen der Ausgabe-Leitungen. Somit gibt es nur noch einen minimalen Kostenwert $s_n^*()$.

Die zu $s_n^*()$ gehörende optimale Implementierung erhält man, indem für jeden optimalen Wert $s_i^*(\cdot)$ die hierfür notwendige Implementierung des gesamten Systemgraphen $\mathfrak{S}_i$ gespeichert wird. Dann können alle Ergebnisse der früheren Ebenen $\leq i-2$ gelöscht werden.

Eingabe: Systemgraph $\mathfrak{S}$

Berechne Ebenen $V_0, \ldots, V_n$ nach Algorithmus 9.1

for $i := 0$ to n
(sukzessive den Graphen um eine Ebene vergrößern)

$\forall \mathfrak{d}_i \in D^{|\mathfrak{d}_i|}$
(für alle Kombinationen der Zeitmodelldomänen der Ausgabe-Signale der aktuellen Ebene i)

Berechne $s_i^*(\mathfrak{d}_i)$ (nach Gleichung (9.1))

Speichere den Wert $s_i^*(\mathfrak{d}_i)$ und die dafür verwendeten Implementierungen l_v^*, $l_{(v,w)}^*$ $(v \in V_i, (v,w) \in E_i)$

Ausgabe: $s_n^*()$

Algorithmus 9.2: Verfahren zum Lösen von $PART^{\text{opt}}$ mittels Dynamischer Programmierung

Alternativ wäre es möglich, für jeden optimalen Wert $s_i^*(\cdot)$ nur die dafür notwendige Implementierung der Blöcke und Wandler der Ebene i zu speichern, sowie einen Zeiger auf die hierfür gewählte Implementierung der $(i-1)$-ten Ebene.

Die Berechnung der optimalen Kosten der i-ten Ebene $s_i^*(\cdot)$ basiert auf allen Ergebnissen der $(i-1)$-ten Ebene. Dadurch ergibt sich ein Abhängigkeitsgraph wie in Abbildung 9.7 dargestellt. Beginnend von nur einer Kombination (aufgrund der festgelegten, eindeutigen Zeitmodelldomänen der Eingabe-Blöcke) wird der Systemgraph sukzessive erweitert, und in jeder Ebene werden für alle gültigen Kombinationen der Zeitmodelldomänen der Ausgangs-Signale die minimalen Kosten $s_i^*(\cdot)$ berechnet. In der Ebene n sind nur noch die Ausgangs-Blöcke enthalten, für die wiederum genau eine Zeitmodelldomänen-Kombination festgelegt ist $(s_n^*())$.

Aus dem Algorithmus 9.2 und der Laufzeitabschätzung zur Berechnung von Gleichung (9.1) ergibt sich eine Laufzeit für den Algorithmus von

$$\mathcal{O}\left(\sum_{i=0}^{n} 3^{|\mathfrak{d}_i|+|\mathfrak{d}_{i-1}|}\,|\mathfrak{d}_{i-1}|\,|\mathfrak{d}_i|\right) = \mathcal{O}\left(n 9^{|\mathfrak{d}_p|}\,|\mathfrak{d}_p|\right),$$

wobei n die Anzahl der Ebenen (d. h. die Anzahl der Funktionsblöcke, die

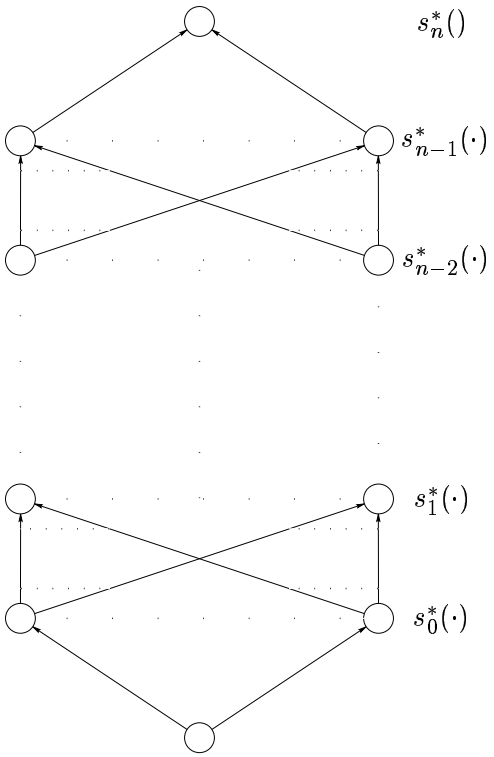


Abbildung 9.7: Abhängigkeitsgraph für $PART^{\text{opt}}$

ein Signal maximal durchlaufen kann) bezeichne und p die Ebene mit der maximalen „Breite“ ($|\mathfrak{d}_p| = \max_{i=0}^{n} |\mathfrak{d}_i|$).

Wenn die „Breite“ des Systemgraphen höchstens logarithmisch zur „Länge“ ist, d. h.

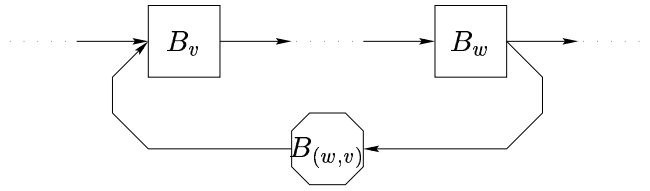
$$|\mathfrak{d}_p| = \mathcal{O}(\log n),$$

benötigt Algorithmus 9.2 nur polynomielle Laufzeit zur exakten Optimierung von $PART^{\text{opt}}$ bzgl. einer linearen Kostenfunktion!

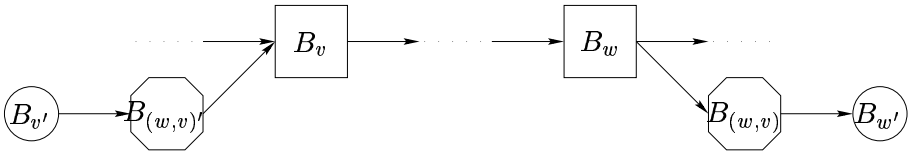
9.1.3 Auflösen von Zykeln im Systemgraphen

Wie bereits angedeutet, funktioniert der Algorithmus 9.2 nur mit zykelfreien Systemgraphen, weil die Einteilung in die einzelnen Ebenen sonst nicht möglich wäre, und außerdem der Abhängigkeitsgraph des Dynamischen Programms nicht azyklisch wäre.

Wenn man den Systemgraphen mittels Breadth-First traversiert, werden diejenigen Kanten als Rückwärts-Kanten markiert, ohne die der Graph zykelfrei wäre (siehe auch [5]).



(a) Zykel



(b) aufgetrennter Zykel

Abbildung 9.8: Zykel in Systemgraphen

Jetzt darf man natürlich eine Rückwärts-Kante (o. B. d. A. sei dies der Wandler $(w, v) \in E$) nicht einfach aus dem Systemgraphen entfernen, da dieser Wandler die Zeitmodelldomänen der für die Blöcke v und w gewählten Implementierungen entsprechend konvertieren muß, und die Kosten für den Wandler entsprechend berücksichtigt werden müssen (Abbildung 9.8(a)).

Man kann den Systemgraphen an dieser Stelle aber auftrennen, einen neuen Wandler $B_{(w,v)'}$ einfügen und die beiden „losen Enden“ mit einem Eingangs-, bzw. Ausgangs-Block versehen (siehe Abbildung 9.8(b)).

Der neu hinzugefügte Wandler besitzt keinerlei Konvertierungs-Funktionalität, kann aber alle Signal-Typen kostenneutral transportieren. Somit besitzt er nur die Implementierungsmöglichkeiten

$$B_{(v,w)',\mathrm{DTSS}} := ((\mathtt{NULL}, \mathtt{CONV}, \mathtt{DTSS}, \mathtt{DTSS}, \mathtt{NULL})$$
$$B_{(v,w)',\mathrm{DESS}} := ((\mathtt{NULL}, \mathtt{CONV}, \mathtt{DESS}, \mathtt{DESS}, \mathtt{NULL})$$
$$B_{(v,w)',\mathrm{DEVS}} := ((\mathtt{NULL}, \mathtt{CONV}, \mathtt{DEVS}, \mathtt{DEVS}, \mathtt{NULL}).$$

Wenn man nun den beiden Ein- und Ausgangs-Blöcken die gleiche Zeitmodelldomäne zuweist, kann man die beiden nach der Optimierung wieder miteinander verbinden und erhält den ursprünglichen Systemgraphen.

Somit konstruiert man aus diesem, azyklischen Systemgraphen 3 verschiedene Systemgraphen, bei denen die Ein- und Ausgangs-Blöcke v' und w' jeweils die gleichen Zeitmodelldomänen DTSS, DESS, bzw. DEVS besitzen. Diese 3 Systemgraphen werden unabhängig voneinander optimiert. Die System-Implementierung mit den niedrigsten Kosten ist die optimale für das System mit dem Zykel (siehe Algorithmus 9.3).

Eingabe: Systemgraph $\mathfrak{S}$
Berechne zykelfreien Graphen $\mathfrak{S}'$ (neue Eingabe-Blöcke seien $v_1, \ldots, v_m$, und die zugehörigen neuen Ausgabe-Blöcke $w_1, \ldots, w_m$)
$\forall (l_1, \ldots, l_m) \in \{\mathtt{DTSS}, \mathtt{DESS}, \mathtt{DEVS}\}^m$ (für alle Kombinationen der Zeitmodelldomänen der neuen Ein-/Ausgabeblöcke)
Berechne die optimalen Kosten $s^*_{(l_1,\ldots,l_m)}$ für den Systemgraphen $\mathfrak{S}'$ mit Algorithmus 9.2 (wobei $l_{v_1} := l_{w_1} := l_1, \ldots, l_{v_m} := l_{w_m} := l_m$)
Ausgabe: $\min_{(l_1,\ldots,l_m)\in\{\mathtt{DTSS},\mathtt{DESS},\mathtt{DEVS}\}^m} s^*_{(l_1,\ldots,l_m)}$

Algorithmus 9.3: Optimierung von zyklischen Systemgraphen mittels Dynamischer Programmierung

Wenn mehrere Zykel in dem Systemgraphen existieren, müssen alle Zeitmodelldomänen-Kombinationen der neuen Ein- bzw. Ausgänge optimiert werden, also 3^m-mal der Algorithmus 9.2 ausgeführt werden (wenn m die Anzahl der Zyklen ist).

9.1.4 Parallelisierbarkeit

Wie bereits erwähnt, wird die Berechnung der optimalen Kosten für die i-te Ebene ausschließlich auf die Kosten der Ebene $i-1$ zurückgeführt. Andererseits werden in jeder Ebene $3^{|\mathfrak{d}_i|}$ verschiedene optimale Kosten berechnet, die nicht voneinander abhängig sind.

Daher ist es durchaus interessant, die Berechnungen der einzelnen $s^*_i(\cdot)$ auf mehrere Prozessoren zu verteilen. Da aber jede Berechnung sämtliche Daten der vorherigen Ebene benutzt, ist vermutlich ein paralleles System mit gemeinsamem Speicher notwendig, um extreme Netzbelastungen zu vermeiden.

Zusammenfassend bleibt festzuhalten, daß die dynamische Programmierung sehr gut geeignet ist für Systeme, die wenig Parallelität aufweisen und bezüglich einer linearen Kostenfunktion optimiert werden sollen. Für Systeme mit höchstens logarithmischer „Breite“ und logarithmischer Anzahl an Signalrückführungen liefert dieses Optimierungsverfahren eine optimale Implementierung in polynomieller Zeit.

Allerdings ist die dynamische Programmierung für nichtlineare Kostenfunktionen ungeeignet, und auch zusätzliche Randbedingungen (wie z. B. maximal zur Verfügung stehende Chipfläche) können nicht berücksichtigt werden.

Daher kann die dynamische Programmierung zwar bei vielen Optimierungsproblemen der Systemgraphen nicht angewendet werden, liefert bei speziellen Problemen aber optimale Implementierungen, ohne exponentiellen Zeitaufwand zu benötigen.

Kapitel 10

Simulated Annealing

Simulated Annealing zählt zu den lokalen Optimierungsverfahren, weil man von einer Lösung ausgehend die Nachbarschaft erkundet und dort nach einer besseren Lösung sucht. Es ist aber nicht garantiert, daß Simulated Annealing eine optimale Lösung liefert. Vielmehr wird man das Verfahren nach einer gewissen Anzahl von Schritten abbrechen und hoffen, daß während der Laufzeit eine nahezu optimale Lösung gefunden wurde. Da dies aber nicht garantiert werden kann, ist Simulated Annealing eine Heuristik.

10.1 Grundlegendes Verfahren

Der Name des Verfahrens ergibt sich aus einem statistischen Modell für thermodynamische Prozesse der Kristallzüchtung, in dem auch die Ursprünge des Simulated Annealing liegen. Und zwar entspricht ein perfektes homogenes Kristallgitter einer Struktur mit minimaler Energie. Diese Gitterstruktur erreicht man, indem das Material erhitzt wird bis es einen amorphen, flüssigen Zustand erreicht. Wenn man nun das Material sehr langsam abkühlt, bleibt bei jeder Temperatur das thermale Gleichgewicht erhalten, und die Atome richten sich derart aus, daß das Kristallgitter eine Struktur mit nahezu minimaler Energie annimmt (siehe [23]).

Dieses Verfahren kann nun auf andere Optimierungsprobleme übertragen werden, indem man sich vorstellt, daß die Kristallstruktur eine Konfiguration des Problems darstellt, und der Wechsel in andere Konfigurationen, genau wie die Änderung der Kristallstruktur, mit abnehmender Temperatur immer schwieriger wird. Der anschauliche Begriff der Temperatur wird beim abstrakten Optimierungsverfahren beibehalten.

Um Simulated Annealing anwenden zu können, wird eine Anfangslösung benötigt (die aber je nach Problemstellung auch per Zufallsgenerator erzeugt werden kann) und ein Nachbarschaftsverhältnis $U(\cdot)$. Dieses Nach-

barschaftsverhältnis bestimmt, zu welchen Konfigurationen man von einer beliebigen anderen Konfiguration wechseln kann.

Definition 10.1. Sei $\mathcal{L}$ die Lösungsmenge eines Optimierungsproblems. Dann ist

$$\begin{aligned} U :& \mathcal{L} \to \mathrm{pot}(\mathcal{L}) \\ & \mathcal{L} \ni x \mapsto U(x) \subseteq \mathcal{L} \end{aligned}$$

eine *Nachbarschaftsfunktion*, falls

$$y \in U(x) \iff x \in U(y).$$

Man sagt, zwei Lösungen x und y sind *benachbart*, falls $y \in U(x)$.

Außerdem wird eine Kostenfunktion $f(x)$ benötigt, die die Kosten für die Lösung x angibt.

Das Verfahren startet bei der Anfangslösung x und wählt einen zufälligen Nachbarn $x' \in U(x)$ aus. Besitzt dieser Nachbar geringere Kosten als die vorherige Lösung (also $f(x') \leq f(x)$), dann wird x' als neue, bessere Lösung akzeptiert. Falls der Nachbar x' höhere Kosten als x verursacht (also $f(x') > f(x)$), kann er immer noch als neue Lösung akzeptiert werden, jedoch nur mit einer von der Kostendifferenz $f(x') - f(x)$ und der aktuellen Temperatur T abhängigen Wahrscheinlichkeit ($\mathrm{Ws}(x'$ wird als Lösung akzeptiert$) = \exp\left(\frac{f(x)-f(x')}{T}\right)$). Dahinter steckt die Idee, daß man zwar die lokalen Minima des Lösungsraums erreichen will, wenn ein Minimum aber „schlecht" ist, dann soll man aus diesem auch wieder „herausklettern" können.

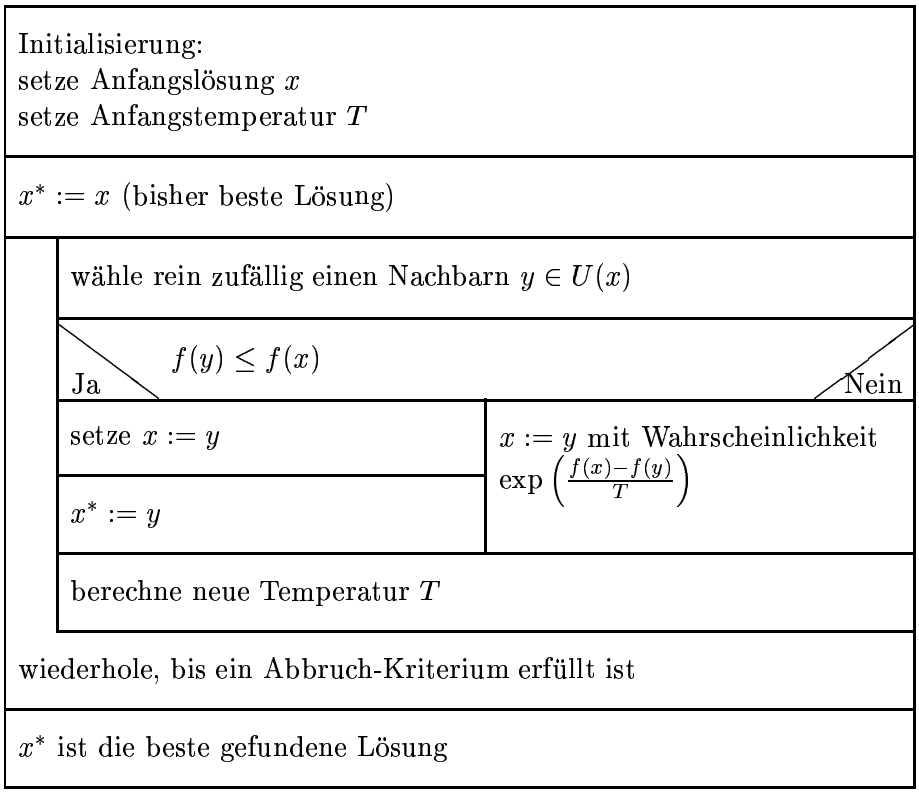
Während der Laufzeit des Algorithmus wird die Temperatur kontinuierlich gesenkt, so daß es immer unwahrscheinlicher wird, daß eine schlechtere Lösung akzeptiert wird. Man kann also nicht mehr so leicht aus der Umgebung des Minimums entkommen, sondern wird vielmehr zu dem Minimum „hingezogen".

Die Struktur von Simulated Annealing wird durch Algorithmus 10.1 nochmals verdeutlicht.

Für die Adaptierung des Verfahrens auf die gegebene Problemstellung ergeben sich hauptsächlich zwei Fragen:

- Wie ist das Nachbarschaftsverhältnis der Lösungen zu wählen?
- Welche Anfangstemperatur T soll gewählt werden? Wie schnell soll T abgekühlt werden, und nach welchem Verfahren? Wann kann der Algorithmus beendet werden? (*cooling scheme*)

Die Nachbarschaftsfunktion sollte so gewählt sein, daß einerseits nicht zu viele Lösungen zueinander benachbart sind, denn sonst springt man zu leicht



Algorithmus 10.1: Algorithmus von Simulated Annealing

aus der Umgebung eines (lokalen) Minimums heraus und kann auf diese Weise das Minimum gar nicht (bzw. nur „per Zufall“) erreichen (Abbildung 10.1). Andererseits müssen die Nachbarschaftsverhältnisse so gestaltet sein, daß man von einer beliebigen Lösung zu allen anderen Lösungen in endlich vielen Schritten gelangen kann. Denn sonst würde der Lösungsraum in zwei Zusammenhangskomponenten zerfallen, und man könnte die optimale Lösung eventuell überhaupt nicht erreichen, weil sie in einer anderen Zusammenhangskomponente liegt (Abbildung 10.2).

Es läßt sich zeigen, daß Simulated Annealing die optimale Lösung finden wird (falls der Lösungsraum aufgrund der Nachbarschaftsfunktion zusammenhängend ist), wenn man die Temperatur nur langsam genug senkt (siehe hierfür z. B. [1]). Dies ist aber nur von theoretischem Interesse, da dafür unter Umständen unendlich viele Schritte notwendig sind.

10.2 Anwendung auf $PART^{opt}$

Beim Branch & Bound-Verfahren wurde der Systemgraph $\mathfrak{S}$ in verschiedene Teilprobleme zerlegt, indem die Implementierung eines ausgewählten Blocks

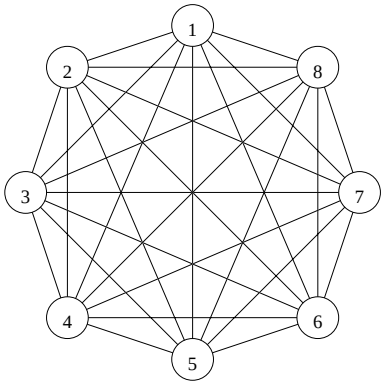


Abbildung 10.1: Schlechtes Nachbarschaftsverhältnis, weil alle Lösungen zueinander benachbart sind.

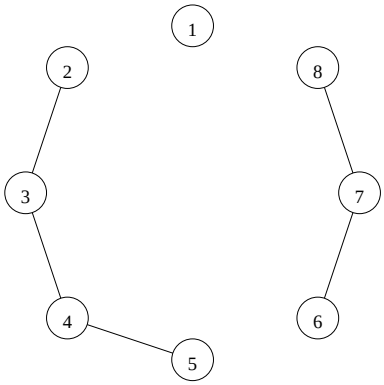


Abbildung 10.2: Schlechtes Nachbarschaftsverhältnis, weil der Lösungsraum in drei Zusammenhangskomponenten zerfällt und nicht alle Lösungen erreicht werden können.

fixiert wurde.

Eine ähnliche Idee kann man für die Nachbarschaftsbeziehungen verfolgen. Es sollen diejenigen Implementierungen $\mathfrak{l}, \mathfrak{l}' \in \mathbb{I}$ Nachbarn sein, die sich in nur einer Funktionsblock-Implementierung unterscheiden. Dabei ist es unerheblich, welche Implementierungen dieser Block in den beiden System-Implementierungen besitzt. Also

$$U(\mathfrak{l}') := \left\{ \mathfrak{l} \in \mathbb{I} \,\middle|\, \exists i \in V \, \forall k \in V \setminus \{i\} : \; l_k = l'_k \right\}.$$

Auf diese Weise hat man eine einfach zu handhabende, symmetrische, nicht zu viele Nachbarn enthaltende Nachbarschaftsfunktion. Trotzdem bleibt der Lösungsraum zusammenhängend.

Bei der Auswahl der Nachbarschaftsbeziehungen werden auch hier wieder die Wandler unberücksichtigt gelassen, da sich ihre Implementierungen aus

den Implementierungen der Funktionsblöcke ergeben.

Eine Start-Implementierung kann erraten werden, wenn man davon ausgeht, daß bei dem Systemgraphen zwischen den benachbarten Funktionsblöcken alle Wandlertypen verfügbar sind (auch wenn sie nicht kostengünstig implementierbar sind). Einfacher und sinnvoller wäre es aber, eine Start-Implementierung zu wählen, deren Implementierungen aus möglichst vielen gleichen Zeitmodelldomänen besteht, also z. B. eine System-Implementierung, die nur DTSS-Implementierungen enthält. Außerdem ist man aufgrund der Überspezifikation des Systems (siehe Kapitel 2.2) im Besitz einer initialen System-Partitionierung.

Die Start-Temperatur sollte so gewählt werden, daß der Übergang zu einem Nachbarn mit höheren Implementierungskosten mit einer Wahrscheinlichkeit nahe 1 erlaubt wird. Dadurch erreicht man ein zufälliges Durchwandern des Lösungsraums, bei dem alle Lösungen als gleichwertig betrachtet werden.

Anschließend wird die Temperatur langsam gesenkt. Im allgemeinen wird dafür eine geometrische Temperaturkurve gewählt, d. h.

$$T_{n+1} := c\,T_n$$

wobei T_n die Temperatur zum Zeitpunkt n ist, und c eine Konstante. c sollte einen Wert nahe unter 1 besitzen, und wird typischerweise aus dem Intervall $[0,8;\ 0,99]$ gewählt (lt. [1]). Somit wird es mit der Zeit immer unwahrscheinlicher, daß ein Nachbar mit größeren Implementierungskosten als neue Lösung akzeptiert wird. Dadurch wird garantiert, daß nur noch Lösungen akzeptiert werden, die die Kosten verkleinern oder zumindest nur sehr wenig vergrößern. Die Lösung wird beim Abkühlen also in ein Minimum „gedrängt“.

Simulated Annealing wird entweder nach einer bestimmten Anzahl Schritte abgebrochen (indem eine „Gefrier-Temperatur“ festgelgt wird) oder aber, wenn die aktuelle Lösung x seit einer bestimmten Zeit nicht mehr verändert wurde, also schon lange kein Übergang zu einem Nachbarn mehr stattgefunden hat.

Bei der ersten Abbruchbedingung weiß man nicht, ob die erhaltene Lösung zumindest ein *lokales* Minimum ist. Dies garantiert die letztere Abbruchbedingung jedoch mit einer großen Wahrscheinlichkeit (wenn man lange genug gewartet hat, ob die aktuelle Lösung nicht doch noch einen günstigeren Nachbarn besitzt). Der Nachteil, daß man unter Umständen sehr lange warten muß bis Simulated Annealing abbricht, läßt sich durch geschickte Wahl der Anfangstemperatur, der Abkühlungskurve und der Abbruchbedingung vermindern. Hierfür wird in [1] ein Polynomialzeit-Abkühlungsverfahren vorgestellt.

Besonders praktisch ist bei Simulated Annealing, daß keinerlei Ansprüche an die Kostenfunktion gestellt werden. Das bedeutet im Fall von $PART^{\text{opt}}$, daß

ohne größere Leistungseinbußen bzgl. der Laufzeit oder auch beliebigen anderen (turing-berechenbaren) Kostenfunktionen optimiert werden kann. Bei jedem Schritt werden die Kosten für den ausgewählten Nachbarn berechnet und die Implementierungskosten für die aktuelle Implementierung (die aber schon aus dem vorherigen Schritt bekannt sind). Es ist zwar wünschenswert, daß die Kostenfunktion möglichst effizient berechenbar ist (weil das Laufzeitverhalten von Simulated Annealing von der Laufzeit der Kostenfunktion mitbestimmt wird), dies ist aber nicht notwendig.

Auch zusätzliche Randbedingungen lassen sich mit Hilfe der Kostenfunktion berücksichtigen. Jedoch ist darauf zu achten, daß die Randbedingungen nicht zu restriktiv sind, so daß große Teile des Lösungsraums ∞ Kosten besitzen. Da aufgrund der Wahrscheinlichkeitsfunktion keine Implementierung mit ∞ Kosten als neue Lösung gewählt werden kann, könnte der Systemgraph durch zu starke Randbedingungen in mehrere Zusammenhangskomponenten zerfallen. Das würde dann bedeuten, daß Teile des Lösungraums nicht mehr erreichbar wären.

Aufgrund der einfachen Struktur und der durch die Randomisierung erstaunlich guten Optimierungsergebnisse wird Simulated Annealing in vielen Bereichen gerne verwendet. Es ist zu erwarten, daß Simulated Annealing bei geschickter Wahl von Anfangstemperatur, Abkühlungskurve und Abbruchbedingung auch für $PART^{\text{opt}}$ gute Approximationsergebnisse liefert.

Kapitel 11

Tabu Search

Tabu Search zählt, wie auch Simulated Annealing, zu den lokalen Suchverfahren und arbeitet als Heuristik, die man nach einer bestimmten Anzahl an Iterationen abbricht, wobei sie das bis dahin beste berechnete Ergebnis liefert.

11.1 Grundlagen

Tabu-Search basiert auf der Idee des Hill-Climbing. Man ersetzt die aktuelle Lösung durch ihren besten Nachbarn. Wenn man aber zu einem lokalen Optimum x gelangt, kann man die Kosten durch Ersetzung der aktuellen Lösung x mit ihrem besten Nachbarn x' nur verschlechtern. Wenn der Nachbar x' nicht gerade ein Sattelpunkt ist, wird der Nachbar von x' mit den günstigsten Kosten wieder x sein. Damit ist die Strategie in einem lokalen Optimum „gefangen“ und kann den Lösungsraum nicht mehr weiter erforschen.

Um solche Situationen zu verhindern, werden besuchte Lösungen, bzw. Züge[1] *tabu* gesetzt, dürfen also nicht mehr besucht bzw. ausgeführt werden.

Bei Tabu-Search ist das Finden von geeigneten Nachbarschaftsverhältnissen noch wichtiger als bei Simulated Annealing. Denn falls die Lösungen zu viele Nachbarn besitzen, wird übermäßig viel Zeit benötigt, um den Nachbarn mit den niedrigsten Kosten zu finden. Bei dem in Abbildung 10.1 gezeigten Extremfall würde Tabu-Search zwar in einem Schritt die optimale Lösung finden, muß dafür aber alle Lösungen des Lösungsraums miteinander vergleichen. Dies wären bei $PART^{\text{opt}}$ exponentiell viele. Der andere

[1] Unter einem Zug wird nicht der Wechsel von Lösung a nach Lösung b verstanden, sondern die dafür notwendige Veränderung im Lösungsvektor (also z. B. der Wechsel der Implementierung eines bestimmten Funktionsblocks). Somit darf insbesondere auch nicht von Lösung c nach Lösung d gewechselt werden, falls damit der inverse Zug von a nach d ausgeführt wird und dieser tabu gesetzt ist.

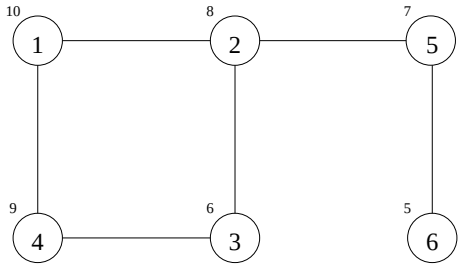


Abbildung 11.1: Lösungsraum (Knotenmenge) mit zugehörigen Kosten (kl. Werte neben den Knoten) und Nachbarschaftsbeziehungen (Kantenmenge)

Extremfall der Nachbarschaften in Abbildung 10.2 läßt Tabu-Search, ebenso wie Simulated Annealing, versagen, weil der Algorithmus die anderen Zusammenhangskomponenten nicht erreichen kann.

Für das tabu-setzen von Lösungen bzw. Zügen gibt es verschiedene Verfahren, die sich deutlich in ihrem Aufwand, aber auch in ihren Ergebnissen unterscheiden [2]:

strict Tabu-Search: Eine bereits besuchte Lösung darf kein zweites mal besucht werden.

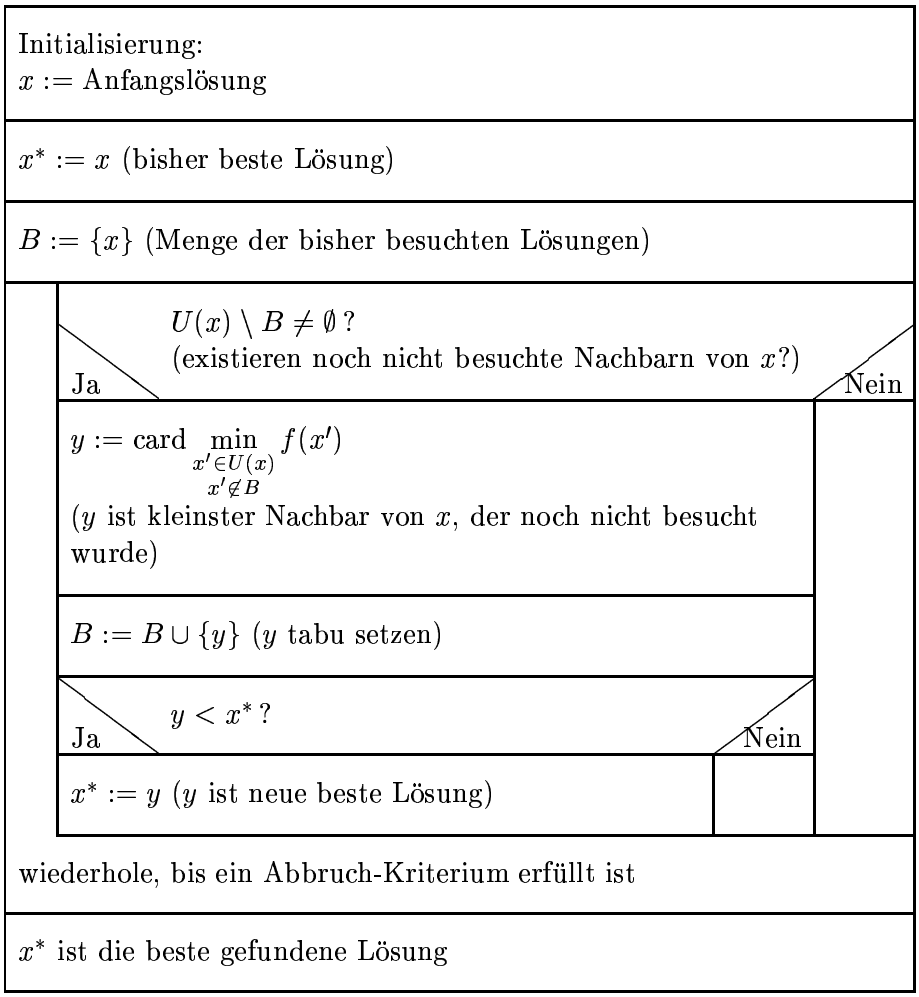
fixed Tabu-Search: Ein ausgeführter Zug darf innerhalb der nächsten n (n konstant) Züge nicht rückgängig gemacht werden.

reactive Tabu-Search: wie fixed Tabu-Search, jedoch ist n nicht konstant, sondern wird vom Algorithmus an das Problem angepaßt.

11.1.1 Strict Tabu-Search

Da keine schon früher besuchte Lösung nochmals betrachtet werden darf, kann es leicht zu Situationen kommen, in denen der Algorithmus abbrechen muß, weil es zu der aktuellen Lösung keinen noch nicht besuchten Nachbarn gibt. Wenn man zum Beispiel in dem in Abbildung 11.1 gezeigten Lösungsraum mit der Anfangslösung 1 beginnt, wird strict Tabu-Search als nächste Lösung den Nachbarn 2 wählen, weil er gegenüber Lösung 4 die geringeren Kosten besitzt. Aus diesem Grund wird im nächsten Schritt die Lösung 3 statt Lösung 5 gewählt. Lösung 1 ist tabu, weil sie bereits besucht wurde. Diese Lösung ist ein lokales Minimum. Da Lösung 2 bereits besucht wurde, kommt als Folgelösung nur noch 4 in Betracht. Im folgenden Schritt muß strict Tabu-Search abbrechen, da seine beiden Nachbarn bereits besucht wurden. Somit ist es nicht möglich, das globale Optimum, nämlich Lösung 6, zu finden.

Der Vorteil von strict Tabu-Search liegt in der einfachen Implementierung. Um die bereits besuchten Lösungen zu vermerken, kann man sich der sog.



Algorithmus 11.1: Strict Tabu-Search

reverse elimination method (REM) bedienen, bei dem die einzelnen Züge in einer geordneten Liste gespeichert werden (siehe z. B. [2]). Es ist aber auch möglich, die besuchten Lösungen in einer Hash-Tabelle abzulegen.

11.1.2 Fixed Tabu-Search

Unter Tabu-Search wird in der Literatur üblicherweise die Variante fixed Tabu-Search verstanden.

Hierbei werden die Invertierungen der letzten n Züge tabu gesetzt, dürfen also im nächsten Schritt nicht verwendet werden. Damit will man erreichen, daß man eine Lösung innerhalb von n Schritten nicht wieder erhalten kann.

Es ist aber nicht notwendig, daß man die Invertierungen der ausgeführten Züge tabu setzt. Je nach Problemstellung ist es durchaus sinnvoll, nicht

die Züge, sondern die Lösungen selbst (wie bei strict Tabu-Search) tabu zu setzen.

Die *Listenlänge* n muß sorgfältig an die Problemgröße und -struktur angepaßt werden. Wird n zu klein gewählt, kann man in einen Zykel um ein lokales Minimum geraten. Wenn man bei dem Beispiel in Abbildung 11.1 eine Listenlänge $n = 4$ wählen würde und die Lösungen selbst tabu setzt, besucht fixed Tabu-Search immer nur die Lösungen 1, 2, 3 und 4, weil der Besuch von Lösung 3 schon wieder erlaubt ist, wenn der Algorithmus bei Lösung 2 ist. Somit wird auch hier das globale Minimum nicht erreicht, weil die Listenlänge zu klein ist.

Wenn n andererseits zu groß gewählt wird, kann es wie bei strict Tabu-Search passieren, daß alle Nachbarn tabu sind und der Algorithmus nicht fortfahren kann. Um das zu verhindern existieren aber Strategien, z. B. daß in diesem Fall derjenige Zug gewählt wird, der am längsten nicht benutzt wurde, also als nächster aller möglichen Züge wieder freigegeben würde.

Fixed Tabu-Search läßt sich auch recht einfach randomisieren, indem man den Algorithmus mit hoher Wahrscheinlichkeit einen erlaubten Zug durchführen läßt und mit geringer Wahrscheinlichkeit einen verbotenen Zug, also einen Nachbarn wählt der eigentlich tabu ist.

Die Verwaltung der tabu gesetzten Züge (bzw. Lösungen) läßt sich recht einfach mittels einer einfach verketteten Liste realisieren. Wenn die Menge der möglichen Züge relativ klein ist, ist aber auch ein Array praktikabel, in dem für alle möglichen Züge (oder deren Inverse) die letzte Ausführungszeit eingetragen ist.

11.1.3 Reactive Tabu-Search

Der Nachteil von fixed Tabu-Search ist, daß man die Listenlänge n à priori festlegen muß und der Erfolg des Algorithmus auch sehr stark von diesem Parameter abhängt.

Daher erhöht bzw. verringert man die Listenlänge anhand der beobachteten Zykel, die im Lösungsraum während der Abarbeitung des Algorithmus durchlaufen werden. Für jede Lösung im Lösungsraum wird gespeichert, wie oft sie und zu welchem Zeitpunkt sie das letzte Mal besucht wurde. Wird eine Lösung öfter als eine bestimmte Schranke r mal besucht, kann man davon ausgehen, daß man einen Zykel durchläuft. Dann erhöht man die Listenlänge n um einen vordefinierten Faktor (also eine geometrische Erhöhung der Listenlänge). Somit muß die Länge der Zykel zwangläufig größer werden und man kann hoffen, daß man auf diese Weise in ein anderes Gebiet des Lösungsraums vordringt und der Zykel nicht mehr durchlaufen wird.

Es ist jedoch möglich, daß selbst die Erhöhung der Listenlänge extrem lange

Zykel, die aufgrund von speziellen Strukturen des Lösungsraums auftreten können, nicht verhindern kann.

Dafür wird eine weitere Schranke r' eingeführt. Wenn eine Lösung öfter als r'-mal besucht wurde, wird ein polynomiell zur durchschnittlichen Zykellänge[2] langer, randomisierter Lauf durch den Lösungsraum ausgeführt. Damit kommt man mit großer Wahrscheinlichkeit aus dem bereits besuchten Gebiet heraus und in einen neuen Lösungs-Teilraum.

Auf diese Weise werden große Gebiete des Lösungsraums besucht. Allerdings wird man so noch keine optimalen Lösungen finden, da man mit großen „Sprüngen" durch den Lösungsraum läuft, weil der Suchraum durch eine große Tabu-Liste zu sehr eingeschränkt wird. Daher sollte die Listenlänge n in geeigneter Weise auch wieder verkleinert werden können, um weitere lokale Minima in den anderen Lösungsraum-Gebieten finden zu können. Wenn die Anzahl der Schritte seit der letzten Erhöhung von n die Größe der durchschnittlichen Zykellänge überschritten hat, kann man davon ausgehen, daß man sich in einem neuen, noch nicht besuchten Gebiet befindet und verringert n um einen kleinen Faktor.

Da die Listenlänge schnell erhöht wird, aber nur langsam (und mit Verzögerung) reduziert wird, kann es (wie bei fixed Tabu-Search) passieren, daß keine neue Lösung mehr erreicht werden kann, weil alle Züge tabu sind. In diesem Fall wird die Listenlänge ebenfalls verringert, so daß immer eine kleine Anzahl an Zügen zur Verfügung steht.

Reactive Tabu-Search ist aufwendiger als fixed oder strict Tabu-Search, weil zusätzlich zur Tabu-Liste alle bisher besuchten Lösungen, deren Besuchs-Häufigkeit und der Zeitpunkt des letzten Besuchs gespeichert werden müssen. Dies wird man natürlich nicht in einem Array machen, da man hierfür i. a. exponentiell viele Felder benötigen würde. Sinnvoll wäre vielmehr eine Hash-Tabelle oder ein Digital Tree (siehe [21]), um die Lösungen zu speichern. Sollte die Binärdarstellung der Lösungen sehr groß sein, aber die Kostenfunktion „nahezu injektiv" (d. h. es gibt nur sehr wenige Lösungen die die gleichen Kosten besitzen), könnte es zur Reduzierung des Speicherplatzbedarfs interessant sein, die Lösungen durch ihre Kosten zu identifizieren.

Außerdem wird reactive Tabu-Search je nach Problemstruktur eine größere Anzahl an Schritten benötigen, als ein optimal eingestelltes fixed Tabu-Search, weil die Tabu-Liste sehr schnell wächst, aber nur langsam verkleinert wird.

Dafür erreicht reactive Tabu-Search aber ein ausgewogenes Verhältnis zwischen einer breitgestreuten Suche im Lösungsraum (diversification) und einer exakten Optimierung von lokalen Minima (intensification).

[2] die Zykellänge läßt sich mit Hilfe des Zeitpunktes des letzten Besuches der Lösung, bestimmen. Mit allen bisher aufgetretenen Zykellängen läßt sich so die durchschnittliche Zykellänge berechnen.

Eine genaue Beschreibung der einzelnen Verfahren mit vergleichenden Testläufen findet sich in [2] und [3].

11.2 Anwendung auf $PART^{\mathbf{opt}}$

Für Tabu Search scheint die bereits für Simulated Annealing verwendete Nachbarschaftsfunktion (siehe Kapitel 10.2) ebenfalls geeignet zu sein. Die Nachbarn einer System-Implementierung lassen sich leicht ermitteln und es existieren nur wenige Nachbarn, so daß es nicht zu aufwendig ist, den günstigsten zu ermitteln. Andererseits kann sehr schnell jede beliebige Implementierung des Systems erreicht werden (nämlich in $|V|$ Schritten).

Es sind zwar alle Varianten von Tabu-Search auf $PART^{\text{opt}}$ anwendbar, aber aufgrund der in [3] vorgestellten Fallbeispiele scheint reactive Tabu-Search speziell für Problemstrukturen mit vielen lokalen Minima sehr effektiv zu sein.

Wenn in einem Schritt die Implementierung l von Block k auf l' verändert wird, sollte nicht nur der Rückwechsel von l' nach l tabu gesetzt werden, sondern der Wechsel zur Implementierung l generell, unabhängig von der aktuellen Implementierung. Denn sonst kann man den tabu-gesetzten Implementierungswechsel mit Hilfe einer dritten Implementierungsmöglichkeit aushebeln (durch $l' \rightsquigarrow l'' \rightsquigarrow l$). Daher sollte die Implementierungsmöglichkeit l von Block k auf die Tabu-Liste gesetzt werden.

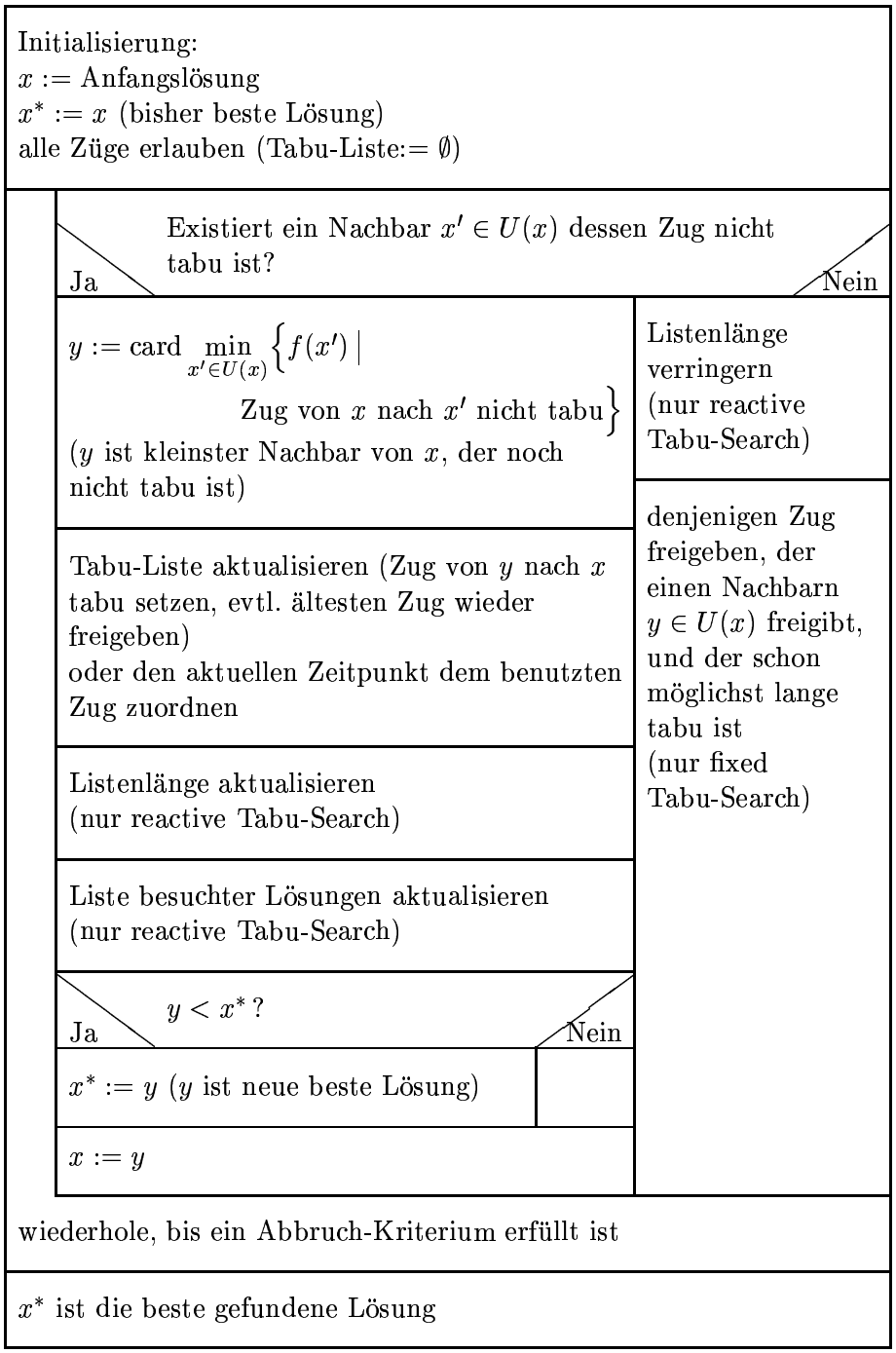
Zur Speicherung der Tabu-Liste bietet sich ein Array an. Das Array umfaßt sämtliche Implementierungsmöglichkeiten aller Funktionsblöcke, besitzt also $\sum_{k \in V} |L_k|$ Elemente. In dem Array wird für jede Implementierungsmöglichkeit aller Blöcke der Zeitpunkt ihrer letzten Verwendung gespeichert. Damit kann man in Konstantzeit überprüfen, ob eine bestimmte Implementierungsmöglichkeit zur Zeit tabu ist oder nicht.

Für die Speicherung der bereits besuchten Lösungen (der Anzahl der Besuche und der Zeitpunkt des letzten Besuchs) sollte ein Hash-Verfahren verwendet werden. Alternativ kann der M-äre Digital Tree (siehe [21]), eine Verallgemeinerung des Digital Tree, verwendet werden. Neben der Anzahl der Besuche und dem Zeitpunkt des letzten Besuchs sollten auch noch die berechneten Kosten für jede besuchte Lösung gespeichert werden. Dies benötigt nur wenig mehr Speicherplatz, reduziert aber unter Umständen die Laufzeit drastisch, da die System-Implementierungen in der Nähe von lokalen Minima vermutlich häufiger vom Algorithmus besucht werden. Daher wird es für die Optimierung von $PART^{\text{opt}}$ vermutlich auch nicht sinnvoll sein, anstatt der System-Implementierung die berechneten Kosten der Implementierung als Hash-Wert bzw. Pfad durch den Baum zu verwenden, denn dann kann die Speicherung der berechneten Kosten natürlich keinen Zeitvorteil ergeben.

Tabu-Search stellt, genauso wie Simulated Annealing, keine Ansprüche an die Kostenfunktion, so daß man auch mit Tabu-Search bzgl. der Laufzeit optimieren kann.

Zusätzliche Randbedingungen können ebenfalls in der Kostenfunktion Berücksichtigung finden, wobei das für Simulated Annealing gesagte auch für Tabu-Search gilt. Wenn jedoch alle Nachbarn, die die Randbedingungen einhalten, tabu gesetzt sind, wird Tabu-Search „notgedrungen“ auch denjenigen Nachbarn als Nachfolge-Implementierung wählen, der die Randbedingung verletzt und somit ∞ Kosten besitzt. Dadurch wird Tabu-Search, im Gegensatz zu Simulated Annealing, als letzte Möglichkeit auch in andere Zusammenhangskomponenten vordringen.

Gute Werte für die verschiedenen Parameter zur Steuerung der Erhöhung und Verringerung der Listenlänge, sowie für die randomisierten Läufe müßten durch Testläufe mit praxisnahen Systemgraphen ermittelt werden. Die Einstellung der Parameter sollte jedoch nicht allzu schwierig sein, da reactive Tabu-Search gegenüber den Parameterwerten relativ robust ist. Von der Güte der Parameter hängt hauptsächlich die Konvergenzgeschwindigkeit ab, und nicht wie beim fixed Tabu-Search die Größe des untersuchten Gebietes.



Algorithmus 11.2: Fixed und Reactive Tabu-Search

Kapitel 12

Genetische Algorithmen

Genetische Algorithmen ahmen Evolutionsprozesse nach, um Optimierungsprobleme zu lösen bzw. zu approximieren. Dafür werden jeweils zwei Lösungen (Individuen) aus einer Menge möglicher Lösungen (Population) selektiert und miteinander gekreuzt. Falls die erzeugte Lösung gut ist, d. h. eine große Fitness besitzt, wird sie „überleben“, ansonsten wird sie wieder entfernt. Dies wird wiederholt, bis das erreichte Ergebnis gut genug ist oder das Verfahren abgebrochen wird.

12.1 Grundlagen

Der grundlegende genetische Algorithmus ist in Algorithmus 12.1 dargestellt.

Je nach Verheiratungsstrategie, Mutation und Auswahl der überlebenden Lösungen werden die Verfahren unterschieden in evolutionäre Programmierung, Evolutions-Strategien und Genetische Algorithmen [34]. Alle diese Varianten werden im allgemeinen unter dem Begriff „Genetische Algorithmen“ subsummiert.

12.1.1 Evolutionäre Programmierung

Bei der evolutionären Programmierung werden alle n Elemente aus $\mathcal{L}$ als Eltern gewählt, jedoch nicht miteinander verheiratet. Vielmehr wird jedes einzelne Individuum zufällig mutiert.

Somit erhält man $2n$ Individuen, von denen nur n überleben, wobei das Überleben eines Individuums von einer auf seiner Fitness basierenden Zufallsfunktion abhängig ist. Die Menge der n überlebenden Individuen wird also probabilistisch bestimmt.

Aufgrund der asexuellen Fortpflanzung der Individuen ist diese Variante gut

Initialisierung: $\mathcal{L}$: vorgegebene Menge möglicher Lösungen (Population)
wähle Individuen aus $\mathcal{L}$ (Eltern)
kreuze die Individuen
mutiere die neu erhaltenen Individuen
aktualisiere die Population $\mathcal{L}$
wiederhole, bis ein Abbruchkriterium erfüllt ist
das beste Individuum aus der Population $\mathcal{L}$ ist die beste bisher gefundene Lösung

Algorithmus 12.1: Genetischer Algorithmus

geeignet für parallele Rechner-Systeme.

12.1.2 Evolutions-Strategien

Die Evolutions-Strategien wurden parallel zur evolutionären Programmierung entwickelt.

Es werden rein zufällig zwei Individuen aus der Population ausgewählt, gekreuzt und das neu entstandene Individuum zufällig mutiert. Dies wird wiederholt, bis man mindestens n neue Individuen erzeugt hat. Aus der $\geq 2n$ Individuen enthaltenden Population überleben (nach einem deterministischen Auswahlprozeß) nur die n fittesten Individuen.

12.1.3 Genetische Algorithmen

Der genetische Algorithmus verheiratet die Eltern zufällig (in Abhängigkeit ihrer Fitness) und erzeugt durch Kreuzung und Mutation n Kinder. Diese n Kinder ersetzen unabhängig von ihrer Fitness die ursprüngliche Population, also ihre Eltern.

Diese drei Verfahrensweisen werden normalerweise nicht in ihrer reinen Form angewendet, sondern derart variiert, daß sie am besten auf das gegebene Problem anwendbar sind.

Die Methoden zur Kreuzung der Individuen, der anschließenden Mutation und auch der Auswahl der fittesten Individuen haben starken Einfluß auf den

Erfolg des Verfahrens und sollten daher ebenfalls an das Problem angepaßt werden.

12.2 Anwendung auf $PART^{opt}$

12.2.1 Wahl der initialen Lösungen

Allen Varianten der genetischen Algorithmen ist gemein, daß man für die Initialisierung mehrere Individuen, d. h. Lösungen benötigt. Diese initialen Lösungen kann man z. B. mit Hilfe eines anderen Optimierungsverfahrens (z. B. Branch & Bound, Simulated Annealing oder Tabu-Search) erzeugen, indem man die erhaltenen Zwischen-Lösungen verwendet. Wenn man davon ausgeht, daß alle Wandler alle Kombinationen der Zeitmodelldomänen unterstützen, kann man die Implementierungen der Funktionsblöcke auch zufällig aus der Menge ihrer Implementierungsmöglichkeiten wählen. Dies ist vermutlich die sinnvollere Alternative, da auf diese Weise eine größere Vielfalt an System-Implementierungen erzeugt wird.

Es ist jedoch auszutesten, wie groß die Population, die man fortpflanzen will, sein sollte. Ist sie zu klein, ist es eventuell nicht möglich, die optimale Lösung zu erreichen. Ist sie andererseits zu groß, dauert das Erzeugen der nächsten Generation zu lange, so daß man viel Zeit mit dem Fortpflanzen schlechter Lösungen verliert.

12.2.2 Wahl der Eltern

Wenn man die Eltern aufgrund ihrer Fitness auswählt, besteht die Gefahr, daß man sich zu sehr auf ein kleines Gebiet im Lösungsraum beschränkt und aus diesem dann nicht mehr herauskommt (also in einem lokalen Minimum gefangen ist). Daher ist für die Optimierung von $PART^{opt}$ vermutlich angebracht, die zu kreuzenden Eltern rein zufällig aus der Population auszuwählen.

12.2.3 Methoden zur Kreuzung der Eltern

Wenn zwei Individuen gekreuzt werden, sollen von jedem ein Teil der „Erbinformationen“ erhalten bleiben.

Um dies zu erreichen, sind drei Standardverfahren üblich:

- Wähle rein zufällig einen Schnitt in der Lösungs-Repräsentation. Der linke Teil des Kind-Erbguts stammt von Elter 1, der rechte Teil von Elter 2 (one-point).

- Wähle mehrere Schnitte in der Lösungs-Repräsentation und wechsle die Vererbung zwischen den beiden Eltern (multi-point).

- Wähle für jede Einheit der Lösungs-Repräsentation rein zufällig aus, ob diese Einheit von Elter 1 oder Elter 2 vererbt wird (uniform).

Alle drei Verfahren scheinen in ihrer Original-Form nicht richtig geeignet, um für $PART^{\text{opt}}$ eine bessere Lösung aus den zwei Eltern zu erzeugen. Das begründet sich in der Tatsache, daß eine Implementierung durch einen Vektor dargestellt wird, bei dem die Anordnung der Elemente nicht mit der Graphenstruktur zusammenhängen muß. Wenn man diese Verfahren bei der Kreuzung anwenden würde, würden mit großer Wahrscheinlichkeit Partitionen gleicher Zeitmodelldomänen der Eltern-Implementierungen nicht vererbt werden. Da aber normalerweise innerhalb einer Zeitmodelldomäne keine Kosten für die Wandler entstehen, wäre eine Vererbung dieser Partitionen durchaus wünschenswert.

Daher erscheint es geschickter, wenn der Systemgraph rein zufällig in zwei Zusammenhangskomponenten zerteilt wird, in der die eine Zusammenhangskomponente die Implementierungen des Elter 1 erbt und die andere Zusammenhangskomponente die Implementierungen von Elter 2. Eine solche Zerlegung läßt sich z. B. erreichen, indem man rein zufällig eine Kante nach der anderen aus dem Systemgraphen entfernt, bis keine Verbindung mehr zwischen den Eingabe- und den Ausgabe-Blöcken existiert. Nun kann man diejenigen Funktionsblöcke, zu denen es noch einen Pfad von einem der Eingabe-Blöcke aus gibt, zu der einen Zusammenhangskomponente zählen, und die, zu denen kein Pfad mehr existiert, zu der anderen. Auf diese Weise wird der Systemgraph einerseits relativ „gerecht“ zerteilt, andererseits wird er tendentiell an einer „schmalen“ Stelle durchgetrennt. Da für einen kleinen Schnitt nur wenige Wandler benötigt werden, um die Implementierungen des einen Elter mit denen des anderen Elter zu verbinden, könnte sich dieser Effekt kostenminimierend auswirken.

12.2.4 Mutation des Kindes

Ob eine zufällige Mutation der durch die Kreuzung erzeugten Implementierung eine Steigerung der Fitness hervorruft, müßte durch Versuche herausgefunden werden.

In [25] wird vorgeschlagen, daß zur Mutation eines Individuums ein einfaches Hill-Climbing-Verfahren verwendet werden sollte. Diese Idee erscheint auch für die Optimierung von $PART^{\text{opt}}$ erfolgreich anwendbar zu sein. Da man nicht erwarten kann, daß der Systemgraph bei der Kreuzung auf ideale Weise in zwei Zusammenhangskomponenten zerfällt, wird die neu erzeugte Implementierung kein lokales Optimum darstellen. Dieses lokale Optimum

kann aber durch ein anschließendes Hill-Climbing erreicht werden. Alternativ ist aber auch ein kurzer Lauf von strict-Tabu-Search oder Simulated Annealing denkbar.

12.2.5 Überleben der Individuen

Die einfachste Methode ist es, stets die n fittesten Individuen, also die n günstigsten Implementierungen, überleben zu lassen.

Bei dieser Strategie ist es auch möglich, zusätzliche Randbedingungen in der Kostenfunktion zu berücksichtigen, indem die Kosten auf ∞ gesetzt werden, falls eine der Randbedingungen verletzt wird. Ein Individuum, das ∞ Kosten besitzt, wird somit nicht überleben, falls mindestens n Individuen existieren, die die Randbedingungen nicht verletzen.

Man sollte aber nicht nach jedem neu erzeugten Individuum die Population aktualisieren, denn dadurch würde die Population unter Umständen ihre Vielfalt verlieren und nur noch sehr ähnliche Implementierungen besitzen. Üblicherweise werden mindestens n neue Implementierungen erzeugt, bevor die Population aktualisiert wird.

Alternativ kann man die n überlebenden Individuen in einem probabilistischen Verfahren auswählen, wobei ein Individuum mit größerer Fitness auch größere Überlebenschancen besitzt, als ein Individuum mit geringer Fitness. Mit dieser Methode läßt sich nochmals die Vielfalt an unterschiedlichen Implementierungen erhöhen. Allerdings muß die Wahrscheinlichkeitsfunktion genau an das Problem angepaßt werden, damit sich im Laufe der Zeit die Qualität der Population auch tatsächlich verbessert.

12.2.6 Abbruchkriterien

Für die Beendigung des genetischen Algorithmus stehen wie auch bei Simulated Annealing und Tabu-Search verschiedene Kriterien zur Verfügung.

Wenn sich über einen gewissen Zeitraum die aktuelle Population durch Kreuzung nicht mehr verbessern läßt, kann man davon ausgehen, daß man (zumindest mit dieser Population) keine bessere Implementierung finden kann. Die beste in der Population enthaltene Implementierung ist dann das gefundene Optimum.

Es ist aber auch zu beliebigen Zeitpunkten möglich, den Algorithmus abzubrechen. Wenn die überlebenden Individuen nicht deterministisch ausgewählt werden, kann es jedoch sein, daß das bisher gefundene Optimum nicht in der aktuellen Population enthalten ist. Daher sollte die beste bisher gefundene Implementierung — wie bei Simulated Annealing und Tabu-Search — zusätzlich gespeichert werden.

Kapitel 13

Greedy-Algorithmen

Greedy-Algorithmen (greedy = gierig, gefräßig) bestimmen ihre Lösung iterativ, wobei in jedem Schritt eine Entscheidung getroffen wird, die zu diesem Zeitpunkt am vielversprechendsten erscheint. Einmal getroffene Entscheidungen werden nicht mehr revidiert [31]. Dadurch sind Greedy-Algorithmen im allgemeinen sehr effizient. Andererseits bedeutet das aber auch, daß man von ihnen keine optimalen Lösungen für komplexe Probleme, wie z. B. $PART^{\text{opt}}$ erwarten kann.

13.1 Einfacher Greedy-Algorithmus

Der denkbar einfachste Greedy-Algorithmus für $PART^{\text{opt}}$ wählt für jeden Funktionsblock k die kostengünstigste Implementierung $\hat{l}_k$ (d. h. $p^{k,\hat{l}_k} = \min_{l_k \in L_k} p^{k,l_k}$). Anschließend wird für alle Wandler die jeweils günstigste gültige (d. h. zu den Nachbarblöcken kompatible) Implementierung gewählt (siehe auch Algorithmus 13.1).

$\forall k \in V$
wähle die günstigste Implementierung für Block k
$\forall (i,j) \in E$
wähle die günstigste, mit den Zeitmodelldomänen der Blöcke i und j kompatible Implementierung für Wandler (i,j)

Algorithmus 13.1: Einfacher Greedy-Algorithmus

Wie in Kapitel 6.1 bereits gezeigt, besitzen Approximationsalgorithmen für $PART^{\text{opt}}$ eine a-posteriori-Approximationsgarantie von

$$\alpha_{post} \leq \frac{\hat{s}}{s_{min}}$$

Für eine lineare Kostenfunktion läßt sich die rechte Seite der Ungleichung in die Funktionsblock-Kosten $\hat{s}^V$, s^V_{min} und die Wandler-Kosten $\hat{s}^E$, s^E_{min} aufspalten, so daß sich die Abschätzung

$$\alpha_{post} \leq \frac{\hat{s}^V + \hat{s}^E}{s^V_{min} + s^E_{min}}$$

ergibt.

Da für alle Funktionsblöcke die jeweils kostengünstigste Implementierung gewählt wurde, gilt $\hat{s}^V = s^V_{min}$ und somit

$$\begin{aligned}
\alpha_{post} &\leq \frac{s^V_{min} + \hat{s}^E}{s^V_{min} + s^E_{min}} \\
&= \frac{s^V_{min}}{s^V_{min} + s^E_{min}} + \frac{\hat{s}^E}{s^V_{min} + s^E_{min}} \\
&\leq 1 + \frac{\hat{s}^E}{s^V_{min} + s^E_{min}} \\
&= 1 + \frac{\hat{s}^E}{s_{min}}.
\end{aligned}$$

$\hat{s}^E$ läßt sich mit s^E_{max} nach oben abschätzen, so daß man als a-priori-Abschätzung

$$\alpha_{pri} \leq 1 + \frac{s^E_{max}}{s_{min}}$$

erhält.

Im allgemeinen wird $s^E_{min} = 0$ sein, da Wandler mit gleicher Ein- und Ausgangs-Zeitmodelldomäne ($d^{(i,j),l}_{in} = d^{(i,j),l}_{out}$) im Normalfall kostenfrei zur Verfügung stehen. Damit kann man die a-priori-Abschätzung noch allgemeiner fassen durch

$$\alpha_{pri} \leq 1 + \frac{s^E_{max}}{s^V_{min}}.$$

Man kann also mit Algorithmus 13.1 bei der Optimierung bzgl. linearer Kostenfunktionen recht gute Ergebnisse erzielen, wenn die maximalen Kosten für die Wandler klein sind im Vergleich zu den minimalen Kosten der Funktionsblöcke. Wenn sämtliche Wandler kostenlos zur Verfügung stünden, wäre der Algorithmus sogar optimal.

Die Laufzeit des Algorithmus beträgt $\mathcal{O}(\sum_{k \in V \cup E} |L_k|)$, da jede Implementierungsmöglichkeit aller Blöcke und Wandler genau einmal betrachtet werden muß. Damit liefert der Algorithmus also in Linearzeit eine Approximation und für bestimmte Systemgraphen sogar eine recht gute.

Man beachte, daß der Algorithmus die Kostenfunktion nicht benutzt, so daß eventuelle Gewichtungen in der Zielfunktion auf die nichtfunktionalen Eigenschaften der einzelnen Blöcke übertragen werden müssen. Dadurch ergibt sich auch, daß der Algorithmus den Verlustfaktor nur für lineare Zielfunktionen garantieren kann.

Ein Nachteil dieser einfachen Heuristik ist, daß jeder Wandler für sämtliche Kombinationen der Ein-/Ausgangs-Zeitmodelldomänen der benachbarten Blöcke eine Implementierungsmöglichkeit zur Verfügung stellen muß, da die Existenz der entsprechenden Wandler bei der Optimierung vorausgesetzt werden.

Außerdem kann der Greedy-Algorithmus keine Randbedingungen berücksichtigen. Dies ergibt sich aus der Tatsache, daß die einmal festgelegte Implementierung eines Blocks nicht mehr revidiert werden kann. Wenn sich also die Wahl der Implementierung eines Blocks im Laufe des Algorithmus als ungünstig herausstellt bzw. durch diese Wahl die Randbedingungen nicht mehr erfüllt werden können, kann diese Wahl nicht mehr rückgängig gemacht werden.

13.2 Erweiterung des Greedy-Algorithmus

Bei dem gerade vorgestellten Greedy-Algorithmus werden die Kosten der Wandler bei der Optimierung überhaupt nicht beachtet. Da die Wandler aber durchaus erhebliche Kosten verursachen können, sollten sie bei der Bestimmung der System-Implementierung berücksichtigt werden.

Dies kann man z. B. erreichen, indem man bei der Wahl der Implementierung eines Funktionsblocks die notwendigen Wandler-Kosten in die Entscheidung mit einbezieht. Denn sobald einige Funktionsblöcke festgelegt sind, muß überprüft werden, ob ein neu festzulegender Funktionsblock seine kostengünstigste Implementierungsmöglichkeit annehmen soll und die Kosten für einen eventuell notwendigen Wandler in Kauf genommen werden, oder ob der Funktionsblock in der gleichen Zeitmodelldomäne wie seine Nachbarblöcke implementiert werden sollte und so die Kosten für Wandler entfallen.

Die Implementierungsmöglichkeit l_k eines Blocks k läßt sich somit durch ihre eigenen Kosten und die für diese Implementierungsmöglichkeit minimal

notwendigen Wandlerkosten bewerten:

$$w^{k,l_k} := p^{k,l_k} + \sum_{(i,k)\in E} \min_{\substack{l_{(i,k)}\in L_{(i,k)} \\ d_{out}^{(i,k),l_{(i,k)}} = d_{in}^{k,l_k}}} p^{(i,k),l_{(i,k)}} + \sum_{(k,j)\in E} \min_{\substack{l_{(k,j)}\in L_{(k,j)} \\ d_{out}^{k,l_k} = d_{in}^{(k,j),l_{(k,j)}}}} p^{(k,j),l_{(k,j)}} \tag{13.1}$$

Falls für eine Implementierungsmöglichkeit kein kompatibler Wandler existieren sollte, ist $w^{k,l_k} = \infty$ (wg. $\min \emptyset := \infty$).

Nachdem ein Funktionsblock k auf die Implementierung $\hat{l}_k$ festgesetzt wurde, können alle inkompatiblen Implementierungsmöglichkeiten seiner benachbarten Wandler (d. h. die Implementierungsmöglichkeiten $l_{(k,j)}$ von Wandler (k,j) mit $d_{out}^{k,\hat{l}_k} \neq d_{in}^{(k,j),l_{(k,j)}}$ bzw. $l_{(i,k)}$ von Wandler (i,k) mit $d_{out}^{(i,k),l_{(i,k)}} \neq d_{in}^{k,\hat{l}_k}$) gestrichen werden. Auf diese Weise werden in Gleichung (13.1) nur diejenigen Wandler-Implementierungsmöglichkeiten berücksichtigt, die noch zur Verfügung stehen. Außerdem stehen nach der Festlegung aller Funktionsblöcke nur noch kompatible Wandler-Implementierungsmöglichkeiten zur Auswahl.

Wie man leicht erkennt, ist bei diesem Verfahren das Optimierungsergebnis abhängig von der Reihenfolge, in der die Implementierungen der Funktionsblöcke festgelegt werden.

Daher sollten die Blöcke mit dem größten Optimierungspotential zuerst festgelegt werden. Zur Bestimmung des Optimierungspotentials von Block k bieten sich die Funktionen

$$o_k := \frac{\max_{l_k\in L_k} w^{k,l_k}}{\min_{l_k\in L_k} w^{k,l_k}}$$

bzw.

$$o_k := \max_{l_k\in L_k} w^{k,l_k} - \min_{l_k\in L_k} w^{k,l_k}$$

an (siehe auch Kapitel 8.2.1). Auf diese Weise werden automatisch diejenigen Funktionsblöcke bevorzugt, bei denen, aufgrund der zur Verfügung stehenden Wandler, nicht alle Zeitmodelldomänen benutzt werden können (weil dann $\max_{l_k\in L_k} w^{k,l_k} = \infty$).

Daraus ergibt sich das in Algorithmus 13.2 angegebene Verfahren.

Die Berechnung aller Kosten $w^{\cdot,\cdot}$ benötigt $\mathcal{O}\left(\sum_{k\in V\cup E} |L_k|\right)$ Zeit, da jede Implementierungsmöglichkeit der Funktionsblöcke genau einmal und der Wandler genau zweimal betrachtet wird. Da bzgl. einer linearen Kostenfunktion optimiert wird, kann man die Anzahl der Implementierungsmöglichkeiten mittels Preprocessing ($\mathcal{O}\left(|V| + |E| + \sum_{k\in V\cup E} |L_k|\right)$ Laufzeit) auf

wähle den noch nicht festgelegten Funktionsblock k mit dem größten Optimierungspotential o_k
wähle die Implementierung $\hat{l}_k$ mit minimalen Kosten ($w^{k,\hat{l}_k} = \min_{l_k \in L_k} w^{k,l_k}$)
entferne inkompatible Implementierungsmöglichkeiten aller benachbarten Wandler $(i,k),(k,j) \in E$
wiederhole, bis alle Blöcke festgelegt sind
$\forall (i,j) \in E$
wähle die günstigste Implementierung $\hat{l}_{(i,j)}$ für Wandler (i,j) ($p^{(i,j),\hat{l}_{(i,j)}} = \min_{l_{(i,j)} \in L_{(i,j)}} p^{(i,j),l_{(i,j)}}$

Algorithmus 13.2: Erweiterter Greedy-Algorithmus

3 bzw. 9 reduzieren, also als konstant ansehen (siehe Kapitel 6.2). Das bedeutet somit einen Zeitbedarf von $\mathcal{O}(|V|+|E|)$ für die Berechnung aller Kosten $w^{\cdot,\cdot}$. Die Berechnung des Optimierungspotential $o_{\cdot}$ aller Blöcke benötigt nochmals $\mathcal{O}(|V|)$ Zeit.

Um jeweils den Funktionsblock mit dem größten Optimierungspotential zu bestimmen, bietet sich eine Priority-Queue (siehe [5, 21]) an. In ihr müssen $|V|$ Funktionsblöcke bzgl. $w^{\cdot,\cdot}$ sortiert werden, so daß das Einfügen und das spätere Entfernen der Blöcke aus der Queue insgesamt $\mathcal{O}(|V| \log |V|)$ Zeit benötigen.

Wenn die Implementierung eines Funktionsblocks festgelegt wurde, werden die Implementierungsmöglichkeiten der benachbarten Wandler aktualisiert. Dies passiert für jeden Wandler zweimal. In diesem Moment muß das Optimierungspotential $o_{\cdot}$ des benachbarten Funktionsblocks (sofern seine Implementierung noch nicht festgelegt wurde) ebenfalls neu berechnet werden. Dafür müssen die entsprechenden Kosten $w^{\cdot,\cdot}$ aktualisiert werden.

Durch die Änderung des Optimierungspotentials muß auch die Priority-Queue aktualisiert werden, was pro Änderung $\mathcal{O}(\log |V|)$ Zeit kostet. Während des gesamten Verfahrens wird also für diesen Aktualisierungsschritt $\mathcal{O}(|E| \log |V| + |E| + |V|)$ Zeit benötigt.

Die Wahl der günstigsten Implementierung für die Wandler dauert insgesamt

$\mathcal{O}(|E|)$, so daß die Gesamtlaufzeit des Algorithmus

$$\mathcal{O}\left(|V| + |E| + \left(\sum_{k \in V \cup E} |L_k|\right) + |E| \log |V| + |V| \log |V|\right)$$
$$= \mathcal{O}\left(\left(\sum_{k \in V \cup E} |L_k|\right) + (|V| + |E|) \log |V|\right)$$

beträgt.

Für den erweiterten Greedy-Algorithmus kann zwar keine bessere a-priori-Approximationsgarantie angegeben werden, es ist aber zu erwarten, daß er bessere Ergebnisse liefert als die einfache Version, die die Wandler-Kosten gar nicht berücksichtigt.

Außerdem kann nicht garantiert werden, daß eine gültige Implementierung gefunden wird, wenn die Wandler nicht sämtliche Zeitmodellkombinationen unterstützen. Die Wahrscheinlichkeit ist aber deutlich größer als bei dem einfachen Greedy-Algorithmus, da Funktionsblöcke, die nicht in allen Zeitmodelldomänen implementierbar sind, das höchste Optimierungspotential zugewiesen bekommen.

Für diese Verbesserungen ist eine etwas höhere Laufzeit durchaus akzeptabel.

Zusätzliche Randbedingungen können aber auch mit diesem Algorithmus nicht berücksichtigt werden.

Kapitel 14

Mehrziel-Optimierung

Häufig reicht es nicht aus, nach nur einem Ziel zu optimieren. Denn ein System, das zwar sehr schnell ist, dafür aber extrem viel Chipfläche benötigt, ist meistens genauso unbrauchbar wie ein flächenoptimales System, welches sehr langsam ist. Im allgemeinen wünscht man sich einen Kompromiß zwischen den Implementierungen, die sich aus den optimalen Lösungen der einzelnen Ziele ergeben.

Wenn zwei Ziele gleiche (oder zumindest nahezu gleiche) optimale Implementierungen besitzen, dann sind diese Implementierungen automatisch geeignete Kompromisse. Ein Beispiel für solche *gekoppelten Ziele* sind Flächenbedarf und Leistungsverbrauch. Häufig werden die einzelnen Zielsetzungen aber zueinander im *Zielkonflikt* stehen (wie z. B. Fläche und Laufzeit oder Fläche und Chipausbeute[1]), so daß die Suche nach einem geeigneten Kompromiß notwendig wird.

14.1 Grundlagen

Die Notation der Mehrziel-Optimierung orientiert sich an der Linearen Programmierung. Anstatt einer einzelnen Zielfunktion m ist jetzt aber ein Zielfunktionsbündel $\mathfrak{m}^\mathsf{T} = (m_1, \ldots, m_k)$ zu minimieren.

$$\begin{aligned} &\text{Min} \begin{pmatrix} m_1(\mathfrak{x}) \\ \vdots \\ m_k(\mathfrak{x}) \end{pmatrix}, \text{ so daß} \\ &\mathfrak{A}\mathfrak{x} \geq \mathfrak{b} \\ &\mathfrak{x} \geq 0 \end{aligned}$$

Die Funktion Min ist nur symbolisch zu verstehen, da das Minimum für skalare Werte definiert ist, aber nicht für Vektoren (da auf Vektoren keine

[1] Für spezielle Systeme können die Beispiele u. U. nicht gelten.

Ordnung existiert).

Daher definiert man den Begriff der *funktional-effizienten Lösung*, der „gute“ Lösungen charakterisiert.

Definition 14.1. Eine Lösung $\mathfrak{x} \in \mathcal{L}(\mathfrak{A}, \mathfrak{b})$ heißt *funktional-effiziente Lösung* oder *Pareto-Optimum*, wenn es keine Lösung $\mathfrak{x}' \in \mathcal{L}(\mathfrak{A}, \mathfrak{b})$ gibt, mit

$$m_i(\mathfrak{x}') \leq m_i(\mathfrak{x}) \quad \forall i = 1, \ldots, k$$
$$m_j(\mathfrak{x}') < m_j(\mathfrak{x}) \quad \text{für mind. ein } j \in \{1, \ldots, k\}.$$

Wenn also $\mathfrak{x}$ eine funktional-effiziente Lösung ist, dann kann man diese Lösung nicht verbessern, ohne daß sie bzgl. mindestens einem Ziel schlechter wird. Häufig wird die Menge der funktional-effizienten Lösungen auch *Pareto-Menge* genannt [10].

Ein sinnvoller Kompromiß wird in dieser Menge zu suchen sein, denn eine andere Lösung ließe sich bzgl. eines oder mehrerer Ziele verbessern, ohne die Ergebnisse der anderen Zielfunktionen zu verschlechtern.

Es stellt sich nun die Frage, wie die Pareto-Menge bestimmt werden kann, bzw. ob diese Menge überhaupt existiert. Die Antwort darauf gibt der folgende Satz:

Satz 14.1 (Effizienztheorem). *Eine Lösung $\mathfrak{x}^* \in \mathcal{L}(\mathfrak{A}, \mathfrak{b})$ ist genau dann funktional-effizient, wenn x^* eine optimale Lösung des linearen Programms*

$$\begin{aligned} &\min \lambda_1 m_1(\mathfrak{x}) + \cdots + \lambda_k m_k(\mathfrak{x}), \textit{ so daß} \\ &\quad \mathfrak{A}\mathfrak{x} \geq \mathfrak{b} \\ &\quad \mathfrak{x} \geq 0 \\ &\quad \lambda_1 + \cdots + \lambda_k = 1 \\ &\quad \lambda_i > 0 \quad \forall i = 1, \ldots, k \end{aligned}$$

ist.

Beweis. Für den Beweis sei auf [12] verwiesen. □

Das bedeutet also, daß man die Mehrziel-Optimierung durchaus auf die Einzelziel-Optimierung reduzieren kann. Allerdings ergibt sich durch diese Abbildung eine parametrische Lösung (mit den Parametern $\lambda_1, \ldots, \lambda_k$), und zwar die Menge aller Linearkombinationen der Zielfunktionen. Geometrisch betrachtet befinden sich die funktional-effizienten Lösungen am Rand der Lösungsmenge (siehe Abbildung 14.1).

Damit sind zwar die als Kompromiß in Frage kommenden Lösungen beschrieben, aber es wurde keine Aussage getroffen, welche Lösung der Pareto-Menge die vom Designer (d. h. dem Entscheidungsträger, der eine geeignete Lösung des Problems sucht) gewünschte ist. Diese Lösung könnte der Designer zwar

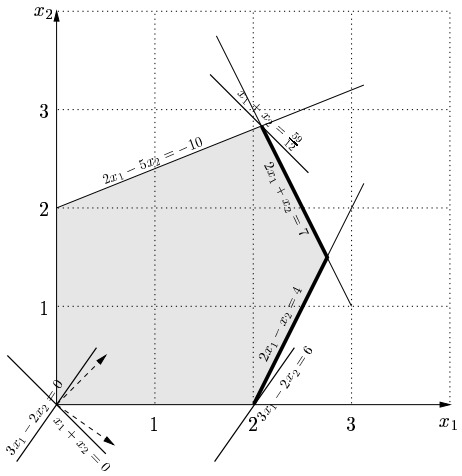


Abbildung 14.1: Pareto-Menge (dicke Linien) eines Mehrkriterien-Problems mit zwei Zielfunktionen

manuell aus der Pareto-Menge auswählen, dies ist aber meistens nicht praktikabel, weil die Pareto-Menge im allgemeinen sehr groß und unüberschaubar wird. Stattdessen wird der Optimierungsalgorithmus den Designer mit Hilfe einer sogenannten Ersatzzielfunktion bei der Auswahl einer geeigneten Lösung zumindest unterstützen.

Im allgemeinen kann die Pareto-Menge bereits aufgrund von Randbedingungen eingeschränkt werden. Zum Beispiel wird die maximal zur Verfügung stehende Chipfläche durch die Größe des Wafers beschränkt. Üblicherweise ist auch die Verzögerungszeit des Systems durch Vorgaben aus der Systemumgebung beschränkt.

Dadurch ergibt sich der Begriff der *befriedigenden Lösung*:

Definition 14.2. Eine Lösung $\mathfrak{x}$ ist eine *befriedigende Lösung*, wenn sie alle Randbedingungen

$$\underline{m}_i \leq m(\mathfrak{x}) \leq \overline{m}_i \quad i = 1, \ldots, k$$

erfüllt. Die Randbedingungen können auch $-\infty$, ∞ sein [40].

Die Menge aller befriedigenden Lösungen wird auch *praktisches Zielgebiet* genannt. (Man beachte, daß eine befriedigende Lösung nicht funktional-effizient sein muß.)

Das praktische Zielgebiet sollte außerdem möglichst groß sein, damit der Entscheidungsspielraum nicht durch zu scharfe Randbedingungen unnötig eingeschränkt wird.

Im folgenden soll davon ausgegangen werden, daß gültige Lösungen eines gegebenen Problems stets aus dem praktischen Zielgebiet stammen. Daher

werden die Randbedingungen nicht mehr weiter aufgeführt, sondern implizit angenommen.

14.2 Wahl eines optimalen Kompromisses

Da, wie bereits erwähnt, die Pareto-Menge im allgemeinen recht groß ist, kann der Designer mit der Auswahl einer geeigneten Lösung aus der Menge aller Kompromißlösungen nicht „alleingelassen“ werden.

Es stellt sich daher die Frage, wie man aus der Menge der funktional-effizienten Lösungen diejenige auswählt, die dem gewünschten Ergebnis am nächsten kommt. Eine solche Lösung heißt *optimale Kompromißlösung.*

Um eine solche Kompromißlösung erhalten zu können, benötigt man eine *Ersatzzielfunktion* (auch *repräsentative Zielfunktion* genannt), die die Zielfunktionen $m_1, \ldots, m_k$ zusammenfaßt. Da die Ersatzzielfunktion eindimensional (und somit meßbar) ist, ist es möglich bzgl. dieser zu optimieren. Die Schwierigkeit besteht nun darin, die sich aus den einzelnen Zielfunktionen ergebenden Optimierungstendenzen in der Ersatzzielfunktion geeignet einzubeziehen. Dabei ist es durchaus erlaubt, daß die Ersatzzielfunktion während des Optimierungsprozesses verändert wird.

Zur Erzeugung von geeigneten Ersatzzielfunktionen haben sich folgende Modelle bewährt:

Nutzenmodelle: die einzelnen Zielfunktionen werden jeweils gewichtet und in der Ersatzzielfunktion zusammengefaßt.

Zielprogrammierung (Goal Programming): es werden die Abstände zu den optimalen Kosten der einzelnen Zielfunktionen in der Ersatzzielfunktion bewertet.

lexikographische Optimierung: es wird iterativ nach den einzelnen Zielfunktionen optimiert und die berechneten optimalen Kosten als Randbedingungen zu der Problembeschreibung hinzugefügt.

14.2.1 Nutzenmodelle

In den Nutzenmodellen ordnet man jedem Einzelziel ein Gewicht zu und wertet so seinen Nutzen.

Im einfachsten Fall werden die einzelnen Ziele m_i mit einem konstanten Gewicht w_i versehen und addiert.

$$m(\mathfrak{x}) := \sum_{i=1}^{k} w_i \, m_i(\mathfrak{x})$$

Da man die Gewichte $w_1, \ldots, w_k$ normieren kann, so daß $\sum_{i=1}^{k} w_i = 1$, kann man sie als die Parameter $\lambda_1, \ldots, \lambda_k$ aus Satz 14.1 interpretieren. Damit ergibt sich, daß die optimale Lösung $\mathfrak{x}$ einer Ersatzzielfunktion m mit konstanter Zielgewichtung stets funktional-effizient ist.

Diese Normierung ist aber nicht notwendig, da ein konstanter Faktor nur die minimalen Kosten verändert, nicht aber die optimale Lösung, die die minimalen Kosten erzeugt.

Beispiel 14.1. Angenommen man sucht einen Kompromiß bzgl. zweier linearer Zielfunktionen m_1 und m_2 und ist bereit, für die Verbesserung von m_1 um eine Einheit, m_2 um drei Einheiten zu verschlechtern. Dann kann man die Ersatzzielfunktion durch

$$m(\mathfrak{x}) = 3m_1(\mathfrak{x}) + 1m_2(\mathfrak{x})$$

beschreiben. Wenn man die Gewichte normiert, erhält man die Ersatzzielfunktion

$$m(\mathfrak{x}) = \frac{3}{4}m_1(\mathfrak{x}) + \frac{1}{4}m_2(\mathfrak{x}).$$

Die optimale Lösung der Ersatzzielfunktion m ist aufgrund von Satz 14.1 funktional-effizient.

Häufig verändert sich aber die Priorität eines einzelnen Ziels, wenn seine Kosten erst einmal ein bestimmtes Niveau unter- oder überschritten haben (siehe auch [30]). Wenn man z. B. einen Kompromiß zwischen Laufzeit und Fläche eines Systems sucht und die Laufzeit des Systems während des Optimierungsprozesses genügend klein geworden ist, wird es sinnvoller sein, im weiteren Verlauf der Optimierung mehr Wert auf die Optimierung der Fläche zu legen, als noch weiter die Minimierung der Laufzeit zu forcieren.

Dies kann man erreichen, indem man die Gewichtung w_i der einzelnen Ziele m_i abhängig von der Lösung $\mathfrak{x}$ macht. Die Gewichte w_i werden durch eine Gewichtsfunktion u_i ersetzt, die von den Kosten der Zielfunktion m_i abhängt.

$$m(\mathfrak{x}) := \sum_{i=1}^{k} u_i \left(m_i(\mathfrak{x})\right)$$

Wenn alle Gewichtsfunktionen u_i, angewendet auf ihre Zielfunktionen m_i streng konvex (d. h. $\frac{\partial^2 (u_i \circ m_i)}{\partial \mathfrak{x}^2}(\mathfrak{x}) > 0 \ \forall \mathfrak{x}$) oder linear sind, ist die minimale Lösung $\mathfrak{x}^*$ der Ersatzzielfunktion m funktional-effizient (siehe auch [40]).

Für andere Gewichtsfunktionen kann keine Aussage über die Pareto-Optimalität ihrer optimalen Lösungen getroffen werden.

Beispiel 14.2. Wenn man die Zielfunktion m_1 aus Beispiel 14.1 nicht mehr konstant gewichten will, sondern ihre Gewichtung bei niedrigen Werten ($z_1(\mathfrak{x}) \leq 10$) kleiner sein soll (nämlich 1 statt 3), würde dies folgende Gewichtsfunktion ergeben (siehe auch Abbildung 14.2):

$$u_1(z) := \begin{cases} 1z & \text{für } z \geq 10 \\ 3z - 20 & \text{sonst} \end{cases}$$

Die Subtraktion im zweiten Fall ist notwendig, damit die Gewichtsfunktion stetig ist.

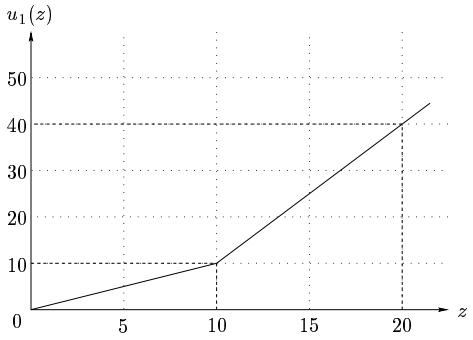


Abbildung 14.2: Gewichtsfunktion $u_1(z)$

m_2 soll weiterhin konstant gewichtet werden, also gilt

$$u_2(z) := 1z.$$

Die Ersatzzielfunktion

$$m(\mathfrak{x}) := u_1(m_1(\mathfrak{x})) + u_2(m_2(\mathfrak{x}))$$

ist konvex, falls die beiden linearen Einzelzielfunktionen m_1 und m_2 monoton steigend sind (läßt sich durch zweifaches Differenzieren überprüfen). Da $u_1 \circ m_1$ und $u_2 \circ m_2$ aber nicht streng konvex sind und $u_1 \circ m_1$ auch nicht linear ist, kann nicht garantiert werden, daß die optimale Lösung $\mathfrak{x}^*$ der Ersatzzielfunktion m funktional-effizient ist.

14.2.2 Zielprogrammierung

Bei der *Zielprogrammierung* (auch *Goal Programming* genannt) werden für jede einzelne Zielfunktion $m_1, \ldots, m_k$ die optimalen Kosten berechnet.

$$s_i^* := \min \left\{ m_i(\mathfrak{x}) \,\middle|\, \mathfrak{x} \in \mathcal{L}(\mathfrak{A}, \mathfrak{b}) \right\}$$

Es werden also k Einzelziel-Optimierungen durchgeführt.

Mit Hilfe einer Metrik[2] d wird nun der Abstand zwischen den Kosten einer Lösung $\mathfrak{x}$ und den jeweiligen Kosten der optimalen Lösungen $\mathfrak{s}^*$ gemessen. Dieser Abstand wird als Güte des Kompromisses gewertet. Wenn der Einzelziel-Vektor „nahe“ bei dem Optimum $\mathfrak{s}^*$ ist, ist die Kompromißlösung gut. Also:

$$m(\mathfrak{x}) := d\left(\mathfrak{m}(\mathfrak{x}), \mathfrak{s}^*\right) = d\Big(\begin{pmatrix} m_1(\mathfrak{x}) \\ \vdots \\ m_k(\mathfrak{x}) \end{pmatrix}, \begin{pmatrix} s_1^* \\ \vdots \\ s_k^* \end{pmatrix}\Big)$$

Üblicherweise wird als Metrik die l_p-Metrik

$$l_p(\mathfrak{x}, \mathfrak{y}) := \|\mathfrak{x} - \mathfrak{y}\|_p := \sqrt[p]{\sum_{i=1}^{n} |x_i - y_i|^p}$$

verwendet, bei der mit größer werdendem p die Abweichungen der einzelnen Kosten $m_1(\mathfrak{x}), \ldots, m_k(\mathfrak{x})$ von den jeweiligen optimalen Kosten $s_1^*, \ldots, s_k^*$ immer stärker „bestraft“ werden.

Falls notwendig, können die einzelnen Summanden $|x_i - y_i|^p$ mit einer Konstanten w_i multipliziert werden, damit die „Strafe“ für ein „schlechtes“ x_i noch zusätzlich gewichtet wird.

Im Fall einer l_1-Metrik reduziert sich die Ersatzzielfunktion auf eine einfache Nutzenfunktion (Abschnitt 14.2.1):

$$\begin{aligned} m(\mathfrak{x}) &= \|\mathfrak{m}(\mathfrak{x}) - \mathfrak{s}^*\|_1 = \sum_{i=1}^{n} |m_i(\mathfrak{x}) - s_i^*| = \sum_{i=1}^{n} (m_i(\mathfrak{x}) - s_i^*) \\ &= \sum_{i=1}^{n} m_i(\mathfrak{x}) - \sum_{i=1}^{n} s_i^* = \text{const} + \sum_{i=1}^{n} m_i(\mathfrak{x}) \end{aligned}$$

Die Entfernung der Absolut-Beträge ist möglich, da bei einem Minimierungsproblem $s_i^* \leq m_i(\mathfrak{x})$ für alle $\mathfrak{x}$ gilt.

Damit ergibt sich also bei der Optimierung mit Hilfe der l_1-Metrik eine funktional-effiziente Lösung.

Für $p = \infty$ (*Maximum-* oder *Tschebyscheff-Norm*) ergibt sich die Ersatzzielfunktion zu

$$m(\mathfrak{x}) = \|\mathfrak{m}(\mathfrak{x}) - \mathfrak{s}^*\|_\infty = \max_{i=1}^{n} |m_i(\mathfrak{x}) - s_i^*| \,.$$

Bei der Minimierung von $m(\mathfrak{x})$ wird also die maximale Abweichung von den optimalen Kosten bzgl. aller Zielfunktionen minimiert. Auch diese Kompromißfunktion liefert ein funktional-effizientes Optimum (siehe auch [12, S. 20–21]).

[2] zur Definition einer Metrik siehe z. B. [13]

Für $1 < p < \infty$,

$$m(\mathfrak{x}) = \|\mathfrak{m}(\mathfrak{x}) - \mathfrak{s}^*\|_p = \sqrt[p]{\sum_{i=1}^{n} |m_i(\mathfrak{x}) - s_i^*|^p},$$

kann keine Aussage über die Pareto-Optimalität einer optimalen Lösung der Ersatzzielfunktion getroffen werden. Das Ergebnis kann funktional-effizient sein, muß es aber nicht (siehe [12, S. 21–22]).

In der Praxis werden hauptsächlich l_p-Metriken mit $p = 1$, $p = 2$ und $p = \infty$ verwendet.

14.2.3 Lexikographische Optimierung

Bei der lexikographischen Optimierung werden die einzelnen Zielfunktionen nach ihrer Wichtigkeit sortiert. Das Problem wird anschließend sukzessive nach den einzelnen Zielfunktionen m_i optimiert und die berechneten optimalen Kosten $s_i^* = m_i(\mathfrak{x}^*)$ werden als neue Randbedingung

$$m_i(\mathfrak{x}) = s_i^*$$

in die nachfolgenden Optimierungen bzgl. $m_{i+1}, \ldots, m_k$ eingefügt.

Das Verfahren kann abgebrochen werden, sobald eine der Optimierungen eine eindeutige Lösung $\mathfrak{x}^*$ liefert. Häufig wird das Verfahren jedoch schon sehr früh abbrechen, weil bei der linearen Optimierung nur endlich viele Zielfunktionen eine Menge an optimalen Lösungen liefert (nämlich wenn die Zielfunktion eine Kante „trifft“ und nicht eine Ecke). Daher sollte man nicht die optimalen Kosten eines Optimierungsergebnisses als neue Randbedingung fordern, sondern gewisse Toleranzen zulassen. Für ein Minimierungsproblem m_i ist die neu erzeugte Randbedingung also

$$m_i(\mathfrak{x}) \leq \frac{1}{c_i}\, s_i^*$$

bzw., wenn m_i eine zu maximierende Einzelzielfunktion ist,

$$m_i(\mathfrak{x}) \geq c_i\, s_i^*,$$

wobei die Konstante $c_i \in [0, 1]$ die Toleranz darstellt, die einzuhalten ist.

Für Zielfunktionen, die als gleichwertig betrachtet werden sollen, ist die lexikographische Optimierung nicht brauchbar. Da sie sich aber nicht durch Nutzenfunktionen darstellen läßt (siehe [40]), ist sie für Probleme, deren Zielfunktionen in Haupt- und Nebenzielfunktionen aufgeteilt werden, als einziges Modell geeignet.

Es sind auch Kombinationen mit den anderen beiden Modellen denkbar. Wenn z. B. die Zielfunktion m_1 wichtig ist (*Hauptzielfunktion*), und die anderen Zielfunktionen $m_2, \ldots, m_k$ gleichwertige *Nebenziele* sind, dann kann

m_1 lexikographisch optimiert werden und anschließend das Optimierungsproblem, mit der zusätzlichen Randbedingung $m_1(\mathfrak{x}) \leq \frac{1}{c_1} s_1^*$, mittels Zielprogrammierung oder Nutzenmodell ein Kompromiß bzgl. $m_2, \ldots, m_k$ gesucht werden.

14.2.4 Anpassen der Wertebereiche

Bei der Einzelzieloptimierung war es unerheblich, in welchem Wertebereich die Zielfunktionen ihre Werte annahmen, da nur nach der Lösung mit den optimalen Kosten gesucht wurde. Bei der Mehrzieloptimierung (speziell bei den Nutzenmodellen und der Zielprogrammierung) hingegen werden die Ergebnisse der einzelnen Zielfunktionen miteinander verglichen. Wenn aber zwei Einzelzielfunktionen unterschiedliche Skalen besitzen (z. B. könnte die eine Zielfunktion einen Bildbereich $[0, 10]$ besitzen, während die andere Zielfunktion einen Bildbereich von $[100, 150]$ besitzt), können die von einer Lösung erzeugten Kosten nicht direkt miteinander verglichen werden. Der Abstand zum Optimum würde bei einer Einzelzielfunktion mit großem Wertebereich-Intervall im allgemeinen größer ausfallen, als bei einer Einzelzielfunktion mit kleinem Wertebereich-Intervall. Daher ist es im allgemeinen sinnvoll, die Einzelzielfunktionen zu normieren, d. h. ihren Wertebereich auf das Intervall $[0, 1]$ abzubilden.

Sei $[a_i, b_i]$ der Wertebereich der Einzelzielfunktion m_i. Dann ist

$$m_i'(\mathfrak{x}) := \frac{m_i(\mathfrak{x}) - a_i}{b_i - a_i}$$

die normierte Einzelzielfunktion von m_i.

Wenn man die normierte Zielfunktion in Nutzenmodellen einsetzt, erkennt man, daß die Normierung durch geeignete Wahl der Gewichte w_i entfallen kann.

$$\begin{aligned} m(\mathfrak{x}) &= \sum_{i=1}^{k} w_i\, m_i'(\mathfrak{x}) = \sum_{i=1}^{k} w_i\, \frac{m_i(\mathfrak{x}) - a_i}{b_i - a_i} = \sum_{i=1}^{k} \frac{w_i}{b_i - a_i}\, (m_i(\mathfrak{x}) - a_i) \\ &= \sum_{i=1}^{k} \Bigg(\underbrace{\frac{w_i}{b_i - a_i}}_{:=\hat{w}_i}\, m_i(\mathfrak{x}) - \underbrace{\frac{w_i\, a_i}{b_i - a_i}}_{=\text{const}} \Bigg) = \text{const} + \sum_{i=1}^{k} \hat{w}_i\, m_i(\mathfrak{x}) \end{aligned}$$

Der Offset $\sum_{i=1}^{k} -\frac{w_i\, a_i}{b_i - a_i}$ ist für die Optimierung mit Nutzenmodellen irrelevant, da er als Konstante keinen Einfluß auf die optimale Lösung $\mathfrak{x}^*$ hat.

Bei der Zielprogrammierung ist es ebenfalls möglich, auf die Normierung zu verzichten, indem man die Gewichtung der einzelnen Summanden der

l_p-Norm entsprechend anpaßt.

$$
\begin{aligned}
m(\mathfrak{x}) &= \sqrt[p]{\sum_{i=1}^{n} w_i \left| m_i'(\mathfrak{x}) - \underbrace{s_i'^*}_{\in\{0,1\}} \right|^p} \\
&= \sqrt[p]{\sum_{i=1}^{n} w_i \left| \frac{m_i(\mathfrak{x}) - a_i}{b_i - a_i} - \underbrace{\frac{s_i^* - a_i}{b_i - a_i}}_{\in\{a_i,b_i\}} \right|^p} \\
&= \sqrt[p]{\sum_{i=1}^{n} w_i \left| \frac{m_i(\mathfrak{x}) - a_i - s_i^* + a_i}{b_i - a_i} \right|^p} \\
&= \sqrt[p]{\sum_{i=1}^{n} \underbrace{\frac{w_i}{|b_i - a_i|^p}}_{:=w_i'} |m_i(\mathfrak{x}) - s_i^*|^p} = \sqrt[p]{\sum_{i=1}^{n} \hat{w}_i \, |m_i(\mathfrak{x}) - s_i^*|^p}
\end{aligned}
$$

Bei der lexikographischen Optimierung ist ein Anpassen der Wertebereiche grundsätzlich nicht notwendig, da die Einzelziele nacheinander optimiert werden, und somit ein Vergleich zwischen den Ergebnissen der einzelnen Zielfunktionen nicht stattfindet.

Somit ist eine Normierung der Einzelzielfunktionen bei den drei vorgestellten Kompromißmodellen nicht notwendig. Trotzdem sollte man genaue Kenntnis über das Verhalten und insbesondere den Wertebereich der einzelnen Zielfunktionen besitzen, damit die Gewichte entsprechend angepaßt werden können.

14.3 Anwendung auf $PART^{\mathbf{opt}}$

Die drei verschiedenen Modelle (Nutzenmodell, Zielprogrammierung und lexikographische Optimierung) lassen sich direkt auf $PART^{\mathrm{opt}}$ übertragen. Die dadurch entstehende Ersatzzielfunktion kann anschließend von einem Einzelziel-Optimierungsalgorithmus oder einer Heuristik minimiert werden. Da die Ersatzzielfunktion unter Umständen recht komplex wird, und eventuell auch zusätzliche Randbedingungen erfüllt werden müssen, kann aber nicht jedes Optimierungsverfahren für jedes Modell eingesetzt werden.

14.3.1 Nutzenmodelle

Das Nutzenmodell läßt sich auf $PART^{\mathrm{opt}}$ recht einfach anwenden, da eine Ersatzzielfunktion erzeugt wird, mit deren Hilfe man das Mehrziel-Optimierungsproblem $PART_{\mathfrak{m}}^{\mathrm{opt}}(\mathfrak{S})$ (mit $\mathfrak{m} = (m_1, \ldots, m_k)^{\mathsf{T}}$) des Systemgraphen $\mathfrak{S}$ auf ein Einziel-Optimierungsproblem $PART_{m}^{\mathrm{opt}}(\mathfrak{S})$ reduziert.

Eingabe: Systemgraph $\mathfrak{S}$ Zielfunktionen $m_1, \ldots, m_k$ Gewichte $w_1, \ldots, w_k$, bzw. Gewichtsfunktionen $u_1, \ldots, u_k$
erzeuge Ersatzzielfunktion $$m(\mathfrak{x}) = \sum_{i=1}^{k} w_i \, m_i(\mathfrak{x})$$ bzw. $$m(\mathfrak{x}) = \sum_{i=1}^{k} u_i(m_i(\mathfrak{x}))$$
optimiere $PART_m^{\mathrm{opt}}(\mathfrak{S})$ bzgl. der Ersatzzielfunktion m mit einem Einziel-Optimierungsverfahren
Ausgabe: Kompromißlösung $\mathfrak{x}^*$

Algorithmus 14.1: Mehrzieloptimierung mit dem Nutzenmodell

Wenn eine konstante Zielgewichtung ($m(\mathfrak{x}) = \sum_{i=1}^{k} w_i \, m_i(\mathfrak{x})$) verwendet wird und die einzelnen Zielfunktionen $m_1, \ldots, m_k$ linear sind, kann $PART_m^{\mathrm{opt}}(\mathfrak{S})$ mit Hilfe der linearen Programmierung oder der dynamischen Programmierung exakt gelöst werden, da die Ersatzzielfunktion m in diesem Fall linear ist.

Falls Zielgewichtsfunktionen $u_1, \ldots, u_k$ verwendet werden oder die Einzelzielfunktionen $m_1, \ldots, m_k$ nicht linear sind, können trotzdem noch Simulated Annealing, Tabu Search und Genetische Algorithmen angewendet werden, da diese Verfahren keine Ansprüche an die Kostenfunktion stellen.

Branch & Bound kann ebenfalls verwendet werden, jedoch müssen die Verfahren zur Bestimmung der unteren Schranke, sowie die Strategie zur Auswahl der Branch-Variablen (siehe Kapitel 8) entsprechend angepaßt werden. Die Abschätzung der unteren Schranke kann z. B. erfolgen, indem man für jede Einzelzielfunktion die untere Schranke der Kosten berechnet und die unteren Schranken dann entsprechend der Ersatzzielfunktion gewichtet und summiert. Die Branch-Variable kann z. B. ausgewählt werden, indem man die nichtfunktionalen Eigenschaften des Funktionsblocks geeignet gewichtet,

zu einer neuen nichtfunktionalen Eigenschaft zusammenfaßt und mit dieser die Branch-Variablen auswählt.

Wie bereits in Abschnitt 14.2.4 erwähnt wurde, muß besonderes Augenmerk auf die Bestimmung der Gewichte bzw. der Gewichtsfunktionen gelegt werden, da diese für die Qualität der berechneten Kompromißlösung entscheidend sind.

Sinnvollerweise berechnet man mehrere Kompromißlösungen durch leichte Variation der Gewichte, bzw. der Gewichtsfunktionen und wählt anschließend aus diesen die beste Lösung aus.

14.3.2 Zielprogrammierung

Sofern nicht die l_1-Metrik als Metrik d für die Zielprogrammierung gewählt wird (deren Äquivalenz zum Nutzenmodell bereits in Abschnitt 14.2.2 gezeigt wurde), kann $PART_m^{\text{opt}}$ nicht mit Hilfe der linearen Programmierung oder der dynamischen Programmierung gelöst werden, weil die Ersatzzielfunktion $m(\mathfrak{x}) := d\,(\mathfrak{m}(\mathfrak{x}), \mathfrak{s}^*)$ nicht linear ist.

Wie auch bei den Nutzenfunktionen, läßt sich die Ersatzzielfunktion aber mit Simulated Annealing, Tabu Search oder genetischen Algorithmen optimieren, da bei diesen Verfahren die Kostenfunktion nicht linear sein muß.

Jedoch sollten die unterschiedlichen Wertebereiche der einzelnen Zielfunktionen in der Ersatzzielfunktion berücksichtigt werden, weil sonst kein sinnvoller Kompromiß gefunden werden kann. Um den Wertebereich $[a_i, b_i]$ einer Zielfunktion m_i zu ermitteln, muß das Problem bzgl. dieser Zielfunktion minimiert *und* maximiert werden. Eine exakte Bestimmung des Wertebereichs ist aufgrund der $\mathcal{NP}$-Vollständigkeit von $PART^{\text{opt}}$ im allgemeinen nicht mit akzeptablem Zeitaufwand möglich.

Jedoch sollte eine relativ genaue Abschätzung ausreichend sein, da man normalerweise auch keinen exakt berechneten Kompromiß benötigt. Dies wird durch die Tatsache, daß die Zielprogrammierung im allgemeinen keine funktional-effiziente Kompromißlösung liefert, zusätzlich bekräftigt.

Eine solche Abschätzung des Wertebereichs kann man bei linearen Einzelzielfunktionen z. B. erreichen, indem man das linearisierte ganzzahlige Programm von $PART_{m_i}^{\text{opt}}(\mathfrak{S})$ für alle $i = 1, \ldots, k$ zweimal löst (einmal für die untere Schranke und einmal für die obere Schranke).

Alternativ kann auch Branch & Bound verwendet werden, indem man den Algorithmus nach einer gewissen Zeit abbricht und aus den Kosten der besten bisher berechneten Lösung und der aktuellen Approximationsgarantie eine obere bzw. untere Schranke (je nach Maximierung bzw. Minimierung des Problems) berechnet. Da die tatsächlichen Kosten für die optimale Lösung zwischen der oberen (bzw. unteren) Schranke und dem bisher berechneten Optimum liegt, erscheint der Mittelwert zwischen den beiden Ko-

Eingabe:
Systemgraph $\mathfrak{S}$
Zielfunktionen $m_1, \ldots, m_k$
Gewichte $w_1, \ldots, w_k$
p (zur Bestimmung der l_p-Metrik)

Für alle Zielfunktionen m_i

- berechne minimale Kosten a_i von m_i
- berechne maximale Kosten b_i von m_i
- berechne optimale Kosten s_i^* von m_i

erzeuge Ersatzzielfunktion

$$m(\mathfrak{x}) = \sqrt[p]{\sum_{i=1}^{n} \frac{w_i}{|b_i - a_i|^p} \; |m_i(\mathfrak{x}) - s_i^*|^p}$$

optimiere $PART_m^{\mathrm{opt}}(\mathfrak{S})$ bzgl. der Ersatzzielfunktion m
mit einem Einziel-Optimierungsverfahren

Ausgabe:
Kompromißlösung $\mathfrak{x}^*$

Algorithmus 14.2: Mehrzieloptimierung durch Zielprogrammierung

sten eine geeignete Abschätzung für die obere (bzw. untere) Intervallgrenze zu sein.

Die optimalen Kosten s_i^* der Einzelzielfunktion m_i sollten aber in jedem Fall die optimalen Kosten oder bei einem Minimierungsproblem die untere Schranke (d. h. $s_i^* \geq a_i$) bzw. bei einem Maximierungsproblem die obere Schranke (d. h. $s_i^* \leq b_i$) sein, damit eine Lösung, deren Kosten besser sind als s_i^* — aufgrund der Messung des Abstandes zwischen s_i^* und a_i (bzw. b_i — nicht schlechter gewertet wird, als die schlechtere, s_i^* erzeugende Lösung.

Als letzte Möglichkeit bliebe noch, daß der Designer selbst die Wertebereiche der Einzelzielfunktionen abschätzt oder geeignete Schätzfunktionen angibt.

Allerdings ist zu beachten, daß die Qualität des Kompromisses stark von der Genauigkeit der Wertebereich-Abschätzungen abhängt. Daher sollte,

sofern möglich, die Abschätzung mit einem linearisierten ganzzahligen Programm erfolgen, da damit (in polynomieller Zeit) die genauesten Ergebnisse zu erwarten sind.

Zusätzlich ist es möglich, bestimmte Zielsetzungen zu bevorzugen, indem eine Gewichtung der Einzelzielfunktionen mittels der Gewichte $w_1, \ldots, w_k$ vorgenommen wird.

Wie in Abschnitt 14.2.2 bereits erwähnt, kann bei der Wahl einer l_p-Metrik mit $1 < p < \infty$ nicht garantiert werden, daß der optimale Kompromiß funktional-effizient ist. Dies ist aber kein wirklicher Nachteil der Zielprogrammierung, da für die Optimierung bzgl. der Ersatzzielfunktion (aufgrund der $\mathcal{NP}$-Vollständigkeit von $PART^{\text{opt}}$) vermutlich eine Heuristik verwendet werden wird, die im allgemeinen keine optimale Lösung, und somit auch keine funktional-effiziente Lösung liefert.

Aufgrund der erforderlichen Berechnung der Wertebereiche werden, sofern die Wertebereiche nicht auf andere Weise abgeschätzt werden, insgesamt $2k+1$ Optimierungen ausgeführt. Das macht die Zielprogrammierung zwar zu einem sehr aufwendigen Verfahren, dafür sind aber keine genauen Kenntnisse über die Kostenfunktionen und den Systemgraphen notwendig.

14.3.3 Lexikographische Optimierung

Bei der lexikographischen Optimierung werden die einzelnen Ziele nacheinander optimiert. Deshalb wird keine Ersatzzielfunktion benötigt, wie bei den Nutzenmodellen oder der Zielprogrammierung.

Allerdings werden k Optimierungsschritte durchgeführt, wobei mit jedem Schritt das Problem, aufgrund der neu hinzukommenden Randbedingungen, schwieriger wird.

Außerdem lassen sich die Randbedingungen nicht in jedem Optimierungsverfahren berücksichtigen. Simulated Annealing ist z. B. für die lexikographische Optimierung nicht anwendbar, weil hierbei die Randbedingungen mit Hilfe der Kostenfunktion eingehalten werden (durch $m(\mathfrak{x}) := \infty$, falls mind. eine der Randbedingungen nicht erfüllt ist). Eine Nachbar-Implementierung, die ∞ Kosten hat, kann aber aufgrund der Wahrscheinlichkeitsverteilung von Simulated Annealing keine Nachfolge-Lösung werden. Dies bedeutet wiederum, daß der Lösungsraum durch viele Randbedingungen derart stark eingeschränkt wird, daß er im Hinblick auf die Nachbarschaftsbeziehungen in mehrere Zusammenhangskomponenten zerfällt, und ein Großteil der Lösungen gar nicht mehr erreichbar ist.

Tabu Search ist für die lexikographische Optimierung zwar anwendbar, wird aber vermutlich viel Zeit benötigen, um akzeptable Ergebnisse zu liefern. Dies ergibt sich aus der Tatsache, daß stets der günstigste nicht tabu-gesetzte Nachbar als Nachfolge-Implementierung gewählt wird, und somit die Im-

Eingabe:
Systemgraph $\mathfrak{S}$
Zielfunktionen $m_1, \ldots, m_k$
Toleranzen $c_1, \ldots, c_k$

Für alle $i = 1, \ldots, k$

- optimiere $PART^{\text{opt}}_{m_i}(\mathfrak{S})$ unter Berücksichtigung der Randbedingungen bzgl. der Einzelzielfunktion m_i mit einem Einziel-Optimierungsverfahren (die berechnete Lösung sei $\mathfrak{x}^*$)
- $s_i^* := m_i(\mathfrak{x}^*)$
- füge neue Randbedingung

  $$m_i(\mathfrak{x}) \leq \frac{1}{c_i}\, s_i^* \quad \text{(falls } m_i \text{ zu minimieren war)}$$

  bzw.

  $$m_i(\mathfrak{x}) \geq c_i\, s_i^* \quad \text{(falls } m_i \text{ zu maximieren war)}$$

  dem Optimierungsproblem hinzu

Ausgabe:
Kompromißlösung $\mathfrak{x}^*$

Algorithmus 14.3: Mehrzieloptimierung durch lexikographische Optimierung

plementierungen mit ∞ Kosten immer nur als letzte Möglichkeit in Frage kommen. Dadurch ergeben sich unter Umständen sehr lange Laufzeiten, um von einem lokalen Optimum zu einem anderen zu gelangen.

Die dynamische Programmierung für $PART^{\text{opt}}$ benötigt eine lineare Kostenfunktion und kann keine Randbedingungen berücksichtigen. Daher ist sie ebenfalls nicht für die lexikographische Optimierung geeignet.

Für die lineare Programmierung hingegen kann man die Randbedingungen als zusätzliche Nebenbedingungen auffassen. Wenn alle Einzelzielfunktionen $m_1, \ldots, m_k$ linear sind, sind auch die erzeugten Randbedingungen lineare Funktionen, können also als Nebenbedingungen verwendet werden.

Falls nicht alle Einzelzielfunktionen linear sind, steht nur Branch & Bound und (mit den bereits genannten Einschränkungen) Tabu Search zur Verfü-

gung. Zur Berechnung der unteren Schranke und der Branch-Variablen gelten bei der lexikographischen Optimierung die gleichen Überlegungen wie für das Nutzenmodell (siehe Abschnitt 14.3.1).

Es ist nicht möglich eine Empfehlung auszusprechen, welches der drei Modelle zur Mehrzieloptimierung am besten geeignet ist, um $PART_{\mathrm{m}}^{\mathrm{opt}}$ zu lösen. Vielmehr haben die einzelnen Modelle grundlegend verschiedene Zielsetzungen: Das Nutzenmodell optimiert $PART_{\mathrm{m}}^{\mathrm{opt}}$, wobei die Kosten der Einzelzielfunktionen in einem bestimmten Verhältnis zueinander stehen. Die Zielprogrammierung sucht eine Kompromißlösung derart, daß die Abweichungen der einzelnen Zielfunktionen von ihren jeweiligen optimalen Kosten möglichst minimal wird. Und die lexikographische Optimierung findet ihre Anwendung, wenn die einzelnen Ziele nicht gleichwertig sind, sondern unterschiedliche Prioritäten besitzen.

Aus diesem Grund muß die Wahl des Kompromißmodells und des Optimierungsverfahrens von der Zielsetzung der Optimierung, von der Beschaffenheit der Einzelzielfunktionen (linear, nicht linear, ...) und von der Kenntnis über den Systemgraphen (z. B. um den Wertebereich der Einzelzielfunktionen mit einfachen Methoden bestimmen zu können) abhängig gemacht werden.

14.4 Interaktive Optimierungsverfahren

Häufig ist es schwierig zu entscheiden, ob eine berechnete Kompromißlösung akzeptiert werden sollte, oder ob ein noch besserer Kompromiß erreicht werden kann. Dies begründet sich in der Tatsache, daß der Designer beim Festlegen der Gewichte vor der Benutzung eines Optimierungsalgorithmus noch keine Informationen über den Lösungsraum besitzt und somit das Gebiet der Kompromißlösungen nur schwer abschätzen kann.

Interaktive Verfahren schlagen deshalb dem Designer eine Kompromißlösung vor, und dieser kann daraufhin die Präferenzen, d. h. die Zielsetzungen verändern. Dieser Wechsel zwischen Berechnung einer Kompromißlösung und Anpassung der Präferenzen wird solange fortgesetzt, bis der Designer mit dem Kompromiß zufrieden ist.

Im Bereich der Multi-Criteria-Analyse existieren verschiedene Verfahren, die eine Interaktion mit dem Designer erlauben. Diese Verfahren stellen jedoch im allgemeinen hohe Ansprüche an die einzelnen Zielfunktionen und den Lösungsraum.

Zum Beispiel verlangt das *Verfahren von Geoffrion, Dyer und Feinberg (GDF)* (siehe z. B. [40]) die Differenzierbarkeit, Monotonie und Konkavität der einzelnen Zielfunktionen. Da die Zielfunktionen für die Optimierung der Systemgraphen nur auf diskreten Werten (nämlich den Implementie-

Eingabe: Systemgraph $\mathfrak{S}$ Zielfunktionen $m_1, \ldots, m_k$ Gewichte $w_1, \ldots, w_k$ p (zur Bestimmung der l_p-Metrik)
Berechne Kompromißlösung x^* mit Hilfe der Zielprogrammierung
aktualisiere Randbedingungen und Gewichte des Optimierungsproblems (interaktiv)
wiederhole bis x^* akzeptiert wird
Ausgabe: Kompromißlösung $\mathfrak{x}^*$

Algorithmus 14.4: Interaktive Mehrzieloptimierung mit Hilfe der Zielprogrammierung

rungsmöglichkeiten) definiert sind, sind diese Forderungen im allgemeinen nicht erfüllbar.

Das *Verfahren nach Zionts und Wallenius* (siehe z. B. [39, 40]) benötigt zwar keine differenzierbaren Einzelzielfunktionen, jedoch müssen die Zielfunktionen linear sein. Da bei der Optimierung des Systemgraphen nach mehreren Zielsetzung üblicherweise auch die Minimierung der Laufzeit ein Ziel sein wird, ist dieses Verfahren nur bedingt einsetzbar.

Es ist aber möglich, die in Kapitel 14.3 vorgestellte Zielprogrammierung in ein interaktives Verfahren umzuwandeln. Hierfür wird mit Hilfe der Zielprogrammierung eine erste Kompromißlösung berechnet. Wenn der Designer ein bestimmtes Ziel verbessern will, so kann er eine zusätzliche Randbedingung einführen, die einen bestimmten Wertebereich dieses Einzelziels — durch Einschränkung des befriedigenden Lösungsraums — erzwingt. Zusätzlich ist es auch möglich, die einzelnen Gewichte der Zielfunktionen zu variieren. Nun wird eine neue Kompromißlösung berechnet, die aufgrund der Randbedingung das festgelegte praktische Zielgebiet einhält. Falls keine gültige Lösung existiert, bricht das Verfahren ab. Es ist aber auch möglich, Randbedingungen während des Verfahrens zu entfernen, falls diese zu restriktiv sind. Die beiden Schritte der Berechnung einer Kompromißlösung und der Wahl neuer Randbedingungen werden solange wiederholt, bis der Designer die Kompromißlösung akzeptiert (siehe Algorithmus 14.4).

Zur Berechnung der Kompromißlösungen durch die Zielprogrammierung ist zu beachten, daß aufgrund der zusätzlichen Randbedingungen, die während des Verfahrens eventuell hinzugefügt werden, nicht alle Einzelziel-Optimierungsverfahren anwendbar sind.

Dieses Verfahren erfordert präzisere Entscheidungen vom Designer als andere interaktive Verfahren, weil neue Randbedingungen unter Umständen zu restriktiv sein können oder aber auch zu schwach. Daher ist es wichtig, daß getroffene Entscheidungen notfalls wieder rückgängig gemacht werden können. Andererseits stellt dieses interaktive Verfahren keine weiteren Anforderungen an die einzelnen Zielfunktionen und kann mit verschiedenen Einzelziel-Optimierungsverfahren betrieben werden.

Kapitel 15

Ergebnisse

15.1 Zusammenfassung

In der vorliegenden Diplom-Arbeit wurde das in [19] vorgeschlagene Modell zur Partitionierung hybrider Systeme diskutiert und die Anwendbarkeit verschiedener Optimierungsverfahren auf das Problem untersucht.

Hierbei wurde der Systemgraph um die Ein-/Ausgabe-Blöcke erweitert, damit der Systemgraph eine konsistente Erweiterung des Graphen-Modells darstellt. Weiterhin wurden die Probleme und Einschränkungen aufgezeigt, die sich bei der Abbildung eines hybriden Systems auf den Systemgraphen ergeben.

Im theoretischen Teil dieser Arbeit wurde bewiesen, daß das Optimierungsproblem des Systemgraphen $PART^{\text{opt}}$ $\mathcal{NP}$-vollständig ist, und daß es keinen polynomiellen Approximationsalgorithmus geben kann, der $PART^{\text{opt}}$ bis auf einen konstanten Verlustfaktor approximiert. Aufgrund der Literatur-Recherche hat sich ergeben, daß es sich bei $PART^{\text{opt}}$ vermutlich um ein völlig neuartiges graphentheoretisches Optimierungsproblem handelt. Somit war eine sinnvolle Reduktion auf bekannte Probleme nicht möglich.

Im praktischen Teil wurden verschiedene Optimierungsalgorithmen vorgestellt und an $PART^{\text{opt}}$ angepaßt.

Aus der Klasse der exakten Optimierungsverfahren wurden die ganzzahlige lineare Programmierung, das Branch & Bound-Verfahren und die dynamische Programmierung vorgestellt. Dabei wurde unter anderem gezeigt, daß sich Systeme mit wenig Parallelität mittels dynamischer Programmierung effizient optimieren lassen bzgl. einer linearen Zielfunktion (wie z. B. minimale Chipfläche). Außerdem ergab sich, daß Branch & Bound ein sehr mächtiges Verfahren ist, um eine optimale Implementierung eines Systemgraphen zu finden, auch wenn die Zielfunktion kompliziert sein sollte und zusätzliche Randbedingungen (wie z. B. maximale Verzögerungszeit oder maximale

Chipfläche) an die optimale Lösung gestellt werden. Die ganzzahlige lineare Programmierung kann nicht so gut an die speziellen Eigenschaften von $PART^{\text{opt}}$ angepaßt werden und ihre Anwendung ist aus Komplexitätsgründen auf lineare Kostenfunktionen beschränkt. Jedoch läßt sich für eine lineare Kostenfunktion m mit dem linearisierten Programm von $PART_m^{\text{opt}}$ mit polynomiellem Zeitaufwand eine relativ genaue untere Schranke der optimalen Implementierungskosten ermitteln.

In der Klasse der Heuristiken wurden Simulated Annealing, Tabu-Search und genetische Algorithmen an $PART^{\text{opt}}$ angepaßt. Alle drei Algorithmen benötigen eine Start-Implementierung, die sie schrittweise zu verbessern versuchen. Für Simulated Annealing und Tabu-Search stellt dies kein Problem dar, denn aufgrund der Überspezifikation liefert der Designer während des Entwurfs automatisch eine mögliche System-Implementierung. Die genetischen Algorithmen benötigen jedoch mehrere Start-Implementierungen, die miteinander gekreuzt werden. In diesem Fall muß man auf zufällig erzeugte Implementierungen zurückgreifen. Alle drei Algorithmen haben den Vorteil, daß man sie zu beliebigen Zeitpunkten abbrechen kann und die beste bisher gefundene Implementierung erhält.

Die vorgestellten Heuristiken vertragen zusätzliche Randbedingungen bis zu einem gewissen Grad recht gut, da Implementierungen, die die Randbedingungen verletzen, nicht als neue Lösung ausgewählt werden. Sind die Restriktionen aber so groß, daß verschiedene Lösungsgebiete durch Bereiche, die die Randbedingungen verletzen, voneinander getrennt werden, so versagen unter Umständen diese Verfahren.

Weiterhin wurden zwei selbstentwickelte Greedy-Algorithmen vorgestellt. Mit diesen beiden Verfahren kann mit sehr geringem Zeitaufwand eine Implementierung des Systemgraphen gefunden werden, vorausgesetzt, es sind die notwendigen Wandler vorhanden. Die Auswahl der Implementierungsmöglichkeiten wurde speziell auf lineare Kostenfunktionen hin optimiert, und für diese kann sogar eine a-priori-Approximationsgarantie berechnet werden.

Vielfach ist es nicht ausreichend, nach nur einem Ziel zu optimieren, sondern es muß ein Kompromiß zwischen verschiedenen konfliktionären Zielen gefunden werden. Hierfür wurden verschiedene Modelle aufgezeigt, die die einzelnen Ziele in einer Ersatzzielfunktion zusammenfassen, die dann in einem Einzelziel-Optimierungsverfahren verwendet werden kann. Es wurde auch diskutiert, welche Einzelziel-Optimierungsverfahren für die einzelnen Kompromißmodelle in Frage kommen, da aufgrund der speziellen Eigenschaften der Ersatzzielfunktionen nicht jedes Verfahren angewendet werden kann.

15.2 Optimierungssoftware

Eine Aufgabe dieser Arbeit war die Untersuchung existierender Software auf ihre Anwendbarkeit für die Optimierung von Systemgraphen. Da das Systemgraphen-Modell auf kein anderes Optimierungsmodell abgebildet werden konnte, kann es entsprechend auch keine Software geben, auf die das vorliegende Optimierungsproblem in einfacher Weise abgebildet werden könnte.

Allgemeine Optimierungssoftware (wie z. B. COSYTEC, LINDO, CPLEX, MIP, ...) ist für die Lösung allgemeiner Optimierungsprobleme konzipiert und somit nicht besonders gut an spezielle Eigenschaften des zu lösenden Optimierungsproblems anpaßbar. Vielmehr werden in diesen Optimierungs-Softwarepaketen im allgemeinen nur Branch & Bound-Verfahren angewendet, die auch ohne besondere Kenntnisse über das Optimierungsproblem die optimale Lösung finden. Will man die Optimierungssoftware an sein Problem anpassen, sind spezielle Kenntnisse in einer Softwarepaket-spezifischen Programmiersprache notwendig.

Daher liegt die Idee nahe, die Optimierungsverfahren für $PART^{\text{opt}}$ komplett selbst zu programmieren. Auf diese Weise können die speziellen Eigenschaften des Systemgraphen im Verfahren berücksichtigt werden und das Verfahren kann relativ einfach in bestehende Entwurfswerkzeuge integriert werden.

Für viele der hier vorgestellten Verfahren werden im Internet Software-Bibliotheken, bzw. der Source-Code zur Anpassung an eigene Probleme zur Verfügung gestellt. Eine Aufstellung der Adressen findet sich in Tabelle 15.1.

15.3 Ausblick

Nachdem nun die verschiedenen Optimierungsverfahren, die auf $PART^{\text{opt}}$ anwendbar sind, aufgezeigt wurden, stellt sich die Frage, welches Verfahren für einen bestimmten gegebenen Systemgraphen am besten geeignet ist. Besonders wichtig wird diese Frage bei der Anwendung von Heuristiken, denn die Qualität ihrer Ergebnisse hängt sehr stark von der Graphenstruktur, der Zielfunktion und den vorhandenen Implementierungsmöglichkeiten der Blöcke und Wandler ab.

Der worst-case wird durch die Aussage über die Nichtapproximierbarkeit von $PART^{\text{opt}}$ bestimmt: Für jeden polynomiellen Approximationsalgorithmus kann ein Systemgraph konstruiert werden, der praktisch nicht approximierbar ist.

Interessanter ist aber der average-case: Wie gut optimieren die Heuristiken „normale“ Systemgraphen? Für die Beantwortung dieser Frage sind ausführliche, repräsentative Benchmarks notwendig. Um einen Benchmark

Lineare Programmierung	SoPlex (`www.zib.de/Optimization/Software/Soplex`)
	lp_solve (`ftp.ics.ele.tue.nl/pub/lp_solve`)
Branch & Bound	ABACUS (`www.informatik.uni-koeln.de/ls_juenger/projects/abacus.html`)
Simulated Annealing	ASA (`www.ingber.com`)
Tabu-Search	Beispiel-Programm (`rtm.science.unitn.it/~battiti/reactive.html`)
Genetische Algorithmen	EPO (`telemann.ltt.rwth-aachen.de/epo`)
	GAlib (`lancet.mit.edu/ga`)

Tabelle 15.1: Adressen für Tools zu den Optimierungsverfahren

ausführen zu können, bedarf es jedoch einer Implementierung des Optimierungsverfahrens und einer Auswahl realistischer Systemgraphen, auf denen die Optimierungsverfahren getestet und bewertet werden können.

Ziel dieser Arbeit war es, einen Überblick über Optimierungsverfahren zu geben, die sich auf $PART^{\mathrm{opt}}$ anwenden lassen. Aufgrund der Breite dieser Aufgabe war eine Implementierung der gefundenen Verfahren nicht vorgesehen und konnte auch nicht vorgenommen werden.

Außerdem sind bisher noch keine hybriden Systeme vorhanden, für deren Funktionseinheiten die nichtfunktionalen Eigenschaften bekannt sind, und die somit auf einen Systemgraphen übertragbar wären. Mit diesem Thema beschäftigt sich unter anderem zur Zeit die Projektgruppe „Entwurf hybrider Systeme“ des Lehrstuhls Technische Informatik an der J. W. G.-Universität Frankfurt/Main. Das Ziel ist es, daß das Entwurfswerkzeug die nichtfunktionalen Eigenschaften selbständig berechnet und dem Systemgraphen zur Verfügung stellt.

Für eine abschließende Bewertung der einzelnen Optimierungsverfahren fehlen also noch

- die Implementierung der Verfahren und
- die Erzeugung realistischer, repräsentativer Systemgraphen als geeignete Basis für einen Benchmark.

Sobald diese beiden wesentlichen Voraussetzungen realisiert sind, können

aufgrund der automatischen Optimierung des hybriden Systems mit Hilfe des Systemgraphens deutlich bessere Implementierungen erwartet werden, als der Designer mit einer manuellen intuitiven Optimierung erreichen kann.

Abbildungsverzeichnis

Algorithmenverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] E. H. L. Aarts und J. Korst. *Simulated annealing and Boltzmann machines: a stochastic approach to combinatorial optimization and neural computing.* John Wiley, Chichester, 1989. ISBN 0-471-92146-7.

[2] R. Battiti. Reactive Search: Toward Self-Tuning Heuristics. In V. J. Rayward-Smith, I. H. Osman, C. R. Reeves und G. D. Smith, Herausgeber, *Modern Heuristic Methods*, Kapitel 4, Seiten 61–83. John Wiley, Chichester, 1996.

[3] R. Battiti und G. Tecchiolli. The Reactive Tabu Search. *ORSA Journal on Computing*, 1993.

[4] A. Bleck, M. Goedecke, S. A. Huss und K. Waldschmidt. *Praktikum des modernen VLSI-Entwurfs.* B.G. Teubner, Stuttgart, 1996. ISBN 3-519-02296-6.

[5] T. H. Cormen, C. E. Leiserson und R. L. Rivest. *Introduction to algorithms.* MIT Press, Cambridge, 1990. ISBN 0-262-03141-8.

[6] P. Crescenzi und V. Kann. A compendium of NP optimization problems. August 1998. URL `http://www.nada.kth.se/theory/compendium/`.

[7] W. Domschke. *Einführung in Operations Research.* Springer, Berlin, 1995. ISBN 3-540-60202-X.

[8] W. Domschke, R. Klein und A. Scholl. Taktische Tabus, Tabu Search — Durch Verbote schneller optimieren. *c't*, Seiten 326–332, 12/1996.

[9] W. Dück. *Diskrete Optimierung.* Vieweg, Braunschweig, 1977. ISBN 3-528-06826-4.

[10] W. Dück. *Optimierung unter mehreren Zielen.* Vieweg, Braunschweig, 1978. ISBN 3-528-06842-6.

[11] B. Eschermann. *Funktionaler Entwurf digitaler Schaltungen.* Springer, Berlin, 1993. ISBN 3-540-56788-7.

[12] G. Fandel. *Optimale Entscheidung bei mehrfacher Zielsetzung.* Springer, Berlin, 1972. ISBN 3-540-06064-2.

[13] G. Fischer. *Lineare Algebra.* Vieweg, Braunschweig, 1986. ISBN 3-528-57217-5.

[14] G. Fischer. *Analytische Geometrie.* Vieweg, Braunschweig, 1992. ISBN 3-528-57235-3.

[15] M. R. Garey und D. S. Johnson. *Computers and Intractability.* W. H. Freeman, San Francisco, 1979. ISBN 0-7167-1044-7.

[16] C. Grimm, F. Heuschen und K. Waldschmidt. Top-Down Design of Mixed-Signal Systems with KANDIS. In *Workshop on System Design Automation (SDA '98).* Dresden, Deutschland, März 1998.

[17] C. Grimm und K. Waldschmidt. Hybride Datenflußgraphen — Ein graphisches Modell zur Darstellung und Optimierung hybrider Systeme. Technischer Bericht, J. W. Goethe-Universität, 08/98.

[18] G. D. Hachtel und F. Somenzi. *Logic Synthesis and Verification Algorithms.* Kluwer Academic Publishers, Boston, 1998. ISBN 0-7923-9746-0.

[19] F. Heuschen, H. Schmitt und K. Waldschmidt. Formale Modellierung der Partitionierung eines gemischt analog/digitalen Systems. In *Gemeinsamer Workshop der GI/ITG/GME – Fachgruppen „Methoden des Entwurfs und der Verifikation digitaler Schaltungen und Systeme“ und „Beschreibungssprachen und Modellierung von Schaltungen und Systemen“.* Braunschweig, Februar 1999.

[20] D. S. Hochbaum, Herausgeber. *Approximation algorithms for NP-hard problems.* PWS Publishing, Boston, 1997. ISBN 0-534-94968-1.

[21] D. E. Knuth. *The Art of Computer Programming*, Band 3: Sorting and Searching. Addison-Wesley, Massachusetts, 1973. ISBN 0-201-03803-X.

[22] O. Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 1998. URL `http://www.mi.informatik.uni-frankfurt.de/people/kullmann/3neu.ps`.

[23] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* John Wiley, Chichester, 1990. ISBN 0-471-92838-0.

[24] K. Mehlhorn. *Graph Algorithms and NP-Completeness.* Springer, Berlin, 1984. ISBN 3-540-13641-X.

[25] H. Mühlenbein. Genetic Algorithms. In E. Aarts und L. Lenstra, Herausgeber, *Local Search in Combinatorial Optimization*, Seiten 131–171. John Wiley, Chichester, 1997.

[26] G. L. Nemhauser und L. A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley, New York, 1988. ISBN 0-471-82819-X.

[27] T. Ottmann und P. Widmayer. *Algorithmen und Datenstrukturen.* Spektrum, Heidelberg, 1996. ISBN 3-8274-0110-0.

[28] R. Paul. *Elektrotechnik und Elektronik für Informatiker.* B.G. Teubner, Stuttgart, 1994. ISBN 3-519-02126-9.

[29] J. Pearl. *Heuristics: intelligent search strategies for computer problem solving.* Addison-Wesley, London, 1984. ISBN 0-201-05594-5.

[30] A. Rapoport. *Decision theory and decision behaviour: normative and descriptive approaches.* Kluwer Academic Publishers, Dordrecht, 1989. ISBN 0-7923-0297-4.

[31] G. Schnitger. Skript zur Vorlesung „Effiziente Algorithmen". WS 96/97.

[32] A. Scholl, G. Krispin, R. Klein und W. Domschke. Branch and Bound — Optimieren auf Bäumen: je beschränkter, desto besser. *c't*, Seiten 336–345, 10/1997.

[33] A. Schrijver. *Theory of linear and integer programming.* John Wiley, 1986. ISBN 0-471-90854-1.

[34] W. M. Spears, K. A. De Jong, T. Bäck, D. B. Fogel und H. de Garis. An Overview of Evolutionary Computation. In *Proceedings of the 1993 European Conference on Machine Learning.* 1993.

[35] H. H. Weber. *Einführung in Operations Research.* Akademische Verlagsgesellschaft, Frankfurt, 1972. ISBN 3-400-00186-4.

[36] M. Weber. *Entscheidungen bei Mehrfachzielen: Verfahren zur Unterstützung von Individual- und Gruppenentscheidungen.* Gabler, Wiesbaden, 1983. ISBN 3-409-82820-6.

[37] I. Wegener. *Theoretische Informatik.* B.G. Teubner, Stuttgart, 1993. ISBN 3-519-02123-4.

[38] L. A. Wolsey. *Integer programming.* John Wiley, New York, 1998. ISBN 0-471-28366-5.

[39] P. L. Yu. *Multiple-Criteria Decision Making: Concepts, Techniques, and Extensions.* Plenum Press, New York, 1985. ISBN 0-306-41965-3.

[40] H. Zimmermann und L. Gutsche. *Multi-Criteria Analyse.* Springer, Berlin, 1991. ISBN 3-540-54483-6.